Authentication [3]

The RPC protocol includes a slot for authentication parameters on every call. The type of authentication used by the server and client determines the contents of the authentication parameters. A server can support the following types of authentication at once:

- `AUTH_DES` passes encrypted time-stamp information, allowing the client and server to perform mutual verification and authentication.
- `AUTH_KERB` passes encrypted Kerberos service ticket information, allowing the client and server to perform mutual verification and authentication.
- `AUTH_NULL` passes no authentication information (this is called *null authentication* and is the default).
- `AUTH_SHORT` is a shorthand form of passing UNICOS style credentials.
- `AUTH_UNIX` passes the UNICOS user ID, group ID, and group lists with each call.

Authentication types are fully described in appendix E, page 165.

The RPC package on the server authenticates every RPC, and, similarly, the RPC client package generates and sends authentication parameters. The authentication subsystem of the RPC package is open-ended; that is, numerous types of authentication are easy to support. This section covers UNICOS, Data Encryption Standard (DES), and Kerberos authentication.

Authentication of caller to service and vice versa are provided through call and reply messages. The call message has two authentication fields: credentials and verifier. The reply message has one authentication field, the response verifier.

The RPC protocol specification uses the XDR language described in appendix C, page 121. This protocol defines the authentication structure to be the following opaque type:

```
enum auth_flavor {
    AUTH_NULL    = 0,
    AUTH_UNIX    = 1,
    AUTH_SHORT   = 2,
    AUTH_DES     = 3,
    AUTH_KERB    = 4
    /* and more to be defined */
};

struct opaque_auth {
    union switch (enum auth_flavor) {
        default: string auth_body<00>;
    };
};
```

Any opaque_auth structure is an auth_flavor enumeration, followed by a counted string whose bytes are opaque to the RPC protocol implementation.

The interpretation and semantics of the data contained within the authentication fields are specified by individual, independent authentication protocol specifications.

If authentication parameters are rejected, the response message contains information stating why they were rejected.

Setting up authentication 3.1

This subsection describes the requirements for null, UNICOS, DES, and Kerberos authentication.

Null authentication requirements 3.1.1

Often, calls must be made in which the client does not have to verify the identity of the server, and the server does not have to know the identity of the client. In this case, the auth_flavor value (the discriminant of the opaque_auth's union) of the RPC message's credentials, verifier, and response verifier is AUTH_NULL. The bytes of the auth_body string are undefined. The string length should be 0. Null authentication is the default; when it is used, the server accepts and performs all service requests.

UNICOS authentication requirements 3.1.2

Sometimes a server might want to limit its services to a restricted set of users. One way of doing this is by using UNICOS authentication. When UNICOS authentication is used, the `auth_flavor` discriminant of the `opaque_auth` structure has the value `AUTH_UNIX`. The bytes of the `auth_body` can then be interpreted as an `authunix_parms` structure, as defined in appendix E, page 165. In addition to the user ID, other information, including a time stamp, a machine name, the user's group ID, and a list of groups to which the user belongs, is sent to the server. The server can use the data passed in the `authunix_parms` structure any way it chooses; that is, it can use any of the fields selectively to allow or disallow services.

Unfortunately, nothing prevents malicious users from writing whatever data they choose into the `authunix_parms` structure before it is sent to the server. Thus, it is very easy for a client to deceive a server into believing it is servicing either a different user or a user who has a different set of attributes.

DES authentication requirements 3.1.3

DES authentication provides stricter security than does UNICOS authentication, allowing a server to obtain a client user's identity with a very high degree of certainty. Moreover, the client user can verify the identity of the server with whom it is communicating. Although it is technologically possible to deceive even DES authentication, to do so on a local subnet requires a lot of computational resources.

DES authentication, which is sometimes called *secure RPC*, requires that the `keyserv`(8) daemon be running on both server and client machines. The administrator must have already assigned each secure RPC user a public key/secret key pair in the `publickey` database. DES users must then register themselves by using the `keyserv` process, either automatically, by logging in with `login`(1), or manually, with the `keylogin`(1) command.

Note: Because the network information service (NIS) manages the `publickey` database, NIS must be configured and running on the Cray Research system for DES authentication to work. Moreover, the Cray Research system must be in the same NIS domain as any host with whom DES authentication will be used.

Kerberos authentication requirements 3.1.4

Kerberos authentication uses encrypted Kerberos service tickets to provide more security than either UNICOS or DES authentication. Kerberos authentication requires that the Kerberos Enigma security package be installed on your system and that the site have an ONC+ license. See the *ONC+ Technology for the UNICOS Operating System*, publication SG–2169, for more information about ONC+. Users must obtain a Kerberos service ticket by using the `kinit`(1) command prior to using `AUTH_KERB` flavor RPC.

Servers using `AUTH_KERB` flavor RPC must register themselves with the authentication software by using the `svc_kerb_reg` library call. See the `kerberos_rpc`(3) man page for more information.

Client authentication 3.2

Suppose a caller creates a new RPC client handle, as in the following command:

```
CLIENT *clnt;
clnt = clntudp_create(address, prognum, versnum,
                wait, sockp)
```

By default, the type of authentication to use is set to `NULL`. However, the RPC client can choose to use UNICOS, DES, or Kerberos authentication by setting `clnt->cl_auth` after creating the RPC client handle.

For UNICOS authentication, the handle would be set as follows:

```
    clnt->cl_auth = authunix_create_default();
```

If an authentication failure occurs, you can use the following command instead:

```
clnt->cl_auth = authunix_create(host,uid,gid,len,aup_gids);
```

For DES authentication, the handle would be set as follows:

```
clnt->cl_auth = authdes_create(servername,credlife,&server_addr,key);
```

For Kerberos authentication, the handle would be set as follows:

```
clnt->cl_auth = authkerb_seccreate(server,instance,realm,window,timehost,
     status);
```

See appendix A, page 67, for descriptions of arguments for `authdes_create` and `authunix_create`.

Server authentication 3.3

People who develop RPC services have a more difficult time dealing with authentication issues than those implementing client applications, because the RPC package passes the service dispatch routine a request that has an arbitrary authentication style associated with it. Consider the fields of a request handle passed to a service dispatch routine:

```
/*
 * An RPC service request
 */
struct svc_req {
    u_long rq_prog;        /* service program number */
    u_long rq_vers;        /* service protocol vers_num */
    u_long rq_proc;        /* desired procedure number */
    struct opaque_auth
            rq_cred;            /* raw credentials from wire */
    caddr_t rq_clntcred;        /* credentials (read only) */
};
```

The `rq_cred` field is mostly opaque, except for the style of authentication credentials, as in the following:

```
/*
 * Authentication information.  Mostly opaque to the programmer.
 */
struct opaque_auth {
    enum_t oa_flavor;   /* style of credentials */
    caddr_t oa_base;    /* address of more auth stuff */
    u_int  oa_length;   /* not to exceed MAX_AUTH_BYTES */
};
```

The RPC package makes the following guarantee to the service dispatch routine:

- The request's `rq_cred` field is well-formed. Thus, the service implementer might inspect the request's `rq_cred.oa_flavor` field to determine which style of authentication the caller used. If `rq_cred.oa_flavor` is `AUTH_UNIX`, the pointer `rq_clntcred` could be cast to a pointer to an `authunix_parms` structure. If `rq_cred.oa_flavor` is `AUTH_DES`, the pointer `rq_clntcred` could be cast to a pointer to an `authdes_cred` structure. If `rq_cred.oa_flavor` is `AUTH_KERB`, the pointer `rq_clntcred` could be cast to a pointer to an `authkerb_clnt_cred` structure. If the style is not one of the styles that the RPC package supports, the service implementer might also want to inspect the other fields of `rq_cred`.

- The request's `rq_clntcred` field is either `NULL` or points to a well-formed structure that corresponds to a supported style of authentication credentials. If `rq_clntcred` is `NULL`, the service implementer might want to inspect the other (opaque) fields of `rq_cred` in case the service knows about a new type of authentication that the RPC package does not.

You can extend the remote users service example so that it computes results for all users except user ID 16, as follows:

```
nuser(rqstp, tranp)
   struct svc_req *rqstp;
   SVCXPRT *transp;
{
   struct authunix_parms *unix_cred;
   unsigned long nusers;
   struct authdes_cred *des_cred;
   struct authkerb_clnt_cred *authkerb_cred;
   int uid;
   int gid;
   int gidlen;
   int gidlist[10];
   /*
    * we don’t care about authentication for null proc
    */
   if (rqstp->rq_proc == NULLPROC) {
       if (!svc_sendreply(transp, xdr_void, 0)) {
           fprintf(stderr, “can’t reply to RPC call\n”);
           exit(1);
        }
        return;
   }
   /*
    * now get the uid
    */
   switch (rqstp->rq_cred.oa_flavor) {
   case AUTH_UNIX:
       unix_cred = (struct authunix_parms *)rqstp->rq_clntcred;
       uid = unix_cred->aup_uid;
       break;
```

(continued)

```
case AUTH_DES:
       des_cred =
           (struct authdes_cred *) rqstp->rq_clntcred;
       if (! netname2user(des_cred->adc_fullname.name,
           &uid, &gid, &gidlen, gidlist))
       {
           fprintf(stderr, “unknown user: %s\n”,
               des_cred->adc_fullname.name);
           svcerr_systemerr(transp);
           return;
       }
       break;
   case AUTH_KERB:
        authkerb_cred =
           (struct authkerb_clnt_cred  *)rqstp->rq_clntcred;
        if (!authkerb_getucred (rqstp, &uid, &gid, gidlen, gidlist)) {
               fprintf (stderr, “unknown user:%s\n”,
                  authkerb_cred->akc_fullname.pname);
               svcerr_systemerr(transp);
               return;
        }
        break;
   case AUTH_NULL:
   default:
       svcerr_weakauth(transp);
       return;
   }
   switch (rqstp->rq_proc) {
   case RUSERSPROC_NUM:
       /*
        * Explicitly disallow user with UID 16
        */
       if (uid == 16) {
           svcerr_systemerr(transp);
           return;
       }
       /*
        *  code here to compute the number of users
        *  and put in variable nusers
        */
       if (!svc_sendreply(transp, xdr_u_long, &nusers) {
           fprintf(stderr, “can’t reply to RPC call\n”);
           exit(1);
       }
       return;
   default:
       svcerr_noproc(transp);
       return;
   }
}
```

You should note the following points:

- It is customary not to check the authentication parameters associated with NULLPROC (procedure number 0). This allows any user to test for the presence of the server, simply by using rpcinfo(8).
- If the authentication parameter's type is not suitable for your service, the server should call svcerr_weakauth. For example, if the client sent credentials of type AUTH_UNIX or AUTH_NULL, and the server required credentials of type AUTH_DES, the server should call svcerr_weakauth.
- The service protocol itself should return the status for access denied. In the case of the previous example, the protocol does not have such a status; therefore, the service primitive svcerr_systemerr is called instead.

The last point underscores the relationship between the RPC authentication package and the services; RPC deals only with authentication and not with access control for individual services. Each service must implement its own access control policy and reflect that policy as return statuses in its protocol.

Record-marking standard
3.4

When RPC messages are passed on top of a byte stream protocol (such as TCP/IP), you should delimit one message from another to detect and possibly recover from user protocol errors. This is called *record marking* (RM). One RPC message fits into one RM record.

A record is composed of one or more record fragments. A *record fragment* consists of a 4-byte header, followed by 0 to 2^{31}–1 bytes of fragment data. The bytes encode an unsigned binary number; as with XDR integers, the byte order is from highest to lowest. The number encodes two values: a Boolean that indicates whether the fragment is the last fragment of the record (bit value 1 implies that the fragment is the last fragment) and a 31-bit unsigned binary value that is the number of bytes in the fragment's data. The Boolean value is the high-order bit of the header; the length is the 31 low-order bits.

Note: This record specification is not in XDR standard form.