                 Authentication Mechanisms for ONC RPC

Status of this Memo

   This memo provides information for the Internet community. It does
   not specify an Internet standard of any kind. Distribution of this
   memo is unlimited.

Copyright Notice

ABSTRACT

   This document describes two authentication mechanisms created by Sun
   Microsystems that are commonly used in conjunction with the ONC
   Remote Procedure Call (ONC RPC Version 2) protocol.

WARNING

   The DH authentication as defined in Section 2 in this document refers
   to the authentication mechanism with flavor AUTH_DH currently
   implemented in ONC RPC. It uses the underlying Diffie-Hellman
   algorithm for key exchange. The DH authentication defined in this
   document is flawed due to the selection of a small prime for the BASE
   field (Section 2.5). To avoid the flaw a new DH authentication
   mechanism could be defined with a larger prime. However, the new DH
   authentication would not be interoperable with the existing DH
   authentication.

   As illustrated in [10], a large number of attacks are possible on ONC
   RPC system services that use non-secure authentication mechanisms.
   Other secure authentication mechanisms need to be developed for ONC
   RPC. RFC 2203 describes the RPCSEC_GSS ONC RPC security flavor, a
   secure authentication mechanism that enables RPC protocols to use
   Generic Security Service Application Program Interface (RFC 2078) to
   provide security services, integrity and privacy, that are
   independent of the underlying security mechanisms.

Table of Contents

1. Introduction

   The ONC RPC protocol provides the fields necessary for a client to
   identify itself to a service, and vice-versa, in each call and reply
   message. Security and access control mechanisms can be built on top
   of this message authentication. Several different authentication
   protocols can be supported.

   This document specifies two authentication protocols created by Sun
   Microsystems that are commonly used: Diffie-Hellman (DH)
   authentication and Kerberos (Version 4) based authentication.

   As a prerequisite to reading this document, the reader is expected to
   be familiar with [1] and [2]. This document uses terminology and
   definitions from [1] and [2].

2. Diffie-Hellman Authentication

   System authentication (defined in [1]) suffers from some problems.
   It is very UNIX oriented, and can be easily faked (there is no
   attempt to provide cryptographically secure authentication).

DH authentication was created to address these problems. However, it
has been compromised [9] due to the selection of a small length for
the prime in the ONC RPC implementation. While the information
provided here will be useful for implementors to ensure
interoperability with existing applications that use DH
authentication, it is strongly recommended that new applications use
more secure authentication, and that existing applications that
currently use DH authentication migrate to more robust authentication
mechanisms.

2.1 Naming

The client is addressed by a simple string of characters instead of
by an operating system specific integer. This string of characters
is known as the "netname" or network name of the client. The server
is not allowed to interpret the contents of the client's name in any
other way except to identify the client. Thus, netnames should be
unique for every client in the Internet.

It is up to each operating system's implementation of DH
authentication to generate netnames for its users that insure this
uniqueness when they call upon remote servers. Operating systems
already know how to distinguish users local to their systems. It is
usually a simple matter to extend this mechanism to the network. For
example, a UNIX(tm) user at Sun with a user ID of 515 might be
assigned the following netname: "unix.515@sun.com". This netname
contains three items that serve to insure it is unique. Going
backwards, there is only one naming domain called "sun.com" in the
Internet. Within this domain, there is only one UNIX(tm) user with
user ID 515. However, there may be another user on another operating
system, for example VMS, within the same naming domain that, by
coincidence, happens to have the same user ID. To insure that these
two users can be distinguished we add the operating system name. So
one user is "unix.515@sun.com" and the other is "vms.515@sun.com".
The first field is actually a naming method rather than an operating
system name. It happens that today there is almost a one-to-one
correspondence between naming methods and operating systems. If the
world could agree on a naming standard, the first field could be the
name of that standard, instead of an operating system name.

2.2 DH Authentication Verifiers

Unlike System authentication, DH authentication does have a verifier
so the server can validate the client's credential (and vice-versa).
The contents of this verifier are primarily an encrypted timestamp.
The server can decrypt this timestamp, and if it is within an
accepted range relative to the current time, then the client must
have encrypted it correctly. The only way the client could encrypt

it correctly is to know the "conversation key" of the RPC session,
and if the client knows the conversation key, then it must be the
real client.

The conversation key is a DES [5] key which the client generates and
passes to the server in the first RPC call of a session. The
conversation key is encrypted using a public key scheme in this first
transaction. The particular public key scheme used in DH
authentication is Diffie-Hellman [3] with 192-bit keys. The details
of this encryption method are described later.

The client and the server need the same notion of the current time in
order for all of this to work, perhaps by using the Network Time
Protocol [4]. If network time synchronization cannot be guaranteed,
then the client can determine the server's time before beginning the
conversation using a time request protocol.

The way a server determines if a client timestamp is valid is
somewhat complicated. For any other transaction but the first, the
server just checks for two things:

(1) the timestamp is greater than the one previously seen from the
same client. (2) the timestamp has not expired.

A timestamp is expired if the server's time is later than the sum of
the client's timestamp plus what is known as the client's "ttl"
(standing for "time-to-live" - you can think of this as the lifetime
for the client's credential). The "ttl" is a number the client
passes (encrypted) to the server in its first transaction.

In the first transaction, the server checks only that the timestamp
has not expired. Also, as an added check, the client sends an
encrypted item in the first transaction known as the "ttl verifier"
which must be equal to the time-to-live minus 1, or the server will
reject the credential. If either check fails, the server rejects the
credential with an authentication status of AUTH_BADCRED, however if
the timestamp is earlier than the previous one seen, the server
returns an authentication status of AUTH_REJECTEDCRED.

The client too must check the verifier returned from the server to be
sure it is legitimate. The server sends back to the client the
timestamp it received from the client, minus one second, encrypted
with the conversation key. If the client gets anything different
than this, it will reject it, returning an AUTH_INVALIDRESP
authentication status to the user.

2.3 Nicknames and Clock Synchronization

   After the first transaction, the server's DH authentication subsystem
   returns in its verifier to the client an integer "nickname" which the
   client may use in its further transactions instead of passing its
   netname. The nickname could be an index into a table on the server
   which stores for each client its netname, decrypted conversation key
   and ttl.

   Though they originally were synchronized, the client's and server's
   clocks can get out of synchronization again. When this happens the
   server returns to the client an authentication status of
   AUTH_REJECTEDVERF at which point the client should attempt to
   resynchronize.

   A client may also get an AUTH_BADCRED error when using a nickname
   that was previously valid. The reason is that the server's nickname
   table is a limited size, and it may flush entries whenever it wants.
   A client should resend its original full name credential in this case
   and the server will give it a new nickname. If a server crashes, the
   entire nickname table gets flushed, and all clients will have to
   resend their original credentials.

2.4 DH Authentication Protocol Specification

   There are two kinds of credentials: one in which the client uses its
   full network name, and one in which it uses its "nickname" (just an
   unsigned integer) given to it by the server. The client must use its
   fullname in its first transaction with the server, in which the
   server will return to the client its nickname. The client may use
   its nickname in all further transactions with the server. There is no
   requirement to use the nickname, but it is wise to use it for
   performance reasons.

   The following definitions are used for describing the protocol:

```
   enum authdh_namekind {
      ADN_FULLNAME = 0,
      ADN_NICKNAME = 1
   };

   typedef opaque des_block[8]; /* 64-bit block of encrypted data */

   const MAXNETNAMELEN = 255;   /* maximum length of a netname */
```

   The flavor used for all DH authentication credentials and verifiers
   is "AUTH_DH", with the numerical value 3. The opaque data
   constituting the client credential encodes the following structure:

```
union authdh_cred switch (authdh_namekind namekind) {
case ADN_FULLNAME:
   authdh_fullname fullname;
case ADN_NICKNAME:
   authdh_nickname nickname;
};
```

The opaque data constituting a verifier that accompanies a client
credential encodes the following structure:

```
union authdh_verf switch (authdh_namekind namekind) {
case ADN_FULLNAME:
   authdh_fullname_verf fullname_verf;
case ADN_NICKNAME:
   authdh_nickname_verf nickname_verf;
};
```

The opaque data constituting a verifier returned by a server in
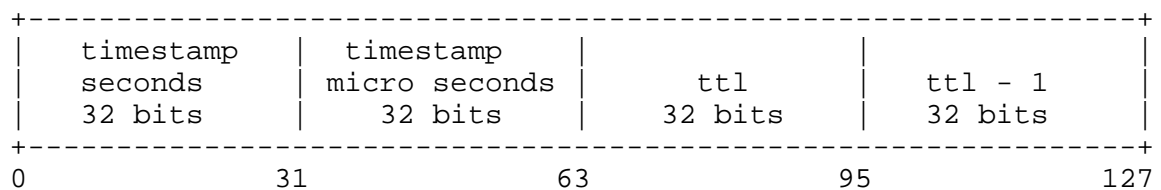response to a client request encodes the following structure:
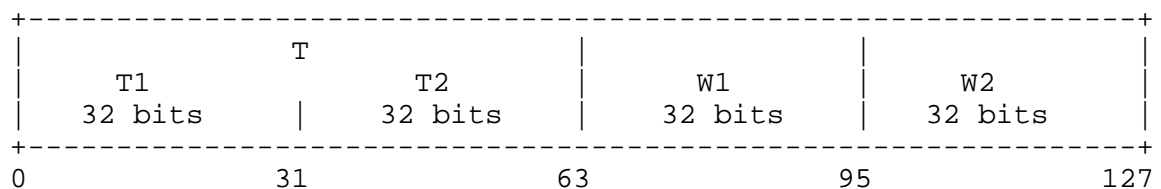
```
struct authdh_server_verf;
```

These structures are described in detail below.

2.4.1 The Full Network Name Credential and Verifier (Client)

First, the client creates a conversation key for the session. Next,
the client fills out the following structure:

```
    +----------------------------------------------------------------+
    |  timestamp   |  timestamp     |            |            |
    |  seconds     | micro seconds  |    ttl     |  ttl - 1   |
    |  32 bits     |    32 bits     |   32 bits  |   32 bits  |
    +----------------------------------------------------------------+
    0              31               63           95           127
```

The fields are stored in XDR (external data representation) format.
The timestamp encodes the time since midnight, January 1, 1970. These
128 bits of data are then encrypted in the DES CBC mode, using the
conversation key for the session, and with an initialization vector
of 0. This yields:

```
    +----------------------------------------------------------------+
    |              T             |            |            |
    |      T1          |    T2    |    W1      |    W2      |
    |   32 bits    |   32 bits    |   32 bits  |   32 bits  |
    +----------------------------------------------------------------+
    0              31               63           95           127
```

where T1, T2, W1, and W2 are all 32-bit quantities, and have some
correspondence to the original quantities occupying their positions,
but are now interdependent on each other for proper decryption. The
64 bit sequence comprising T1 and T2 is denoted by T.

The full network name credential is represented as follows using XDR
notation:

```
struct authdh_fullname {
   string name<MAXNETNAMELEN>;  /* netname of client           */
   des_block key;               /* encrypted conversation key  */
   opaque w1[4];                /* W1                          */
};
```

The conversation key is encrypted using the "common key" using the
ECB mode. The common key is a DES key that is derived from the
Diffie-Hellman public and private keys, and is described later.

The verifier is represented as follows:

```
struct authdh_fullname_verf {
   des_block timestamp;         /* T (the 64 bits of T1 and T2) */
   opaque w2[4];                /* W2                           */
};
```
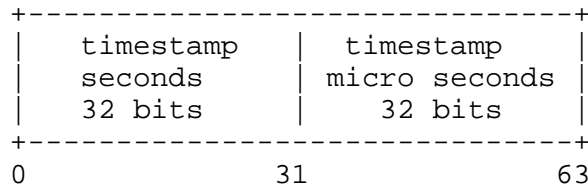
Note that all of the encrypted quantities (key, w1, w2, timestamp) in
the above structures are opaque.

The fullname credential and its associated verifier together contain
the network name of the client, an encrypted conversation key, the
ttl, a timestamp, and a ttl verifier that is one less than the ttl.
The ttl is actually the lifetime for the credential. The server will
accept the credential if the current server time is "within" the time
indicated in the timestamp plus the ttl. Otherwise, the server
rejects the credential with an authentication status of AUTH_BADCRED.
One way to insure that requests are not replayed would be for the
server to insist that timestamps are greater than the previous one
seen, unless it is the first transaction. If the timestamp is
earlier than the previous one seen, the server returns an
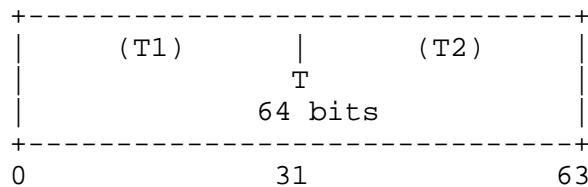authentication status of AUTH_REJECTEDCRED.

The server returns a authdh_server_verf structure, which is described
in detail below. This structure contains a "nickname", which may be
used for subsequent requests in the current conversation.

2.4.2 The Nickname Credential and Verifier (Client)

   In transactions following the first, the client may use the shorter
   nickname credential and verifier for efficiency. First, the client
   fills out the following structure:

```
      +-------------------------------+
      |   timestamp   |   timestamp   |
      |   seconds     | micro seconds |
      |   32 bits     |    32 bits    |
      +-------------------------------+
      0               31              63
```

   The fields are stored in XDR (external data representation) format.
   These 64 bits of data are then encrypted in the DES ECB mode, using
   the conversation key for the session. This yields:

```
      +-------------------------------+
      |     (T1)      |     (T2)      |
      |             T                 |
      |           64 bits             |
      +-------------------------------+
      0               31              63
```

   The nickname credential is represented as follows using XDR notation:

   struct authdh_nickname {
      unsigned int nickname;   /* nickname returned by server   */
   };

   The nickname verifier is represented as follows using XDR notation:

   struct authdh_nickname_verf {
      des_block timestamp;   /* T (the 64 bits of T1 and T2) */
      opaque w[4];   /* Set to zero   */
   };

   The nickname credential may be reject by the server for several
   reasons. An authentication status of AUTH_BADCRED indicates that the
   nickname is no longer valid. The client should retry the request
   using the fullname credential. AUTH_REJECTEDVERF indicates that the
   nickname verifier is not valid. Again, the client should retry the
   request using the fullname credential.

2.4.3 The Nickname Verifier (Server)

   The server never returns a credential. It returns only one kind of
   verifier, i.e., the nickname verifier. This has the following XDR
   representation:

   struct authdh_server_verf {
      des_block timestamp_verf; /* timestamp verifier (encrypted)   */
      unsigned int nickname;   /* new client nickname (unencrypted) */
   };

   The timestamp verifier is constructed in exactly the same way as the
   client nickname credential. The server sets the timestamp value to
   the value the client sent minus one second and encrypts it in DES ECB
   mode using the conversation key. The server also sends the client a
   nickname to be used in future transactions (unencrypted).

2.5 Diffie-Hellman Encryption

   In this scheme, there are two constants "BASE" and "MODULUS" [3].
   The particular values Sun has chosen for these for the DH
   authentication protocol are:

      const BASE = 3;
      const MODULUS = "d4a0ba0250b6fd2ec626e7efd637df76c716e22d0944b88b";

   Note that the modulus is represented above as a hexadecimal string.

   The way this scheme works is best explained by an example. Suppose
   there are two people "A" and "B" who want to send encrypted messages
   to each other. So, A and B both generate "secret" keys at random
   which they do not reveal to anyone. Let these keys be represented as
   SK(A) and SK(B). They also publish in a public directory their
   "public" keys. These keys are computed as follows:

      PK(A) = (BASE ** SK(A)) mod MODULUS
      PK(B) = (BASE ** SK(B)) mod MODULUS

   The "**" notation is used here to represent exponentiation. Now, both
   A and B can arrive at the "common" key between them, represented here
   as CK(A, B), without revealing their secret keys.

A computes:

    CK(A, B) = (PK(B) ** SK(A)) mod MODULUS

while B computes:

    CK(A, B) = (PK(A) ** SK(B)) mod MODULUS

These two can be shown to be equivalent:

    (PK(B) ** SK(A)) mod MODULUS = (PK(A) ** SK(B)) mod MODULUS

We drop the "mod MODULUS" parts and assume modulo arithmetic to simplify
things:

    PK(B) ** SK(A) = PK(A) ** SK(B)

Then, replace PK(B) by what B computed earlier and likewise for PK(A).

    (BASE ** SK(B)) ** SK(A) = (BASE ** SK(A)) ** SK(B)

which leads to:

    BASE ** (SK(A) * SK(B)) = BASE ** (SK(A) * SK(B))

This common key CK(A, B) is not used to encrypt the timestamps used
in the protocol. Rather, it is used only to encrypt a conversation
key which is then used to encrypt the timestamps. The reason for
doing this is to use the common key as little as possible, for fear
that it could be broken. Breaking the conversation key is a far less
damaging, since conversations are relatively short-lived.

The conversation key is encrypted using 56-bit DES keys, yet the
common key is 192 bits. To reduce the number of bits, 56 bits are
selected from the common key as follows. The middle-most 8-bytes are
selected from the common key, and then parity is added to the lower
order bit of each byte, producing a 56-bit key with 8 bits of parity.

Only 48 bits of the 8-byte conversation key are used in the DH
Authentication scheme. The least and most significant bits of each
byte of the conversation key are unused.

3. Kerberos-based Authentication

   Conceptually, Kerberos-based authentication is very similar to DH
   authentication. The major difference is, Kerberos-based
   authentication takes advantage of the fact that Kerberos tickets have

encoded in them the client name and the conversation key. This RFC
does not describe Kerberos name syntax, protocols and ticket formats.
The reader is referred to [6], [7], and [8].

3.1 Naming

A Kerberos name contains three parts. The first is the principal
name, which is usually a user's or service's name. The second is the
instance, which in the case of a user is usually NULL. Some users
may have privileged instances, however, such as root or admin. In
the case of a service, the instance is the name of the machine on
which it runs; that is, there can be an NFS service running on the
machine ABC, which is different from the NFS service running on the
machine XYZ. The third part of a Kerberos name is the realm. The
realm corresponds to the Kerberos service providing authentication
for the principal. When writing a Kerberos name, the principal name
is separated from the instance (if not NULL) by a period, and the
realm (if not the local realm) follows, preceded by an "@" sign. The
following are examples of valid Kerberos names:

        billb
        jis.admin
        srz@lcs.mit.edu
        treese.root@athena.mit.edu

3.2 Kerberos-based Authentication Protocol Specification

The Kerberos-based authentication protocol described is based on
Kerberos version 4.

There are two kinds of credentials: one in which the client uses its
full network name, and one in which it uses its "nickname" (just an
unsigned integer) given to it by the server. The client must use its
fullname in its first transaction with the server, in which the
server will return to the client its nickname. The client may use
its nickname in all further transactions with the server. There is no
requirement to use the nickname, but it is wise to use it for
performance reasons.

The following definitions are used for describing the protocol:

```
enum authkerb4_namekind {
    AKN_FULLNAME = 0,
    AKN_NICKNAME = 1
};
```

The flavor used for all Kerberos-based authentication credentials and
verifiers is "AUTH_KERB4", with numerical value 4. The opaque data
constituting the client credential encodes the following structure:

```
union authkerb4_cred switch (authkerb4_namekind namekind) {
case AKN_FULLNAME:
   authkerb4_fullname fullname;
case AKN_NICKNAME:
   authkerb4_nickname nickname;
};
```

The opaque data constituting a verifier that accompanies a client
credential encodes the following structure:

```
union authkerb4_verf switch (authkerb4_namekind namekind) {
case AKN_FULLNAME:
   authkerb4_fullname_verf fullname_verf;
case AKN_NICKNAME:
   authkerb4_nickname_verf nickname_verf;
};
```

The opaque data constituting a verifier returned by a server in
response to a client request encodes the following structure:

```
struct authkerb4_server_verf;
```

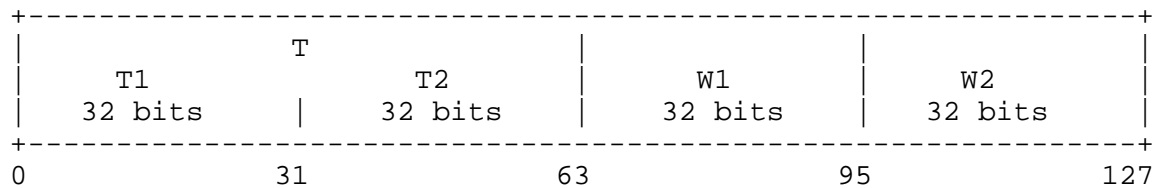These structures are described in detail below.

3.2.1 The Full Network Name Credential and Verifier (Client)

First, the client must obtain a Kerberos ticket from the Kerberos
Server. The ticket contains a Kerberos session key, which will
become the conversation key. Next, the client fills out the
following structure:

```
    +----------------------------------------------------------------+
    |   timestamp   |   timestamp   |            |                   |
    |    seconds    | micro seconds |    ttl     |    ttl - 1        |
    |    32 bits    |    32 bits     |  32 bits   |    32 bits        |
    +----------------------------------------------------------------+
    0              31              63           95                 127
```

The fields are stored in XDR (external data representation) format.
The timestamp encodes the time since midnight, January 1, 1970.
"ttl" is identical in meaning to the corresponding field in Diffie-
Hellman authentication: the credential "time-to-live" for the

conversation being initiated. These 128 bits of data are then
encrypted in the DES CBC mode, using the conversation key, and with
an initialization vector of 0. This yields:

```
+----------------------------------------------------------------+
|               T                |           |           |       |
|      T1        |      T2        |    W1     |    W2     |       |
|    32 bits     |    32 bits     |  32 bits  |  32 bits  |       |
+----------------------------------------------------------------+
0               31               63          95              127
```

where T1, T2, W1, and W2 are all 32-bit quantities, and have some
correspondence to the original quantities occupying their positions,
but are now interdependent on each other for proper decryption. The
64 bit sequence comprising T1 and T2 is denoted by T.

The full network name credential is represented as follows using XDR
notation:

```
struct authkerb4_fullname {
    opaque ticket<>;          /* kerberos ticket for the server */
    opaque w1[4];             /* W1                             */
};
```

The verifier is represented as follows:

```
struct authkerb4_fullname_verf {
    des_block timestamp;         /* T (the 64 bits of T1 and T2) */
    opaque w2[4];                /* W2                           */
};
```
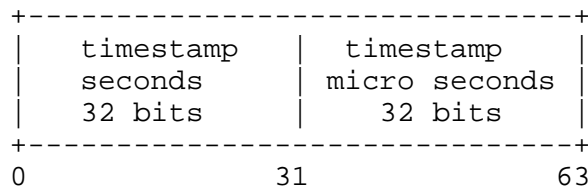
Note that all of the client-encrypted quantities (w1, w2, timestamp)
in the above structures are opaque. The client does not encrypt the
Kerberos ticket for the server.

The fullname credential and its associated verifier together contain
the Kerberos ticket (which contains the client name and the
conversation key), the ttl, a timestamp, and a ttl verifier that is
one less than the ttl. The ttl is actually the lifetime for the
credential. The server will accept the credential if the current
server time is "within" the time indicated in the timestamp plus the
ttl. Otherwise, the server rejects the credential with an
authentication status of AUTH_BADCRED. One way to insure that
requests are not replayed would be for the server to insist that
timestamps are greater than the previous one seen, unless it is the
first transaction. If the timestamp is earlier than the previous one
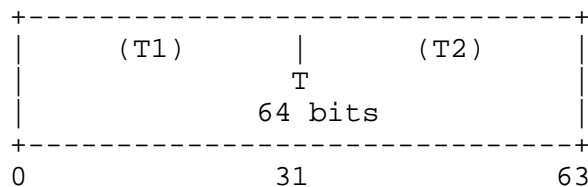seen, the server returns an authentication status of
AUTH_REJECTEDCRED.

The server returns a authkerb4_server_verf structure, which is
described in detail below. This structure contains a "nickname",
which may be used for subsequent requests in the current session.

3.2.2 The Nickname Credential and Verifier (Client)

In transactions following the first, the client may use the shorter
nickname credential and verifier for efficiency. First, the client
fills out the following structure:

```
    +------------------------------+
    |   timestamp    |  timestamp  |
    |   seconds      | micro seconds |
    |   32 bits      |    32 bits   |
    +------------------------------+
    0               31            63
```

The fields are stored in XDR (external data representation) format.
These 64 bits of data are then encrypted in the DES ECB mode, using
the conversation key for the session. This yields:

```
    +------------------------------+
    |     (T1)       |     (T2)    |
    |              T               |
    |           64 bits            |
    +------------------------------+
    0               31            63
```

The nickname credential is represented as follows using XDR notation:

```
struct authkerb4_nickname {
   unsigned int nickname;       /* nickname returned by server   */
};
```

The nickname verifier is represented as follows using XDR notation:

```
struct authkerb4_nickname_verf {
   des_block timestamp;         /* T (the 64 bits of T1 and T2) */
   opaque w[4];                 /* Set to zero                  */
};
```

The nickname credential may be reject by the server for several
reasons. An authentication status of AUTH_BADCRED indicates that the
nickname is no longer valid. The client should retry the request
using the fullname credential. AUTH_REJECTEDVERF indicates that the
nickname verifier is not valid. Again, the client should retry the

request using the fullname credential. AUTH_TIMEEXPIRE indicates
that the session's Kerberos ticket has expired. The client should
initiate a new session by obtaining a new Kerberos ticket.

3.2.3 The Nickname Verifier (Server)

The server never returns a credential. It returns only one kind of
verifier, i.e., the nickname verifier. This has the following XDR
representation:

```
struct authkerb4_server_verf {
    des_block timestamp_verf; /* timestamp verifier (encrypted)    */
    unsigned int nickname;    /* new client nickname (unencrypted) */
};
```

The timestamp verifier is constructed in exactly the same way as the
client nickname credential. The server sets the timestamp value to
the value the client sent minus one second and encrypts it in DES ECB
mode using the conversation key. The server also sends the client a
nickname to be used in future transactions (unencrypted).

3.2.4 Kerberos-specific Authentication Status Values

The server may return to the client one of the following errors in
the authentication status field:

```
enum auth_stat {
    ...
    /*
     * kerberos errors
     */
    AUTH_KERB_GENERIC = 8,  /* Any Kerberos-specific error other
                               than the following                 */
    AUTH_TIMEEXPIRE = 9,    /* The client's ticket has expired     */
    AUTH_TKT_FILE = 10,     /* The server was unable to find the
                               ticket file.  The client should
                               create a new session by obtaining a
                               new ticket                          */
    AUTH_DECODE = 11,       /* The server is unable to decode the
                               authenticator of the client's ticket */
    AUTH_NET_ADDR = 12      /* The network address of the client
                               does not match the address contained
                               in the ticket                       */

    /* and more to be defined */
};
```

4. Security Considerations

   The DH authentication mechanism and the Kerberos V4 authentication
   mechanism are described in this document only for informational
   purposes.

   In addition to the weakness pointed out earlier in this document (see
   WARNING on page 1), the two security mechanisms described herein lack
   the support for integrity and privacy data protection. It is strongly
   recommended that new applications use more secure mechanisms, and
   that existing applications migrate to more robust mechanisms.

   The RPCSEC_GSS ONC RPC security flavor, specified in RFC 2203, allows
   applications built on top of RPC to access security mechanisms that
   adhere to the GSS-API specification. It provides a GSS-API based
   security framework that allows for strong security mechanisms. RFC
   1964 describes the Kerberos Version 5 GSS-API security mechanism
   which provides integrity and privacy, in addition to authentication.
   RFC 2623 [14] describes how Kerberos V5 is pluggued into RPCSEC_GSS,
   and how the Version 2 and Version 3 of the NFS protocol use Kerberos
   V5 via RPCSEC_GSS. The RPCSEC_GSS/GSS-API/Kerberos-V5 stack provides
   a robust security mechanism for applications that require strong
   protection.

5. REFERENCES

   [1]   Srinivasan, R., "Remote Procedure Call Protocol Version 2", RFC
         1831, August 1995.

   [2]   Srinivasan, R., "XDR: External Data Representation Standard",
         RFC 1832, August 1995.

   [3]   Diffie & Hellman, "New Directions in Cryptography", IEEE
         Transactions on Information Theory IT-22, November 1976.

   [4]   Mills, D., "Network Time Protocol (Version 3)", RFC 1305, March
         1992.

   [5]   National Bureau of Standards, "Data Encryption Standard",
         Federal Information Processing Standards Publication 46, January
         1977.

   [6]   Miller, S., Neuman, C., Schiller, J. and J. Saltzer, "Section
         E.2.1: Kerberos Authentication and Authorization System",
         December 1987.

[7]   Steiner, J., Neuman, C. and J. Schiller, "Kerberos: An
      Authentication Service for Open Network Systems", pp. 191-202 in
      Usenix Conference Proceedings, Dallas, Texas, February, 1988.

[8]   Kohl, J. and C. Neuman, "The Kerberos Network Authentication
      Service (V5)", RFC 1510, September 1993.

[9]   La Macchia, B.A., and Odlyzko, A.M., "Computation of Discrete
      Logarithms in Prime Fields", pp. 47-62 in "Designs, Codes and
      Cryptography", Kluwer Academic Publishers, 1991.

[10] Cheswick, W.R., and Bellovin, S.M., "Firewalls and Internet
      Security," Addison-Wesley, 1995.

[11] Linn, J., "The Kerberos Version 5 GSS-API Mechanism", RFC 1964,
      June 1996.

[12] Linn, J., "Generic Security Service Application Program
      Interface, Version 2", RFC 2078, January 1997.

[13] Eisler, M., Chiu, A., and Ling, L., "RPCSEC_GSS Protocol
      Specification", RFC 2203, September 1997.

[14] Eisler, M., "NFS Version 2 and Version 3 Security Issues and the
      NFS Protocol's Use of RPCSEC_GSS and Kerberos V5", RFC 2623,
      June 1999.

6. AUTHOR'S ADDRESS

Alex Chiu
Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303

Phone: +1 (650) 786-6465
EMail: alex.chiu@Eng.sun.com

7. Full Copyright Statement

Acknowledgement