CMU-ITC-84-005

Users, Groups, and Access Lists

An Implementor's Guide

CMU-ITC-84-005
06 August 84 15:12

M. Satyanarayanan

Information Technology Center
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

Table of Contents

Preface

This document is a reference guide to the protection mechanism in VICE. It is expected that the typical reader is either:

- an implementor of a VICE subsystem with controlled access to offered services.
- or, an implementor of a user-friendly interface (on a workstation) to query and manipulate the protection domain in VICE.

End-users are not expected to use the facilities described here directly.

Two related facilities are described in this manual:

A VICE server, called the ***Protection Server****.*
An instance of this server runs on each cluster server and handles remote requests via an RPC interface. This server deals with queries and changes to the protection domain.

A library of C subroutines for dealing with access lists.
This library is linked in with each VICE server which wishes to enforce protection on the objects it is responsible for. The VICE File Server will be the first user of this package, using it to enforce protection on files. The Protection Server will itself use this package to protect its long-term data structures. Other VICE servers, such as Database Servers and Print Servers, may use this package too.

Note that this is a preliminary definition. Changes are likely to be made in the light of implementation and usage experience.

1. Key Concepts

The fundamental protection question is "Can agent *X* perform operation *Y* on object *Z*?"

The set of agents about whom such a question can be asked is referred to as the VICE ***Protection Domain***. The set of operations and the set of objects are specific to each VICE subsystem; however there is only one protection domain in VICE.

For each object, an ***Access List*** is a function that maps the protection domain to the set of operations valid for that object.

The protection domain is composed of ***Users*** and ***Groups***:

- From our point of view, a user is a an entity uniquely identified by a character string called its ***UserName***. Nothing further is assumed about a user. Philosophically, a user is an entity that is capable of authenticating itself to VICE, can be held responsible for its actions, and can be charged for resource consumption. Typically a user is a human being.

- A group is a set of other groups and users, and is uniquely identified by a ***GroupName***. A group possesses certain ***Naming*** and ***Membership*** properties which are central to the protection mechanism.

1.1. Naming

A username is an arbitrary alphameric string of length less than PRS_MAXNAMELEN. Upper-lower case distinctions are ignored.

Associated with each group is a user called its ***Owner***. Typically the owner is the creator of the group; however ownership of a group may be transferred between mutually consenting users.

A groupname is a two-tuple of the form *Prefix : Suffix*, where the prefix is the owner's username and the suffix is an arbitrary string of alphameric characters. No interpretation is placed on the suffix of a groupname. However the character "." is allowed in the suffixes and may be used to superimpose structure on groupnames. For example, "Bovik:Friends", "Bovik:Friends.CatLovers", and "Bovik:Friends.CatHaters" could be the names of three groups owned by user "Bovik", with the latter two being disjoint refinements of the first. It should be emphasised that such an interpretation of groupnames is purely by convention; the protection system treats all groups of a user as unrelated entities. The maximum length of a groupname is PRS_MAXNAMELEN[1].

---

[1] Obviously a user with a name of length PRS_MAXNAMELEN can own no groups!

Initially, there is a single user called "System." System is an omnipotent user: no protection checks apply to it. In this regard System fulfils the same role that a superuser fulfils in Unix systems. It should be obvious that only highly trustworthy system administrators should be capable of authenticating themselves as System; more so than in Unix systems because of the size and scale of VICE. The access list mechanism provides a way to delegate most administrative responsibility without all administrator's being capable of authenticating themselves as System.

The names of groups owned by System can have their prefixes omitted. Thus "System:AllStudents" has the alias "AllStudents." To avoid ambiguity, usernames must be distinct from the suffixes of the groups owned by System.

Two names have special semantics. The username "Anonymous" stands for "anyone who is not an authenticated user of VICE." The groupname "System:AnyUser" has all users of VICE (except Anonymous) as its implicit members: users do not have to be explicitly added to this group. One has to have a username to be a member of System:AnyUser. These names can be used in access lists to specify very liberal access policies. Certain restrictions apply to these names: Anonymous cannot be made a member of any group; AnyUser cannot be made to have any explicit members, nor can it be made a member of any group.

1.2. Membership

As mentioned earlier, a group is essentially a set whose elements are users and other groups. The constituent elements of a group are referred to as its ***Members***. The ***IsAMemberOf*** relation holds between a user or group *X* and a group *G*, if and only if *X* is a member of *G*. For each *X*, the reflexive, transitive closure of the IsAMemberOf relation defines a subset of the protection domain. This subset is referred to as the ***Current Protection Subdomain (CPS)*** of *X*, and plays a crucial role in the protection mechanism. Less formally, the CPS is the set of all groups that *X* is a member of, either directly or indirectly; it also includes *X* itself.

1.3. Rights

A ***Right*** is a bit position in a 32-bit integer mask. No further interpretation of rights is imposed by the access list package.

Each user of the access list package has to do the following:

- Construct a C header file with symbolic definitions for rights.
- Define a mapping between rights and operations on the class of objects being protected.

This mapping is not relevant to the access list package itself, but is needed to interpret the result of a protection check performed using the package.

As an example, consider a hypothetical VICE server which implements a classified bulletin board. Entries on this bboard are of security rating *Unclassified*, *Secret*, or *TopSecret*. The server recognizes six rights, with symbolic names *ReadUnclassified*, *WriteUnclassifed*, *ReadSecret*, *WriteSecret*, *ReadTopSecret*, and *WriteTopSecret*. These rights occupy bit positions 0 to 5 of a 32-bit integer mask. The interpretation of these rights is obvious.

In the above example, one could have assumed that anyone who could read a notice at a certain security level could also post notices at that level. In that case there would only be three rights, symbolically referred to as *AccessUnclassified*, *AccessSecret*, and *AccessTopSecret*, corresponding to mask bit positions 0, 1, and 2. The point of this example is that the choice of rights and their semantics is a matter for individual VICE servers to decide. The only restriction placed by the access list package is that there can be at most 32 rights associated with each object.

The Protection Server is a VICE server whose protected objects are users and groups. It recognizes two rights: PRS_EXAMINE and PRS_MANIPULATE. If one possesses PRS_EXAMINE rights on a user or group, one is allowed to execute those operations which return membership information about that user or group. Possession of PRS_MANIPULATE rights allows deletion, renaming and modification of the membership status. Table 3-1 specifies the exact semantics of these rights.

1.4. Access Lists

An entry in an access list is a two-tuple of the form *(User or Group, Rights Mask)*. An access list contains two lists of such entries: one called a ***Positive Rights List*** and the other a ***Negative Rights List***. An entry of the form *(X, R)* in a positive rights list implies that user or group X possesses the set of rights defined by mask R. In a negative rights list it implies that X is denied the rights defined by R. If the entry is present in both lists, the negative rights override, and X is denied R.

Negative rights are a means to specify rapid, selective, revocation of rights on sensitive objects to specific users or groups. This is intended as a mechanism for handling emergencies. Usually a negative rights list will be empty; a user or group will be denied rights to an object because of the absence of an appropriate entry in the Positive Rights List of the object.

The total rights possessed by a user *U* on an object *O* is the union of all the rights that the members of *U*'s CPS possess on *O*. In other words, *U* possesses the maximal rights that is collectively possessed

by all of the groups that he is a direct or indirect member of. Suppose *A* is an arbitrary access list and *C* is the CPS of *U*. The rights possessed by *U* on *O* is determined as follows:

1. Let *M* and *N* be rights masks, initially empty.
2. For each element of *C*, if there is an entry in the positive rights list of *A*, OR *M* with the rights portion of the entry.
3. For each element of *C*, if there is an entry in the negative rights list of *A*, OR *N* with the rights portion of the entry.
4. Remove from *M*, those rights which are specified in *N*.
5. *M* now specifies the rights that *U* possesses on *O*.

The access list package supports two physical representations for access lists: an internal format and an external format. The internal format stores integer representations of user and group names, and is designed for compactness and rapid access checks. It is the format in which access lists are represented on secondary storage and used in VICE servers. The external format represents user and group names as character strings, and is intended to be used by clients of VICE servers.

2. The Access List Package

The access list package consists of a C header file and a library of subroutines to deal with access lists. The package is designed so that the user (typically a VICE server) is completely insulated from the implementation details of the access list mechanism. The package also contains routines to read access lists from and to write them to Unix files.

2.1. Data Structures

The data structures used in this package are defined in the header file "al.h", and are described below.

```
#define AL__VERSION "$Header$"

typedef
  struct
    {
    int Id;                          /*internally-used ID of user or group*/
    int Rights;                      /*mask*/
    }
  AL__AccessEntry;
/*
The above access list entry format is used in VICE
*/
```

```
#define AL__ALISTVERSION  1          /*Identifies current format of access lists*/
typedef
  struct
    {
    int MySize;                      /*size of this access list in bytes, including MySize itself*/
    int  Version;                      /*to deal with upward compatibility in ancient files; <= AL_ALISTVERSION*/
    int  TotalNoOfEntries;             /*no of slots in ActualEntries[]; redundant, but used for convenience*/
    int PlusEntriesInUse;            /*stored forwards from ActualEntries[0]*/
    int MinusEntriesInUse;           /*stored backwards from ActualEntries[TotalNoOfEntries-1]*/
    AL__AccessEntry ActualEntries[1]; /*Actual array bound is TotalNoOfEntries*/
    }
  AL__AccessList;
/*
Used in VICE. This is how acccess lists are stored on secondary storage.
*/
```

```
typedef
  struct
    {
    RPC__Integer NoOfPlusEntries;
    RPC__Integer NoOfMinusEntries;
    RPC__Integer OffsetOfMinusEntries;  /*from ActualEntries[0].*/
    RPC__String ActualEntries;          /*See format description below*/
    }
  AL__ExternalAccessList;
```

```
/*
Used in dealings with clients via RPC.  Input and output RPC parameters will typically contain this data structure as the
SeqBody of an RPC_CountedBS.  The ActualEntries field consists of two lists: the first for the Plus entries and the second for
the Minus entries.  Each entry consists of a username or groupname followed by a decimal number representing the rights
mask for that name.  Each entry in the list looks as if it had been produced by printf() using a format list of "%s\t%d\n".
*/
```

2.2. Routines

The library "libal.a" contains the following routines to manipulate access lists:

```
/*
NOTE:  Unless otherwise specified, these routines return 0 on success and -1 on failure of any kind.

The access list package has routines to allocate, free, byte-swap and reverse byte-swap access lists and CPSs in internal and
external format.  Don't clobber the bytes preceding the allocated data structures --- the storage allocator uses this information.
*/

int AL_NewAlist(IN MinNoOfEntries, OUT Al)
  int MinNoOfEntries;
  AL_AccessList **Al;
  {
  /*
  Creates an access list capable of holding at least MinNoOfEntries entries.
  Returns 0 on success; aborts if we run out of memory.
  */
  }

int AL_FreeAlist(INOUT Al)
  AL_AccessList **Al;
  {
  /*
  Releases the access list defined by Al.
  Returns 0 always.
  */
  }

int AL_htonAlist(INOUT Al)
  AL_AccessList *Al;
  {
  /*
  Converts the access list defined by Al to network order.
  Returns 0 always.
  */
  }

int AL_ntohAlist(INOUT Al)
  AL_AccessList *Al;
  {
  /*
  Converts the access list defined by Al to host order.
  Returns 0 always.
  */
  }

int AL_NewExternalAlist(IN MinNoOfEntries, OUT R)
  int MinNoOfEntries;
  RPC_CountedBS **R;
```

```
{
/*
On successful return, R defines an external access list big enough
  to hold MinNoOfEntries full-sized entries.
Returns 0 on success; aborts if insufficient memory.
NOTE:  The caller may set the SeqLen field of the RPC_CountedBS to the number of bytes
     actually used. Then the assumption about full-sized entries only means that the
     malloc()ed storage is larger than typically necessary; RPC does not have to see
     the excess bytes.  AL_FreeExternalAlist() deals with this properly.
*/
}

int AL_FreeExternalAlist(INOUT R)
  RPC_CountedBS **R;
  {
  /*
  Releases the external access list defined by R.
  Returns 0 always.
  */
  }

int AL_htonExternalAlist(INOUT EA)
  AL_ExternalAccessList *EA;
  {
  /*
  Converts the external access list defined by EA to network order.
  Returns 0 always.
  */
  }

int AL_ntohExternalAlist(INOUT EA)
  AL_ExternalAccessList *EA;
  {
  /*
  Converts the external access list defined by EA to host order.
  Returns 0 always.
  */
  }

int AL_NewCPS(IN MinNoOfEntries, OUT ICPS)
  int MinNoOfEntries;
  PRS_InternalCPS **ICPS;
  {
  /*
  On successful return, ICPS defines an internal CPS which is
    capable of holding at least MinNoOfEntries entries.
  Returns 0 on success; aborts if we run out of memory.
  */
  }

int AL_FreeCPS(INOUT C)
  PRS_InternalCPS **C;
  {
  /*
  Releases the internal CPS defined by C.
  Returns 0 always.
  */
  }

int AL_htonCPS(INOUT C)
```

```
  PRS_InternalCPS *C;
  {
  /*
  Converts the CPS defined by C to network byte order.
  Returns 0.
  */
  }

int AL_ntohCPS(INOUT C)
  PRS_InternalCPS *C;
  {
  /*
  Converts the CPS defined by C to host byte order.
  Returns 0 always.
  */
  }

int AL_NewExternalCPS(IN MinNoOfEntries, OUT R)
  int MinNoOfEntries;
  RPC_CountedBS **R;

  {
  /*
  On successful return, R defines a newly-created external CPS which is
    big enough to hold MinNoOfEntries full-sized entries.
  Returns 0 on success; aborts if insufficient memory.
  NOTE:
    The caller may set the SeqLen field of the RPC_CountedBS to the number of bytes
    actually used. Then the assumption about full-sized entries only means that the
    malloc()ed storage is larger than typically necessary; RPC does not have to see
    the excess bytes. AL_FreeExternalCPS deals with this properly.
  */
  }

int AL_FreeExternalCPS(INOUT R)
  RPC_CountedBS **R;
  {
  /*
  Releases the external access list defined by R.
  Returns 0 always.
  */
  }

int AL_htonExternalCPS(INOUT EC)
  PRS_ExternalCPS *EC;
  {
  /*
  Converts the external CPS defined by EC to network byte order.
  Returns 0 always.
  */
  }

int AL_ntohExternalCPS(INOUT EC)
  PRS_ExternalCPS *EC;
  {
  /*
  Converts the external CPS defined by EC to host byte order.
  Returns 0 always.
  */
  }
```

```
int AL__Externalize(IN Alist, OUT ExternalRep)
  AL__AccessList *Alist;
  RPC__CountedBS **ExternalRep;
  {
  /*
  Converts the access list defined by Alist into the newly-created
    external access list in ExternalRep.
  Non-translatable Ids are coverted to their Ascii integer representations.
  Returns 0 always.
  */
  }

int AL__Internalize(IN ExternalRep, OUT Alist)
  AL__ExternalAccessList *ExternalRep;
  AL__AccessList **Alist;
  {
  /*
  On successful return, Alist will define a newly-created access list
    corresponding to the external access list defined by ExternalRep.
  Returns 0 on successful conversion.
  Returns -1 if ANY name in the access list is not translatable.
  */
  }

int AL__CheckRights(IN Alist, IN CPS, OUT WhichRights)
  AL__AccessList *Alist;
  PRS__InternalCPS *CPS;
  int *WhichRights;
  {
  /*
  Returns in WhichRights, the rights possessed by CPS on Alist
  */
  }

int AL__Initialize(IN Version, IN pdbFile, IN pcfFile)
   char *Version;
   char *pdbFile;
   char *pcfFile;
  {
  /*
  Initializes the access list package.
  Version should always be AL__VERSION.
  pdbFile is a string defining the protection database file; set to NULL for default.
  pcfFile is a string defining the protection configuration file; set to NULL for default.
  */
  }

int AL__NameToId(IN Name, OUT Id)
  char *Name;
  int *Id;
  {
  /*
  Translates the username or groupname defined by Name to Id.
  Returns 0 on success, -1 if translation fails.
  */
  }

int AL__IdToName(IN Id, OUT Name)
  int Id;
```

```
  char Name[1 + PRS_MAXNAMELEN];
  {
  /*
  Translates Id and returns the corresponding username or groupname in Name.
  Returns 0 on success, -1 if Id is not translatable.
  */
  }

int AL_GetInternalCPS(IN Id, OUT ICPS)
  int Id;
  PRS_InternalCPS **ICPS;
  {
  /*
  On successful return, ICPS defines a newly-created data structure,
    corresponding to the internal CPS of Id.
  Return 0 on success; -1 if Id is not a valid user or group id.
  */
  }

int AL_GetExternalCPS(IN Id, OUT ECPS)
  int Id;
  RPC_CountedBS **ECPS;
  {
  /*
  On successful return, ECPS defines a newly-created data structure,
    corresponding to the external CPS of Id.
  Return 0 on success; -1 if Id is not a valid user or group id.
  */
  }

int CaseFoldedCmp(IN s1, IN s2)
  char *s1, *s2;
  {
  /* same as strcmp() except that case differences are ignored */
  }
```

2.3. Examples

As an example of how these routines may be used, consider the following examples modelled on the VICE File Server:

```
#include <rpc/rpc.h>
#include <prs/prs.h>
#include <prs/al.h>
#include <prs/prs_fs.h>

PRS_InternalCPS *ThisUser;          /* Initialized after connection to point to this user's CPS */


int Fetch(ViceFileName)
  char *ViceFileName;
  {
```

```
  AL_AccessList *AI;
  int MyRights;

  Obtain the access list AI, to be used in the protection check from the parent directory of ViceFileName.
  AL_CheckRights(AI, ThisUser, &MyRights);

  if (PRS_FILEREAD & MyRights = = 0)
    return(/* failure indication */);

  Do actual file transmission here
  }


int Store(ViceFileName)
  char *ViceFileName;
  {
  Identical to Fetch; except:
    use PRS_FILEINSERT if you want to allow only creation of new files,
    use (PRS_FILEWRITE|PRS_FILEINSERT) if you want to allow writing new or existing files.
  }


int GetFileStat(ViceFileName)
  char *ViceFileName;
  {
  RPC_CountedBS *ExtRep;
  AL_AccessList *AI;
  int MyRights;

  Obtain the access list AI from the parent directory of ViceFileName
  PRS_CheckRights(AI, ThisUser, &MyRights);

  if (PRS_FILELOOKUP & MyRights = = 0)
    return(failure indication);

  AL_Externalize(AI, &ExtRep) < 0)

  Now ExtRep can be sent to the client, along with other file status info

  AL_FreeExternalAlist(ExtRep);
  }

int SetFileStat(ViceFileName)
  char *ViceFileName;
  {
  RPC_CountedBS *ExtRep;
  AL_AccessList *NewAI, *OldAI;

  Obtain the access list OldAI from the parent directory of ViceFileName

  PRS_CheckRights(OldAI, ThisUser, &MyRights);

  if (PRS_FILEWRITE & MyRights = = 0)
    return(failure indication);

  Obtain client-supplied ExtRep

  if (AL_Internalize(ExtRep, &NewAI) < 0)
    return(failure indication);
```

```
Write out the access list NewAl to the parent directory of ViceFileName

 do other SetFileStat() processing
PRS_FreeAlist(NewAl);
}
```

3. Protection Server RPC Calls

This chapter describes the primitives of the Protection Server. The calls are described in a format that assumes that you are using the VICE RPC mechanism to make remote procedure calls to the Protection Server. The types of the arguments specified in these calls are the types defined in the RPC manual [Satyanarayanan84a]. The header file "al.h" contains the definitions for the symbolic constants used in the descriptions.

It is assumed that all connections to the Protection Server are secure, authenticated, RPC connections. The username of a client is the value of the ClientID parameter in the corresponding RPC_Bind call.

During the implementation and refinement of this subsystem, some restrictions may be placed on the primitives:

1. Each VICE cluster server will have a Protection Server running on it. Initially one of these will be a master, and is the only one which will service primitives that change the protection domain. Such requests will result in a return code of PRS_FAIL from all the other Protection Servers. All other requests (i.e., queries) may be directed to any Protection Server. The descriptions of the calls indicate whether they can only be serviced by the master.

2. There will be limitation on the membership properties of groups. The purpose of this restriction is allow a quick implementation without spending a major amount of time on efficient transitive closure algorithms. These limitations will be specified in a later release of this document. Most probably groups may only be allowed users as members; they may not have other groups as members.

The header file "prs.h" contains definitions for the rights PRS_EXAMINE and PRS_MANIPULATE, and definitions for the data types involved in calls to protection server:

```
#define PRS_VERSION "$Header$"

#define PRS_MAXNAMELEN 100          /*Maximum length of group and user names*/

#define PRS_SYSTEMID 100            /*Userid of System*/

#define PRS_ANONYMOUSID 101         /*Userid of the fake user Anonymous*/

#define PRS_ANYUSERID -101          /*GroupId of System:AnyUser*/

#define PRS_PDBNAME "/usr/local/lib/vice.pdb"
                                    /*default location of protection data base*/

#define PRS_PCFNAME "/usr/local/lib/vice.pcf"
                                    /*default location of configuration file*/
```

```
typedef
  struct
    {
    int NoOfEntries;                    /*in IdList*/
    int IdList[1];                      /*Actual bound is NoOfEntries. List of ids in this subdomain.
                                        Sorted in ascending order*/
    }
  PRS_InternalCPS;
/*
Used only in VICE. Typically obtained via access list package routine AL_GetInternalCPS.
*/
```

```
typedef
  struct
    {
    RPC_Integer NoOfEntries;            /*number of names in NameList*/
    RPC_String NameList;                /*list of blank separated names in this subdomain */
    }
  PRS_ExternalCPS;
/*
Used in dealings with clients. Typically transmitted as the SeqBody of an RPC_CountedBS parameter.
*/
```

The rights requirements for various Protection Server operations are specified in Table 3-1 below.

	System Only	PRS_EXAMINE	PRS_MANIPULATE
On Users			
PRS_NewUser	X		
PRS_DeleteUser			X
PRS_RenameUser			X
PRS_GetCPS		X	
PRS_ListDirectMembership		X	
PRS_GetProtection		X	
PRS_SetProtection			X
PRS_ListGroups		X	
On Groups			
PRS_NewGroup			
PRS_RenameGroup			X
PRS_DeleteGroup			X
PRS_ListDirectMembers		X	
PRS_ListDirectMembership		X	
PRS_GetCPS		X	
PRS_GetProtection		X	
PRS_SetProtection			X
PRS_AddToGroup			X
PRS_RemoveFromGroup			X

Note: System can always perform any operation.

Table 3-1: Rights Required for Protection Server Operations

PRS_GetCPS

Obtain CPS of user or group

Call:

```
int PRS_GetCPS( IN RPC_String Name, IN RPC_Integer Format,
                OUT RPC_BoundedBS Subdomain )
```

Parameters:

Name	Name of a user or group
Format	PRS_INTERNAL or PRS_EXTERNAL. VICE servers should request the internal format in order to use Subdomain in calls to the access list package. All other clients should request the external format.
Subdomain	The current protection subdomain of this user or group. Depending on what was specified for Format, this RPC_BoundedBS is to be interpreted as of type PRS_InternalCPS or PRS_ExternalCPS.

Completion Codes:

PRS_SUCCESS	All went well
PRS_NOACCESS	You do not have PRS_EXAMINE rights on Name
PRS_NOSUCHNAME	Name does not correspond to a user or group.
PRS_FAIL	Something else went wrong

Given a user or group name, this call returns its current protection subdomain. This is the reflexive, transitive closure of all the groups that this user or group is a member of.

PRS_NewUser

Create a new user

Call:

```
int PRS_NewUser( IN RPC_String UserName )
```

Parameters:

UserName	The name of the new user.

Completion Codes:

PRS_SUCCESS	All went well.
PRS_NOACCESS	You are not System.
PRS_DUPLICATENAME	A user (or a group belonging to System) is already called UserName.
PRS_FAIL	Something else went wrong

This call is used to add new users to the system. To use this call, you must be authenticated as System to the Protection Server.

May only be directed to the master Protection Server.

PRS_NewGroup

Create a new group

Call:

```
int PRS_NewGroup( IN RPC_String GroupName )
```

Parameters:

GroupName	Name of the new group.

Completion Codes:

PRS_SUCCESS	Created the new group.
PRS_NOACCESS	You were not System and the prefix of GroupName was not your user name.
PRS_DUPLICATENAME	There is already a group called GroupName.
PRS_FAIL	Something else went wrong

If you are not System, the prefix portion of the name must be your user name. System can create groups with any prefix. The newly created group has an empty access list. You and System always possess all rights on all your groups.

May only be directed to the master Protection Server.

PRS_DeleteUser

Get rid of a user

Call:

```
int PRS_DeleteUser( IN RPC_String UserName )
```

Parameters:

UserName	Name of the user to be deleted.

Completion Codes:

PRS_SUCCESS	All went well
PRS_NOACCESS	You are not System and you do not possess PRS_MANIPULATE rights on UserName.
PRS_NOSUCHNAME	UserName is not a valid user name.
PRS_NOTEMPTY	This user still has some groups.
PRS_FAIL	Something else went wrong

Removes the specified user. Prior to deletion, this user should have no groups. Use PRS_RenameGroup to preserve important groups which were created by this user and which continue to be of importance.

May only be directed to the master Protection Server.

PRS_DeleteGroup

Get rid of a group

Call:

```
int PRS_DeleteGroup( IN RPC_String GroupName )
```

Parameters:

GroupName	Name of the group to be deleted.

Completion Codes:

PRS_SUCCESS	All went well.
PRS_NOACCESS	You were not System, your user name did not correspond to the prefix of GroupName, and you did not possess PRS_MANIPULATE rights on GroupName.
PRS_NOSUCHNAME	There is no group with the specified name.
PRS_FAIL	Something else went wrong

A user can always delete any of his groups. System and any user with PRS_MANIPULATE rights on a group may also delete it.

May only be directed to the master Protection Server.

PRS_RenameUser

Change the name of a user

Call:

```
int PRS_RenameUser( IN RPC_String OldName,
                    IN RPC_String NewName )
```

Parameters:

OldName	What the user is currently known as.
NewName	What the user should be called in future.

Completion Codes:

PRS_SUCCESS	All went well.
PRS_NOACCESS	You are not System and you do not possess PRS_MANIPULATE rights on OldName.
PRS_DUPLICATENAME	A user or system group called NewName already exists
PRS_NOSUCHNAME	A user by name NewName does not exist.
PRS_FAIL	Something else went wrong

A user cannot rename himself, unless he possesses PRS_MANIPULATE rights on himself. All the groups belonging to this user are automatically renamed to have NewName as their prefix.

May only be directed to the master Protection Server.

PRS_RenameGroup

Change the name of a group.

Call:

```
int PRS_RenameGroup( IN RPC_String OldName,
                     IN RPC_String NewName )
```

Parameters:

OldName	What the group is currently known as.
NewName	What the group should be called in future.

Completion Codes:

PRS_SUCCESS	All went well
PRS_NOACCESS	You do not possess PRS_MANIPULATE rights on OldName, or the prefix of NewName is not your user name. System can perform arbitrary renaming of groups.
PRS_DUPLICATENAME	NewName is already the name of a group or a user.
PRS_NOSUCHNAME	There is no group by name OldName.
PRS_FAIL	Something else went wrong

Performs renaming of a group, leaving its membership properties unaltered. Ownership of a group may be transferred by this primitive: the new owner must request this rename and he should possess PRS_MANIPULATE rights on OldName. Unless you are System, NewName must have a prefix corresponding to your user name.

May only be directed to the master Protection Server.

PRS_ListDirectMembers

Enumerate the immediate members of a group

Call:

```
int PRS_ListDirectMembers( IN RPC_String GroupName,
                           OUT RPC_Integer HowMany,
                           OUT RPC_BoundedBS MemberList )
```

Parameters:

GroupName	Which group to enumerate
HowMany	The number of members in MemberList
MemberList	A series of RPC_Strings specifying the members of GroupName.

Completion Codes:

PRS_SUCCESS	All went well.
PRS_NOACCESS	You are not System and you do not possess PRS_EXAMINE rights on GroupName.
PRS_NOSUCHNAME	GroupName is not the name of a group.
PRS_FAIL	Something else went wrong

Gives you the immediate members of GroupName: i.e., no transitive closure is performed.

PRS_ListDirectMembership

Enumerate the immediate membership of a user or group

Call:

```
int PRS_ListDirectMembership( IN RPC_String Name,
                              OUT RPC_Integer HowMany,
                              OUT RPC_BoundedBS MembershipList )
```

Parameters:

Name	The name of a user or group
HowMany	The number of names in MemberList
MembershipList	The names of the groups which Name is an immediate member of.

Completion Codes:

PRS_SUCCESS	All went well.
PRS_NOACCESS	You are not System, and you do not possess PRS_EXAMINE rights on Name.
PRS_NOSUCHNAME	No user or group called Name exists.
PRS_FAIL	Something else went wrong

This primitive applies to both users and groups. Gives you the groups which this user or group is an immediate member of. It differs from the primitive PRS_GetCPS in that no transitive closure is performed here.

PRS_AddToGroup

Make a user or group a member of an existing group

Call:

```
int PRS_AddToGroup( IN RPC_String Name, IN RPC_String ToGroup
                  )
```

Parameters:

Name	The user or group to be added
ToGroup	The group which Name must be made a member of.

Completion Codes:

PRS_SUCCESS	All went well.
PRS_NOACCESS	You are not System and you do not possess PRS_MANIPULATE rights on ToGroup.
PRS_NOSUCHNAME	Either Name does not exist, or ToGroup is not the name of a group.
PRS_FAIL	Something else went wrong

No rights need be possesed on Name. The truly paranoid may consider this a shortcoming. If Name is already a member of ToGroup, this call is a nop.

May only be directed to the master Protection Server.

PRS_RemoveFromGroup

Remove a user or group from an existing group

Call:

```
int PRS_RemoveFromGroup( IN RPC_String Name,
                         IN RPC_String FromGroup )
```

Parameters:

Name	The user or group to be removed
FromGroup	The group from which Name must be removed

Completion Codes:

PRS_SUCCESS	All went well.
PRS_NOACCESS	You do not possess PRS_MANIPULATE rights on FromGroup.
PRS_NOSUCHNAME	Either Name does not exist, or FromGroup is not the name of a group, or Name is not currently a member of FromGroup.
PRS_FAIL	Something else went wrong

No rights need be possessed on Name. This is probably not a shortcoming even for the truly paranoid, since full control should be maintained by the owner of FromGroup.

May only be directed to the master Protection Server.

PRS_GetProtection

Obtain the access list of a user or group

Call:

```
int PRS_GetProtection( IN RPC_String Name,
                       OUT RPC_BoundedBS CurrentAccessList )
```

Parameters:

Name	The name of the user or group whose access list is desired
CurrentAccessList	In format PRS_ExternalAList.

Completion Codes:

PRS_SUCCESS	All went well
PRS_NOACCESS	You are not System and you do not possess PRS_EXAMINE rights on Name.
PRS_NOSUCHNAME	Name is not the name of a user or group.
PRS_FAIL	Something else went wrong

Returns the access list of Name in external format. Note that the external format is not intended to be directly viewed by humans; the caller may need to perform further formatting and beautification.

PRS_SetProtection

Specify a new access list for a user or group

Call:

```
int PRS_SetProtection( IN RPC_String Name,
                       IN RPC_BoundedBS NewAccessList )
```

Parameters:

Name	The user or group whose access list is to be changed
NewAccessList	In format PRS_ExternalAList

Completion Codes:

PRS_SUCCESS	All went well
PRS_NOACCESS	You are not System, and you do not possess PRS_MANIPULATE rights on Name.
PRS_NOSUCHNAME	Name is not the name of a user or group.
PRS_FAIL	Something else went wrong. Perhaps NewAccessList was of improper format.

Replaces the existing access list by a new one. For human interaction, the caller should interpose a front-end program which allows individual entries to be added or deleted.

May only be directed to the master Protection Server.

PRS_ListGroups

Enumerate the groups owned by a user

Call:

```
int PRS_ListGroups( IN RPC_String UserName,
                    OUTRPC_Integer HowMany,
                    OUT RPC_BoundedBS GroupList )
```

Parameters:

UserName	The user whose groups are to be enumerated
HowMany	The number of groups in GroupList
GroupList	A series of RPC_Strings, each specifying a group owned by UserName.

Completion Codes:

PRS_SUCCESS	All went well
PRS_NOACCESS	You are not System, and you do not possess PRS_Examine rights on UserName.
PRS_NOSUCHNAME	UserName is not the name of a user.
PRS_FAIL	Something else went wrong

Appendix I
Summary of Protection Server RPC Calls

Note: The numbers in square brackets indicate the page on which the call is described.

[18]

```
PRS_GetCPS(IN RPC_String Name, IN RPC_Integer Format,
OUT RPC_BoundedBS Subdomain)
```

[19]

```
PRS_NewUser(IN RPC_String UserName)
```

[20]

```
PRS_NewGroup(IN RPC_String GroupName)
```

[21]

```
PRS_DeleteUser(IN RPC_String UserName)
```

[22]

```
PRS_DeleteGroup(IN RPC_String GroupName)
```

[23]

```
PRS_RenameUser(IN RPC_String OldName, IN RPC_String NewName)
```

[24]

```
PRS_RenameGroup(IN RPC_String OldName, IN RPC_String NewName)
```

[25]

```
PRS_ListDirectMembers(IN RPC_String GroupName,
OUT RPC_Integer HowMany, OUT RPC_BoundedBS MemberList)
```

[26]

```
PRS_ListDirectMembership(IN RPC_String Name,
OUT RPC_Integer HowMany, OUT RPC_BoundedBS MembershipList)
```

[27]

```
PRS_AddToGroup(IN RPC_String Name, IN RPC_String ToGroup)
```

[28]

```
PRS_RemoveFromGroup(IN RPC_String Name,
IN RPC_String FromGroup)
```

[29]

```
PRS_GetProtection(IN RPC_String Name,
OUT RPC_BoundedBS CurrentAccessList)
```

[30]

```
PRS_SetProtection(IN RPC_String Name,
IN RPC_BoundedBS NewAccessList)
```

[31]

```
PRS_ListGroups(IN RPC_String UserName,
OUTRPC_Integer HowMany, OUT RPC_BoundedBS GroupList)
```

Appendix II
Usage Notes for the ITC SUN Systems

Two header files, /usr/local/include/prs/prs.h and /usr/local/include/prs/al.h should be included in all programs which use the access list package. You may also need to use the RPC header file, /usr/local/include/rpc/rpc.h

For each subsystem you will need a header file giving the interpretation of rights. A sample, for the VICE file system, is given in /usr/local/include/prs/prs_fs.h.

The access list package is in /usr/local/lib/libal.a.

The VICE protection database is in /usr/local/lib/vice.pdb, and the corresponding configuration file is in /usr/local/lib/vice.pcf.

The global integer variable AL__DebugLevel may be declared as an extern by users of the access list package. It is initialized to 0 and may be set to obtain debugging output; higher values yield more verbose output.