Kerberos Authentication in Sun RPC

Carl Smith

The purpose of this report is to describe the Kerberos authentication flavor for Sun RPC and its use for NFS

Sun RPC *[RFC1057]* has an extensible authentication mechanism which allows new authentication flavors to be added from time to time. In SunOS 4.0 DES-based authentication was added (with the regrettable misnomer "secure RPC"). SunOS 5.0 added support for authentication based on Kerberos version 4 *[Steiner]*.

The specification for DES-based Sun RPC authentication has been available for quite some time and many NFS suppliers implement it *[ONC]*. The goal of this report is to describe the protocol for the Kerberos-based RPC authentication flavor in sufficient detail to allow independent, interoperable implementations to be built.

We assume the reader is already familiar with DES *[DES]*, the authentication framework for Sun RPC, and Kerberos. The specification of DES-based RPC authentication is briefly reviewed because Kerberos-based RPC authentication is very similar.

Authentication in Sun RPC

An authentication flavor consists of two parts: a *credential* (containing, in some form the user's identity), and a *verifier (*a method of proving the credential valid). Authentication flavors are named AUTH_something (AUTH_UNIX, AUTH_DES, and so on).

UNIX-style authentication (AUTH_UNIX, also known as AUTH_SYS) consists of a credential containing the user's user ID, group ID, groups to which the user belongs, and the name of the machine on which the credential was composed. There is no verifier. The overwhelming majority of NFS clients and servers use AUTH_UNIX.

DES-style authentication (AUTH_DES) uses a Diffie-Hellman public key cryptosystem *[Diffie]* to exchange DES session keys which are used for all subsequent encryption.

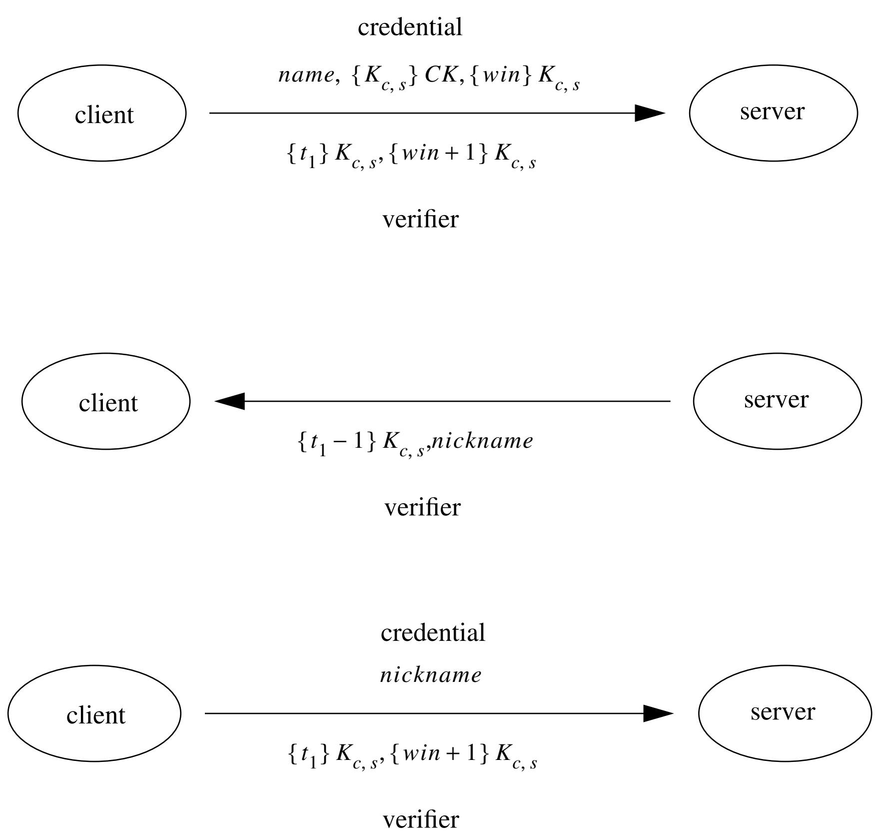
We follow the notation found in *[Steiner]*:

TABLE 1. Kerberos Notation

c	client
s	server
$T_{c,s}$	ticket for client **c** to use server **s**
$K_{c,s}$	session key for client **c** and server **s**
CK	Diffie-Hellman key for client **c** and server **s**
$\{x\}K$	**x** encrypted with key K

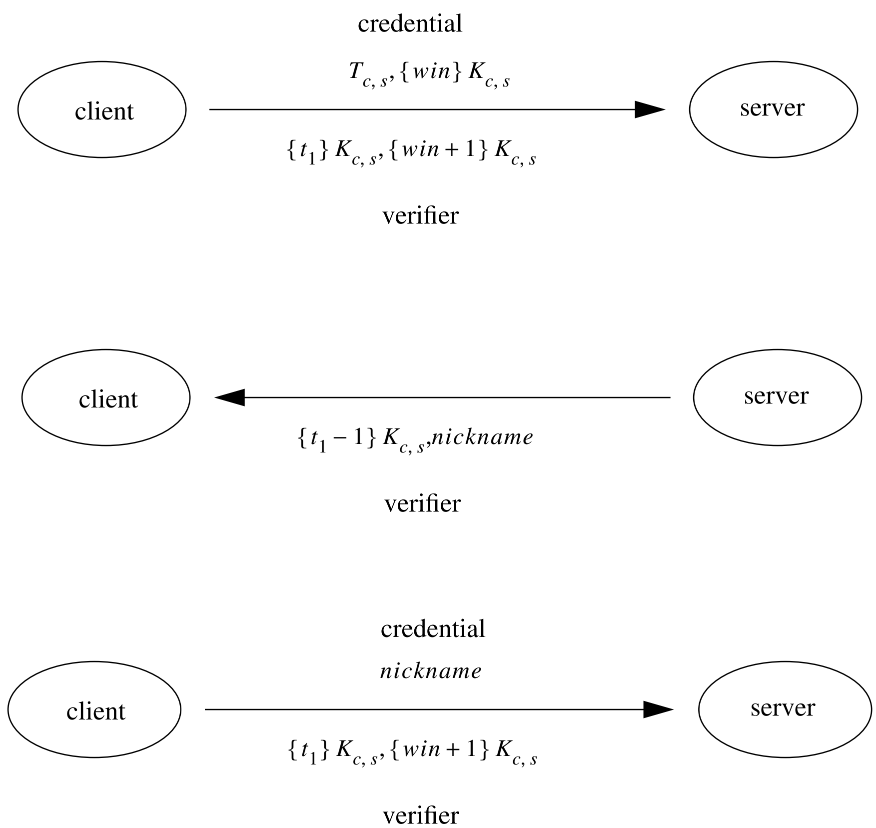
Note: we will have no reason to refer to an unencrypted ticket, so when we use the term *ticket* we mean the service ticket encrypted with the service's private key.

Using this notation, an AUTH_DES RPC exchange looks like:.



Kerberos-based authentication

While DES-based authentication must pass a "netname" and uses a public key cryptosystem to exchange encrypted session keys, Kerberos-based authentication takes advantage of the fact that the service ticket already contains both the principal's name and a copy of the session key. An AUTH_KERB RPC exchange looks like:



That is, the essential difference between AUTH_DES and AUTH_KERB is that DES-based authentication passes a "netname" and public-key encrypted session key while Kerberos-based authentication passes the (encrypted) service ticket.

AUTH_KERB details

Although conceptually very similar to AUTH_DES, AUTH_KERB differs in a number of small ways, all of which affect implementation interoperability.

Assumptions

Time synchronization

While AUTH_DES has the concept of a time window within which its credentials are valid, it does not place restrictions on the clocks of the client or server. The client is required to determine the time bias between itself and the server and compensate for the difference by adjusting the window time specified to the server.

Kerberos itself places loose time synchronization requirements on the hosts using its protocol (services must be able to detect stale tickets). Because of its use of Kerberos tickets, AUTH_KERB inherits these requirements: client and server clocks must be synchronized within 5 minutes of each other. Various methods of time synchronization are available *[DTS]*, *[RFC1059]*, *[RFC1119]*, *[RFC1305]*.

Well-known names

Kerberos users are identified by a primary name, instance, and realm. The RPC authentication code implementing AUTH_KERB largely ignores the realm (the supporting Kerberos library code does not, of course) and, under most circumstances the instance as well. It is assumed that user names are the same between client and server. This enables a server to translate a primary name into user identification information. Two forms of well-known names are used (we omit the realm):

`root.host`	represents a privileged user on client `host`
`user.ignored`	represents the user whose user name is `user`. The instance is ignored.

Encryption specifics

AUTH_KERB uses CBC mode when sending a full name credential (i.e. a credential that includes the ticket and window) and ECB mode otherwise. The session key is used as the initial input vector for CBC mode. Use of Kerberos version 4's nonstandard "propagating CBC" is avoided. It has been shown to have undesirable effects on message integrity *[Bellovin]*.

In the following we assume familiarity with XDR encoding *[RFC1014]* and use the notation

```
xdr_type(object)
```

to mean *object* XDRed as a `type`. The length referred to below is the size, in bytes of the credential or verifier, rounded (as XDR requires) up to 4-byte units. Using this notation, the full name credential and verifier are obtained as follows:

```
xdr_long(timestamp.seconds)
xdr_long(timestamp.useconds)
xdr_long(window)
xdr_long(window - 1)
```

is encrypted with CBC and input vector equal to the session key. The output is two DES cipher blocks, which we represent as

```
CB0
CB1.low
CB1.high
```

The credential is

```
xdr_long(AUTH_KERB)
xdr_long(length)
xdr_enum(AKN_FULLNAME)
xdr_bytes(ticket)
xdr_opaque(CB1.high)
```

and the verifier is

```
xdr_long(AUTH_KERB)
xdr_long(length)
xdr_opaque(CB0)
xdr_opaque(CB1.low)
```

For a “nickname” exchange, we do the following:

```
xdr_long(timestamp.seconds)
xdr_long(timestamp.useconds)
```

is encrypted with ECB to obtain

```
ECB0
```

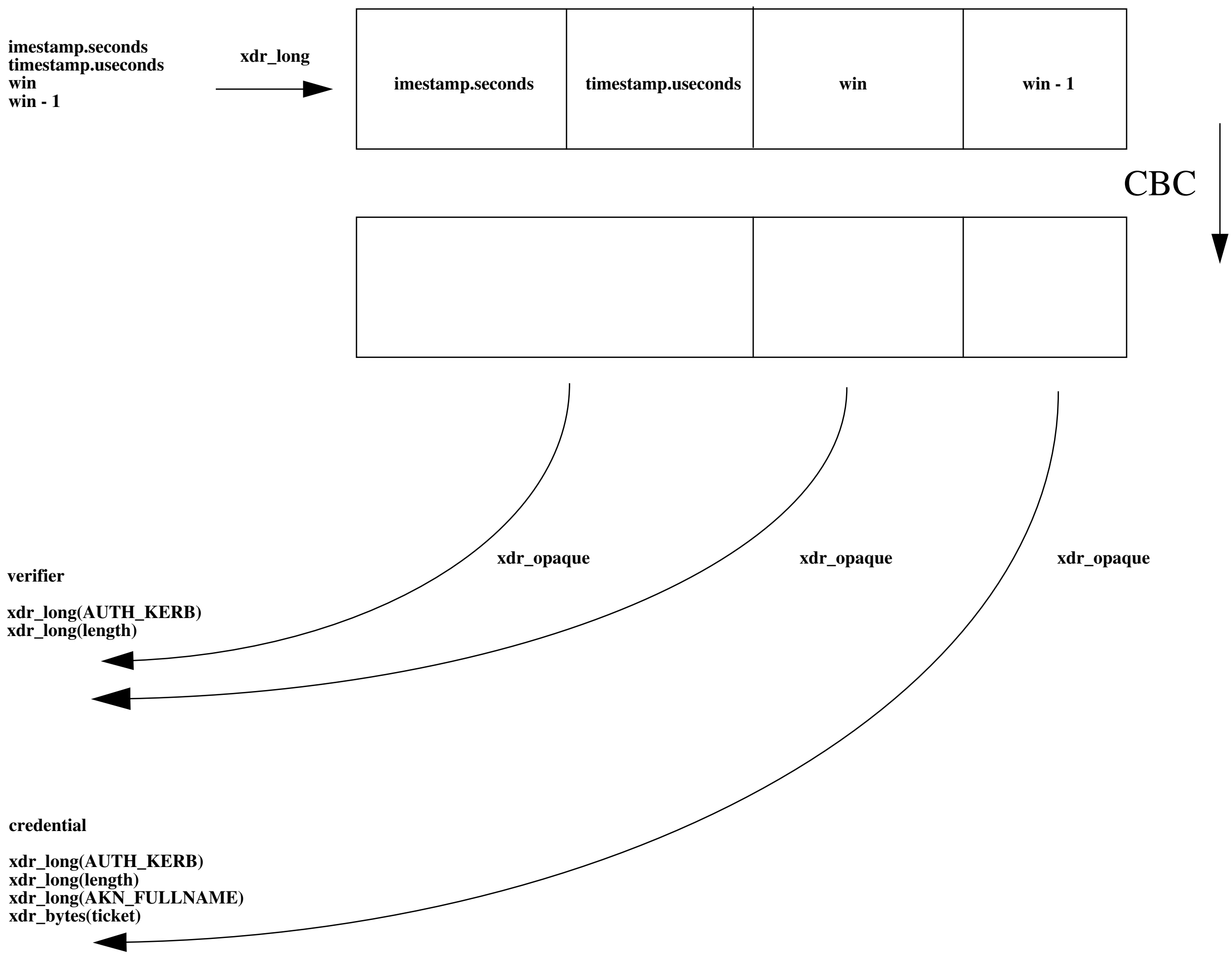
and the credential is

```
xdr_long(AUTH_KERB)
xdr_long(length)
xdr_enum(AKN_NICKNAME)
xdr_opaque(akc_nickname)
```

while the verifier is

```
xdr_long(AUTH_KERB)
xdr_long(length)
xdr_opaque(ECB0)
xdr_opaque(0)
```

This is undoubtedly confusing. A diagram will help clarify the XDR and encryption order:



NFS and AUTH_KERB

The following information is specific to the SunOS 5.x implementation of AUTH_KERB and necessarily contains UNIX-specific details.

kerbd

To avoid compiling Kerberos code into our kernels, NFS uses a proxy RPC daemon, `kerbd`, to make Kerberos library or protocol requests for it. This daemon exports three procedures:

KGETKCRED	is used by server-side RPC to check the authenticator presented by the client.
KSETKCRED	given a primary name, instance, and realm, returns the encrypted ticket and DES session key
KGETUCRED	is UNIX-specific: it returns the user's user ID, group ID, and groups list assuming that the primary name maps directly to a user name known to the server. Note that the reliance on the user's name to determine the groups to which the user belongs means that set group ID programs will not behave as expected with respect to authenticated NFS service.

Well-known names

The NFS service on server *s* is assumed to have the well-known principal name `nfs`.*s*. A privileged user on client *c* is assumed to have primary name `root` and instance *c*.

Flow of control

In order to put the use of `kerbd` and AUTH_KERB in perspective, this section follows an NFS mount request from start to finish. Note that since mount requests are done as root in SunOS, the user's identity throughout this example will be `root`.*c*.

1. client, *c*, makes MOUNTPROC_MOUNT request to server, *s*, to obtain the file handle for the directory to be mounted.

   Note: mount RPC calls are normally made with authentication flavor AUTH_NONE and "authenticated" via a "privileged port" mechanism.
2. client mount program makes an NFS mount system call, handing the client kernel (among other data) the file handle, mount flavor (assumed to be Kerberos in this case), time synchronization address, and the server's well-known name, "`nfs`.*s*".
3. client kernel contacts server at time synch address to derive client-server time bias
4. client kernel makes KSETKCRED RPC call to local `kerbd` to obtain ticket and session key
5. client kernel makes NFSPROC_GETATTR RPC call to server's NFS service, using full name credential and verifier described above.
6. server receives RPC call and makes KGETKCRED RPC call to its local `kerbd` to check the ticket presented by the client. The server's `kerbd` and the Kerberos library decrypt the ticket and return, among other data, the principal name and DES session key.
7. server checks that the ticket is still valid, uses the session key just obtained to decrypt the DES-encrypted portions of the credential and verifier, and checks the verifier for validity. Many Kerberos-specific RPC authentication errors may be returned at this point:

AUTH_BADCRED	if the verifier is invalid (the decrypted *win* in the credential and the decrypted *win* + 1 in the verifier do not match) or the timestamp does not lie within the window
AUTH_REJECTEDCRED	if a replay is detected

AUTH_BADVERF	if the verifier appears garbled (few sanity checks may be placed on the verifier - ensuring that the timestamp's microseconds field is valid is one).

Once the server is satisfied with the credential and verifier, it caches the client's identity and allocates a nickname (small integer) to be returned in the NFS reply.

8. server checks that the client is in the same realm as the server. If it is, the server makes a KGETUCRED call to its local `kerbd` to translate the principal's primary name into UNIX credentials. If it isn't, the user is marked as anonymous.

9. server checks these credentials against the file system's export information. There are three cases to consider: if the KGETUCRED call failed and anonymous requests are allowed, the UNIX credentials of the specified anonymous user are assigned; if the KGETUCRED call fails and anonymous requests are not allowed, the NFS call fails with RPC authentication error AUTH_TOOWEAK; otherwise (the KGETUCRED has succeeded and the user's UNIX credentials have been assigned), normal protection checking follows, including checking for root permission.

10. server constructs an NFS reply and sends it, including the nickname and server's verifier

11. client receives the reply, decrypts and validates the verifier, and remembers the nickname for future calls

The client now makes a second NFS call to the server. Much of this repeats steps 5 - 11 above.

12. client kernel makes NFSPROC_STATVFS RPC call to server's NFS service, using nickname credential and verifier described above.

13. server receives RPC call and validates the nickname. If it is out of range, the RPC authentication error AUTH_BADCRED is returned.

14. server uses the session key just obtained to decrypt the DES-encrypted portions of the verifier, and checks the verifier for validity. These Kerberos-specific RPC authentication errors may be returned at this point:

AUTH_REJECTEDVERF	If the timestamp is invalid, a replay is detected, or the timestamp does not lie within the window
AUTH_TIMEEXPIRE	If the service ticket has expired.

15. server uses the nickname to retrieve the caller's UNIX credentials

16. server checks these credentials against the file system's export information.

17. server constructs an NFS reply and sends it, including the nickname and server's verifier

18. client receives the reply, decrypts and validates the verifier, and remembers the nickname for future calls

19. client's NFS mount system call returns

Miscellaneous

Use of Kerberos authentication for NFS may lead to some surprises. Unlike AUTH_DES, there will come a time when the user's ticket-granting ticket has expired, requiring another invocation of `kinit`. When this occurs, NFS service for Kerberos mounts will fail until a new ticket-granting ticket is obtained.

Appendix I: Kerberos authentication in RPCL

```
#define AUTH_KERB 4

/*
 * There are two kinds of credentials: one in which the client sends
 * the (previously encrypted) Kerberos ticket, and one in which it
 * uses its "nickname" (just an unsigned integer) given to it by
 * the server.  The client must use its full name in its first
 * transaction with the server; the server will return to the client
 * a nickname.  The client may use its nickname in all further
 * transactions with the server (until the ticket expires).  There
 * is no requirement to use the nickname, but it is wise to use it
 * for performance reasons.
 */
enum authkerb_namekind {
     AKN_FULLNAME = 0,
     AKN_NICKNAME = 1
 };

/*
 * A fullname contains the network name of the client, an encrypted
 * conversation key and the window.  The window is actually a
 * lifetime for the credential.  If the time indicated in the
 * verifier timestamp plus the window has passed, then the server
 * should expire the request and not grant it.  To insure that
 * requests are not replayed, the server should insist that
 * timestamps be greater than the previous one seen, unless it is
 * the first transaction.
 * In the first transaction, the server checks instead that the
 * window verifier is one less than the window.
 */
struct authkerb_fullname {
     KTEXT_ST ticket;             /* encrypted service ticket */
     unsigned long window;        /* encrypted window */
};

/*
 * A credential is either a fullname or a nickname
 */
union authkerb_cred switch(authkerb_namekind akc_namekind){
     case AKN_FULLNAME:
         authkerb_fullname akc_fullname;
     case AKN_NICKNAME:
         unsigned long akc_nickname;
};

/*
 * A timestamp encodes the time since midnight, January 1, 1970.
 */
struct timestamp {
     unsigned long seconds;      /* seconds */
     unsigned long useconds;     /* and microseconds */
};
```

```
/*
 * Verifier: client variety
 */
struct authkerb_verf_clnt {
      timestamp akv_timestamp;    /* encrypted timestamp */
      unsigned long akv_winverf;  /* encrypted window verifier */
};

/*
 * Verifier: server variety
 * The server returns (encrypted) the same timestamp the client
 * gave it minus one second.  It also tells the client its nickname
 * to be used in future transactions (unencrypted).
 */
struct authkerb_verf_svr {
      timestamp akv_timeverf;     /* encrypted verifier */
      unsigned long akv_nickname; /* new nickname for clnt */
};
```

Appendix II: Bibliography

[Bellovin] Bellovin, Steven M., Merritt, Michael,
"Limitations of the Kerberos Authentication System,"
USENIX Conference Proceedings, USENIX Associatiion, Berkeley, CA, January 1991.

[DES] Data Encryption Standard,
Federal Information Processing Standard Publication 46-1,
National Bureau of Standards, U.S. Department of Commerce,, January 22, 1988.

[Diffie] Diffie, W. and Hellman, M.E.,
"New Directions in Cryptography,"
IEEE Transactions on Information Theory, Volume IT-22, Number 6, November, 1976.

[DTS] Digital Equipment Corporation,
Digital Time Service Functional Specification Version T.1.0.5,
1989.

[ONC] ONC/NFS Technology Guide
Sun Microsystems, Inc., Fall 1992.

[RFC1014] Sun Microsystems, Inc.,
RFC 1014,
"XDR: External Data Representation Standard,"
June 1987.

[RFC1057] Sun Microsystems, Inc.,
RFC 1057,
"RPC: Remote Procedure Call Protocol specification: Version 2,"
June, 1988.

[RFC1059] Mills, D.L.,
RFC 1059,
"Network Time Protocol (version 1) specification and implementation,"
July, 1988.

[RFC1119] Mills, D.L.,
RFC 1119,
"Network Time Protocol (version 2) specification and implementation,"
September, 1989.

[RFC1305] Mills, D.L.,
RFC 1305,
"Network Time Protocol (Version 3): Specification, implementation, and analysis,"
March, 1992.

[Steiner] Steiner, Jennifer G, Neuman, Clifford, and Schiller, Jeffrey J.
"Kerberos: An Authentication Service for Open Network Systems,."
USENIX Conference Proceedings, USENIX Association, Berkeley, CA, June 1988.