Network Working Group   Wilkinson
Internet-Draft   YFS
Intended status: Informational   March 18, 2013
Expires: September 19, 2013

rxgk: GSSAPI based security class for RX
draft-wilkinson-afs3-rxgk-03

Abstract

rxgk is a security class for the RX RPC protocol. It uses the GSSAPI framework to provide authentication, confidentiality and integrity protection. This document provides a general description of rxgk. A further document will provide details of integration with specific RX applications.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 19, 2013.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

Table of Contents

1. Introduction

rxgk is a GSSAPI [RFC2743] based security class for the rx [RX] protocol. It provides authentication, confidentiality and integrity protection for rx RPC calls, using a security context established using any GSSAPI mechanism with PRF [RFC4401] support. The External Data Representation Standardard, XDR [RFC4506], is used to represent data structures on the wire and in the code fragments contained within this document.

Architecturally, rxgk is split into two parts. The rxgk rx security class provides strong encryption using previously negotiated ciphers and keys. It builds on the Kerberos crypto framework for its encryption requirements, but is authentication mechanism independent -- the class itself does not require the use of either Kerberos, or GSSAPI. The security class simply uses a previously negotiated encryption type, and master key. The master key is never directly used, but instead a per-connection key is derived for each new secure connection that is established.

The second portion of rxgk is a service which permits the negotiation of an encryption algorithm, and the establishment of a master key. This is done via a separate RPC exchange with a server, prior to the setup of any rxgk connections. The exchange establishes an rxgk token, and a master key shared between client and server. This exchange is protected within a GSSAPI security context.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Time Representation

rxgk expresses absolute time as a 64-bit integer. This contains the time relative to midnight, or 0 hour, January 1, 1970 UTC, represented in increments of 100 nanoseconds, excluding any leap seconds. Negative times, whilst permitted by the representation, MUST NOT be used within rxgk.

```
typedef hyper rxgkTime;
```


3. Encryption Framework

Bulk data encryption within rxgk is performed using the encryption framework defined by RFC3961 [RFC3961]. Any algorithm which is defined using this framework and supported by both client and server may be used.

3.1. Key Usage Values

In order to avoid using the same key for multiple tasks, key derivation is employed. To avoid any conflicts with other users of these keys, key usage numbers are allocated within the application space documented in section 4 of RFC4120 [RFC4120].

```
const RXGK_CLIENT_ENC_PACKET            = 1026;
const RXGK_CLIENT_MIC_PACKET            = 1027;
const RXGK_SERVER_ENC_PACKET            = 1028;
const RXGK_SERVER_MIC_PACKET            = 1029;
const RXGK_CLIENT_ENC_RESPONSE          = 1030;
const RXGK_SERVER_ENC_TOKEN             = 1036;
```

The application of these key usage numbers is specified in Section 8.

4. Security Levels

rxgk supports the negotiation of a range of different security levels. These, along with the protocol constants that represent them during key negotiation, are:

Authentication only (0) Provides only connection authentication, without either integrity or confidentiality protection. This mode of operation provides higher throughput, but is vulnerable to man in the middle attacks. This corresponds to the traditional 'clear' security level.

Integrity (1) Provides integrity protection only. Data is protected from modification by an attacker, but not against eavesdropping. This corresponds to the traditional 'auth' security level, authenticating the data payload as well as the Rx connection.

Encryption (2) Provides both integrity and confidentiality protection. This corresponds to the traditional 'crypt' security level.

The authentication only, or clear, security level provides faster throughput, at the expense of connection security. The 'clear'

security level is vulnerable to a man in the middle altering the data passed over the connection, whereas the 'integrity' security level prevents such attacks by sending a cryptographic checksum of the data being transmitted.

```
enum RXGK_Level {
    RXGK_LEVEL_CLEAR = 0,
    RXGK_LEVEL_AUTH = 1,
    RXGK_LEVEL_CRYPT = 2
};
```

5. Token Format

An rxgk token is an opaque identifier which is specific to a particular application's implementation of rxgk. The token is completely opaque to the client, which just transmits it from server to server. The token MUST permit the receiving server to identify the corresponding user and session key for the incoming connection -- whether that be by decrypting the information within the token, or making the token a large random identifier which keys a lookup table on the server, or some other mechanism. It is assumed that such

mechanisms will conceptually "encrypt" a token by somehow associating the "encrypted" token with the associated unencrypted data, and will "decrypt" an encrypted token by using that association to find the unencrypted data. As such, this document will use "encrypt" and "decrypt" to refer to these operations on tokens. If the token is an encrypted blob, it should be encrypted using the key usage RXGK_SERVER_ENC_TOKEN.

The token MUST NOT expose the session key on the wire. The token MUST be sufficiently random that an attacker cannot predict suitable token values by observing other connections. An attacker MUST NOT be able to forge tokens which convey a particular session key or identity.

6. Key Negotiation

rxgk uses an independent RX RPC service for key negotiation. The location of this service is application dependent. Within a given application protocol, a client MUST be able to locate the key negotiation service, and that service MUST be able to create tokens which can be read by the application server. The simplest deployment has the negotiation service running on every application server, on the same transport endpoints, but using a separate, dedicated, rx service ID.

The rxgk key negotiation service uses the service ID 34567.

GSS security context negotiation requires that the initiator specify a principal name for the acceptor; in the absence of application-specific knowledge, when using rxgk over a port number registered with IANA, the registered service name SHOULD be used to construct the target principal name as <service name>@%lt;hostname%gt; using the name type GSS_C_NT_HOSTBASED_SERVICE.

The key negotiation protocol is defined by the following RPC-L:

```
typedef int RXGK_Enctypes<>;
typedef opaque RXGK_Data<>;

struct RXGK_StartParams {
```

```
        RXGK_Enctypes enctypes;
        RXGK_Level levels<>;
        unsigned int lifetime;
        unsigned int bytelife;
        opaque client_nonce<>;
    };

    struct RXGK_ClientInfo {
        int errorcode;
        int enctype;
        RXGK_Level level;
        unsigned int lifetime;
        unsigned int bytelife;
        rxgkTime expiration;
        opaque mic<>;
        RXGK_Data token;
        opaque server_nonce<>;
    };

    package RXGK_

    GSSNegotiate(IN RXGK_StartParams *client_start,
                 IN RXGK_Data *input_token_buffer,
                 IN RXGK_Data *opaque_in,
                 OUT RXGK_Data *output_token_buffer,
                 OUT RXGK_Data *opaque_out,
                 OUT unsigned int *gss_major_status,
                 OUT unsigned int *gss_minor_status,
                 OUT RXGK_Data *rxgk_info) = 1;
```

The client populates RXGK_StartParams with its preferred options. The enctypes and levels parameters are lists of values supported by

the client, and MUST be ordered from best to worst, with the client's favoured option occurring first within the list. The parameters are:

enctypes: List of encryption types from the Kerberos Encryption Type Number registry created in RFC3961 and maintained by IANA. This list indicates the encryption types that the client is prepared to support.

levels: List of supported rxgk transport encryption levels. See Section 4 for allowed values.

lifetime: The maximum number of seconds that a connection key should be used before rekeying. A value of 0 indicates that the connection should not be rekeyed based on its lifetime. This lifetime is advisory.

bytelife: The maximum amount of data that can be transferred over the connection before it should be rekeyed, expressed as log base 2 of the number of bytes. A value of 0 indicates that there is no limit on the number of bytes that may be transmitted. The byte lifetime is advisory -- a connection that is over its byte lifetime should be permitted to continue, but clients SHOULD attempt to establish a new context at their earliest convenience.

clientnonce: A client-generated string of random bytes, to be used as input to the key generation.

The client then calls gss_init_sec_context() to obtain an output token to send to the server. The GSS service name is application dependent.

The client then calls RXGK_GSSNegotiate, as defined above. This takes the following parameters:

client_start The client params structure detailed above. This should remain constant across the negotiation.

input_token_buffer The token produced by a call to gss_init_sec_context().

opaque_in An opaque token, which was returned by the server following a previous call to GSSNegotiate in this negotiation. If this is the first call, opaque_in should be zero-length.

output_token_buffer The token output by the server's call to

gss_accept_sec_context().

opaque_out An opaque token, which the server may use to preserve state information between multiple calls in the same context negotiation. The client should use this value as opaque_in in its next call to GSSNegotiate.

gss_major_status The major status code output by the server's call to gss_accept_sec_context().

gss_minor_status The minor status code returned by gss_accept_sec_context(). Implementors should note that minor status codes are not portable between GSSAPI implementations.

rxgk_info If gss_major_status == GSS_S_COMPLETE this contains an encrypted block containing the server's response to the client. See below.

The client proceeds through a GSS security context initialization loop, with alternating calls to gss_init_sec_context() and the GSSNegotiate() RPC, until an error or success condition is reached. Each call to GSSNegotiate will return both an output token from GSS_Accept_sec_context() and an output opaque that are to be used as an inputs for a subsequent call to GSSNegotiate, if such a subsequent call is necessary.

Different GSS mechanisms will require a different number of full (or half) round trips. The structure of the loop, with success and error conditions noted (noting that RX level errors may occur as well but are not mentioned as part of the loop structure), is as follows:

The client calls GSS_Init_sec_context(), supplying an input token if one was returned by a previous call to GSSNegotiate(). The client MUST set the mutual_req_flag, conf_req_flag, and integ_req_flag boolans to true.

If the major status code from GSS_Init_sec_context() includes a fatal error code, the negotiation loop is in an error condition and terminates. If the major status code is GSS_S_COMPLETE and and the mutual_state flag is not true, or the major status code is GSS_S_COMPLETE and the conf_avail flag is not true, or the major status code is GSS_S_COMPLETE and the integ_avail flag is not true, the negotiation loop is in an error condition and terminates. If the major status code is GSS_S_COMPLETE and the output token is zero length, this is a success condition and the negotiation loop terminates (this cannot happen on the first iteration of the loop). Otherwise, if the major status

code does not include GSS_S_CONTINUE_NEEDED, the negotiation loop is in an error condition and terminates. If the major status code includes GSS_S_CONTINUE_NEEDED, the output token is sent to the server, per the next step.

The client calls GSSNegotiate(), supplying the output token from GSS_Init_sec_context() and an input opaque if one was returned by a previous call to GSSNegotiate().

The server calls GSS_Accept_sec_context(), supplying the token it received from the client as input. If there is an output token from GSS_Accept_sec_context(), the server returns it to the client in the output_token_buffer field of the GSSNegotiate() RPC, along with the major and minor status codes from the call to GSS_Accept_sec_context(). If the major status code includes GSS_S_CONTINUE_NEEDED, the server also returns an opaque identifier in the opaque_out field of the RPC, which will allow the server to associate a future RPC call with this partially formed security context. If the major status code is GSS_S_COMPLETE, the server constructs an RXGK_ClientInfo structure per below.

The client receives the results of the GSSNegotiate() RPC. If the major status code is not GSS_S_COMPLETE and does not include GSS_S_CONTINUE_NEEDED, the negotiation loop is in an error condition and terminates. If the most recent call to GSS_Init_sec_context() returned the major status code GSS_S_COMPLETE and an output token, the negotiation loop is in a success condition and terminates. If the most recent call to GSS_Init_sec_context() returned a major status code including GSS_S_CONTINUE_NEEDED and the output_token_buffer is zero length, the negotiation loop is in error and terminates. Otherwise, the client proceeds to begin the next cycle of the negotiation loop.

Upon successful completion, rxgk_info contains the XDR representation of a RXGK_ClientInfo structure, encrypted using gss_wrap() with confidentiality protection. The client should decrypt this structure using gss_unwrap().

ClientInfo contains the following server populated fields:

errorcode A policy (rather than connection establishment) error code. If non-zero, an error has occurred, the resulting key negotiation has failed, and the rest of the values in this structure are undefined. These policy error codes are from

com_err tables [COMERR] and may represent such conditions as insufficient authorization or that the client has too many

active connections to the service. Error codes may be RXGK errors (see section Section 10) or from an application-specific table.

enctype The encryption type selected by the server. This SHALL be one of the types listed by the client in its StartParams structure.

level The rxgk security level selected by the server, see Section 4 for allowed values.

lifetime The connection lifetime, in seconds, as determined by the server. The server MAY honor the client's request, but the server MUST choose a value at least as restrictive as the value requested by the client. A value of zero indicates that the connection should not be rekeyed based on its lifetime.

bytelife The maximum amount of data (as log base 2 of the number of bytes) that may be transfered using this key. The server MAY honor the client's request, but the server MUST choose a value at least as restrictive as the value requested by the client. A value of 0 indicates that the connection should not be rekeyed based on the number of bytes transmitted over the connection.

expiration The time, expressed as an rxgkTime, at which this token expires. The expiration time MAY be set administratively by the server, and SHOULD reflect the expiration time of the underlying GSSAPI credential. The token SHOULD NOT expire later than the underlying GSSAPI credential.

mic The result of calling gss_get_mic() [RFC2744] over the XDR encoded representation of the StartParams request received by the server.

token An rxgk token. This is an opaque blob, as detailed in Section 5.

server_nonce The nonce used by the server to create the K0 used

within the rxgk token.

Upon receiving the server's response, the client must verify that the mic contained within it matches the MIC of the XDR representation of the StartParams structure it sent to the server (this prevents a man in the middle from performing a downgrade attack). The client SHOULD also verify that the server's selected connection properties match those proposed by the client.

The client may then compute K0, by taking the nonce it sent to the server (client_nonce) and the one it has just received (server_nonce), combining them together, and passing them to gss_psuedo_random() with the GSS_C_PRF_KEY_FULL option:

```
        gss_pseudo_random(gssapi_context,
                          GSS_C_PRF_KEY_FULL,
                          client_nonce || server_nonce,
                          K,
                          *K0);
```

|| is the concatenation operation.

K, the desired output length, is the key generation seed length as specified in the RFC3961 profile of the negotiated enctype.

The ouput of gss_pseudo_random must then be passed through the random-to-key operation specified in the RFC3961 profile for the negotiated enctype in order to obtain the actual key K0.

The gss_pseudo_random() operation is deterministic, ensuring that the client and server generate the same K0. The gssapi_context parameter is the same context used in the client's gss_init_sec_context() call and the server's gss_accept_sec_context() call.

7. Combining Tokens

7.1. Overview

A client may elect to combine multiple rxgk tokens in its possession into a single token. This allows an rx connection to be secured

using a combination of multiple, individually established identities, which provides additional security for a number of application protocols.

Token combination is performed using the CombineTokens RPC call. The client has two keys -- K0 and K1, and two tokens, T0 and T1. The client calls the CombineTokens RPC with T0 and T1 and negotiates the enctype and security level of the new token, received as Tn. Tn contains the new key Kn, as computed by the server. Using the negotiated enctype returned by the server, the client then locally combines the two keys using a defined combination algorithm to produce Kn.

7.2. Key Combination Algorithm

Assume that the tokens being combined are T0 and T1, with initial keys K0 and K1. The new initial key for the combined token, Kn is computed using the KRB-FX-CF2 operation, described in section 5.1 of [RFC6113]. The PRF+ operations will correspond to their respective key enctypes, and the random-to-key operation will correspond to the negotiated new enctype. The constants pepper1 and pepper2 required by this operation are defined as the ASCII strings "AFS" and "rxgk" respectively.

7.3. RPC Definition

The combine keys RPC is defined as:

```
struct RXGK_CombineOptions {
    RXGK_Enctypes enctypes;
    RXGK_Level levels<>;
};

struct RXGK_TokenInfo {
    int errorcode;
    RXGK_Enctype enctype;
    RXGK_Level level;
    unsigned int lifetime;
```

```
        unsigned int bytelife;
        rxgkTime expiration;
    }

    CombineTokens(IN RXGK_Data *token0, IN RXGK_Data *token1,
                  IN RXGK_CombineOptions *options,
                  OUT RXGK_Data *new_token,
                  OUT RXGK_TokenInfo *info) = 2;
```

7.4. Server Operation

The server receives token0 and token1 from the RPC call, as well as the options suggested by the client. Upon receipt, the server decrypts these tokens using its private key. Providing this decryption is successful, it now has copies of the initial key (K0) from both tokens. The server then chooses an enctype and security level from the lists supplied by the client in the options argument. The server SHOULD select the first entry from each list which is acceptable in the server's configuration, so as to respect any preferences indicated by the client. The server then performs the key combination algorithm detailed above to obtain the new key, Kn. The server then constructs a new token as follows. The expiration is set to the minimum of the expiration values of the original tokens.

The lifetime, bytelife, and any application-specific data fields are each combined so that the result is the most restrictive of the two values in each of the original tokens. The identity information associated with the tokens are combined in an application-specific manner to yield the identity information in the combined token (the identity combining operation may be non-commutative). This new token contains the derived key, Kn. The new token is encrypted with the server's private key, as normal, and returned to the client. The enctype and level chosen by the server are returned in the info parameter, along with the computed lifetime, bytelife, and expiration. If the server is unable to perform the CombineTokens operation with the given arguments, a nonzero value is returned in the errorcode element of the info parameter; errorcode is zero for a successful CombineTokens operation. If errorcode is nonzero, the values of the other fields in the RXGK_TokenInfo structure and the value of new_token are undefined. Nonzero values for errorcode should be com_err codes [COMERR], from an RX, RXGK, or application-specific table. See section Section 10 for RXGK error codes. section

Section 8.3. For example,

RXGEN_OPCODE used when the server will refuse all CombineTokens requests.

RXGK_BAD_ETYPE used when none of the enctypes supplied by the client are acceptable to the server.

RXGK_BAD_LEVEL used when one of the security levels supplied by the client are acceptable to the server.

RXGK_EXPIRED used when one or more of the input tokens was already expired.

To reduce the potential for denial of service attacks, servers SHOULD only offer the CombineTokens operation to clients connecting over a secured rxgk connection. CombineTokens SHOULD NOT be offered over an RXGK_LEVEL_CLEAR connection.

7.5. Client Operation

As detailed within the overview, the client calls the CombineTokens RPC using two tokens, T0 and T1, within its possession, as well as an RXGK_CombineOptions structure containing a list of acceptable enctypes and a list of acceptable security levels for the new token. The client SHOULD supply these lists sorted by preference, with the most preferred option appearing first in the list. The client then receives a new token, Tn, from this call, as well as an RXGK_TokenInfo structure containing information relating to Tn. The client needs the level element of the info parameter to determine

what security level to use the new token at, and the enctype parameter to know which enctype's random-to-key function to use in generating Kn. With the negotiated enctype, the client can then perform the key combination algorithm described in section Section 8.3. The client can only make use of Tn to establish an rxgk protected connection if it can derive Kn, which it can only do if it already knows K0 and K1.

Clients MUST use an rxgk secured connection for the CombineTokens operation.

8. The rxgk Security Class

8.1. Overview

When a new connection using rxgk is created by the client, the client stores the current timestamp as an rxgkTime (start_time for the rest of this discussion), and then uses this, along with other connection information, to derive a transport key from the current user's master key (see Section 8.3).

This key is then used to protect the first message the client sends to the server. The server follows the standard RX security establishment protocol, and responds to the client with a challenge [RX]. rxgk challenges simply contain a random nonce selected by the server.

Upon receiving this challenge, the client uses the transport key to encrypt an authenticator, which contains the server's nonce, and some other connection information. The client sends this authenticator, together with start_time and the current user's rxgk token, back to the server.

The server decrypts the rxgk token to determine the master key in use, uses this to derive the transport key, which it in turn uses to decrypt the authenticator, and thus validate the connection.

8.2. Rekeying

As part of connection negotiation, the server and client agree upon advisory lifetimes (both time, and data, based) for connection keys. Each connection has a key number, which starts at 0. When a connection exceeds one of its lifetimes, either side may elect to increment the key number. When the other endpoint sees a key number increment, it should reset all of its connection counters. Endpoints should accept packets encrypted with either the current, previous, or next key number, to allow for resends around the rekeying process.

The key version number is contained within the 16 bit spare field of the RX header (used by previous security layers as a checksum field), and expressed as an unsigned value in network byte order. If rekeying would cause this value to wrap, then the key version number

MAY be stored locally as a 32-bit integer on both endpoints with only the low 16 bits transmitted on the wire. If an endpoint cannot store a per-connection 32-bit key version number when the 16-bit key version number would wrap, that endpoint MUST terminate the connection.

8.3. Key Derivation

In order to avoid the sharing of keys between multiple connections, each connection has its own transport key, TK, which is derived from the master key, K0. Derivation is performed using the PRF+ function defined in [RFC4402], combined with the random-to-key function of K0's encryption type, as defined in RFC3961. The PRF input data is the concatenation of the rx epoch, connection ID, start_time and key number, all in network byte order. This gives:

```
TK = random-to-key(PRF+(K0, L,
                        epoch || cid || start_time || key_number))
```

L is the key generation seed length as specified in the RFC3961 profile.

epoch, cid and key_number are passed as 32 bit quantities, start_time is a 64 bit value.

Note that start_time is selected by the client when it receives the server's challenge, and shared with the server as part of its response. Thus both sides of the negotiation are guaranteed to use the same value for start_time.

8.4. The Challenge

The rxgk challenge is an XDR encoded structure with the following signature:

```
struct RXGK_Challenge {
    opaque nonce[20];
};
```

nonce: 20 octets of random data.

8.5. The Response

The rxgk response is an XDR encoded structure, with the following signature:

```
struct RXGK_Response {
    rxgkTime start_time;
    RXGK_Data token;
    opaque authenticator<>
};
```

start_time: The time since the Unix epoch (1970-01-01 00:00:00Z), expressed as an rxgkTime (see Section 2).

authenticator: The XDR encoded representation of an RXGK_Authenticator, encrypted with the transport key, and key usage RXGK_CLIENT_ENC_RESPONSE.

8.5.1. The Authenticator

```
struct RXGK_Authenticator {
    opaque nonce[20];
    opaque appdata<>
    RXGK_Level level;
    unsigned int epoch;
    unsigned int cid;
    unsigned int call_numbers<>;
};
```

nonce: A copy of the nonce from the challenge.

appdata: An application specific opaque blob.

level: The desired security level for this particular connnection. This MUST NOT be less than the security level originally negotiated.

epoch: The rx connection epoch.

cid: The rx connection ID.

call_numbers: The set of current rx call numbers for all available channels; unused channels should report a call number of zero. The length of this vector indicates the maximum number of calls per connection supported by the client.

8.6. Checking the Response

To check the validity of an rxgk response, the authenticator should be decrypted, the nonce from the decrypted authenticator compared with the nonce sent in the RXGK_Challenge, and the connection ID and epoch compared with that of the current connection. The call number vector (call_numbers) should be supplied to the rx implementation. Failure of any of these steps MUST result in the failure of the security context.

8.7. Packet Handling

The way in which the rxgk security class handles packets depends upon the requested security level. As noted in Section 4, 3 levels are currently defined -- authentication only, integrity protection and encryption.

8.7.1. Authentication Only

When running at the clear security level, RXGK_LEVEL_CLEAR, no manipulation of the payload is performed by the security class.

8.7.2. Integrity Protection

Packet payloads transmitted in the auth security level, RXGK_LEVEL_AUTH, consist of an opaque blob of MIC data followed by the unencrypted original payload data.

The MIC data is generated by calling the RFC3961 get_mic operation using a key and a data input. The RXGK_CLIENT_MIC_PACKET key usage number MUST be used for packets transmitted from the client to the server. The RXGK_SERVER_MIC_PACKET key usage number MUST be used for packets transmitted from the server to the client. The following data structure is the get_mic operation data input:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             epoch                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              cid                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          call number                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           sequence                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        security index                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          data length                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                    original packet payload                    ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

All fields MUST be in network byte order. The data length field specifies the length of the original packet payload excluding padding required for encryption routines.

The packet is transmitted with the following payload:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                              MIC                              ~
|                                                               |
|               +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
|               |                                               |
+-+-+-+-+-+-+-+-+                                               |
|                                                               |
~                     original packet payload                   ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Note: The length of the MIC depends on which RFC3961 encryption type is used. In particular, the original packet payload may not be word-aligned.

Note: The data prepended to the original packet payload during the MIC generation is not transmitted.

8.7.3. Encryption

Using the encryption security level, RXGK_LEVEL_CRYPT, provides both integrity and confidentiality protection.

The existing payload is prefixed with a psuedo header, to produce the following plaintext data for encryption before transmission. All fields MUST be represented in network byte order for encryption.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             epoch                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              cid                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          call number                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            sequence                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         security index                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          data length                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                     original packet payload                   ~
```

```
   |                                                               |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The data length is the length of the following data in octets, and is necessary so the receiving end can remove any padding added by the encryption routines.

This plaintext is encrypted using an RFC3961 style encrypt() function, with the connection's transport key, using key usage RXGK_CLIENT_ENC_PACKET for messages from client to server, and RXGK_SERVER_ENC_PACKET for messages from server to client, and the encrypted block transmitted to the peer.

9. RXGK protocol error codes

This document specifies several erorr codes for use by RXGK implementations (see section Section 10 for the com_err table). In general, when an endpoint receives any such error code, it should abort the current operation. The various codes allow some information about why the operation failed to be conveyed to the peer

so that future requests will be more likely to succeed. The circumstances in which each error code should be used are as follows:

RXGK_INCONSISTENCY Used for errors internal to the security class, such as when invariant assertions are violated. For example, when an incoming packet to a server contains flags that do not match the server's idea of the connection state, or attempting to allocate a new connection where a connection already exists.

RXGK_PACKETSHORT The size of the packet is too small. Used when a server is constructing a challenge packet but the required data would be larger than the server's allowed packet size. Used when a reply packet received by the server is smaller than the expected size of a response packet. Also for the analogous situations on the other side of the challenge/response exchange.

RXGK_BADCHALLENGE A challenge or response packet (of the expected size) failed to decode properly or contained nonsense or

useless data.

RXGK_BADETYPE Used when the supplied encryption type(s) are invalid or impermissible, such as for the GSSNegotiate and CombineTokens RPCs, when the client-supplied enctype list does not contain any entries that are acceptable to the server.

RXGK_BADLEVEL Used when the supplied security level(s) are invalid or impermissible, such as for the GSSNegotiate and CombineTokens RPCs, when the client-supplied list of security levels does not contain any entries that are acceptable to the server.

RXGK_BADKEYNO The client or client's token indicates the use of a key version number that is not present on the server. May also be used when a key is presented that is not a valid key.

RXGK_EXPIRED The client presented an expired credential or token.

RXGK_NOTAUTH The caller is not authorized for the requested operation or the presented credentials are invalid. In particular, may also be used when credentials are presented that have a start time in the future. Note that many application error tables already include codes for "permission denied", which take precedence over this general error code.

RXGK_BAD_TOKEN The client failed to present a token or the presented token is invalid. For cases including but not limited to: wrong size, fails to decode, zero or negative lifetime, starts too far in the future, and too long a lifetime.

RXGK_SEALED_INCON Encrypted or checksummed data does not verify or correctly decode. The checksum is invalid, the sealed copy of the sequence and/or call number does not match the current state, or similar situations.

RXGK_DATA_LEN The packet is too large, contains a zero-length iovec entry, or otherwise presents an unacceptable or invalid data

length.

10. AFS-3 Registry Considerations

This document requests that the AFS-3 registrar include a com_err error table for the RXGK module, as follows:

```
error_table RXGK
ec RXGK_INCONSISTENCY, "Security module structure inconsistent"
ec RXGK_PACKETSHORT, "Packet too short for security challenge"
ec RXGK_BADCHALLENGE, "Invalid security challenge"
ec RXGK_BADETYPE, "Invalid or impermissible encryption type"
ec RXGK_BADLEVEL, "Invalid or impermissible security level"
ec RXGK_BADKEYNO, "Key version number not found"
ec RXGK_EXPIRED, "Token has expired"
ec RXGK_NOTAUTH, "Caller not authorized"
ec RXGK_BAD_TOKEN, "Security object was passed a bad token"
ec RXGK_SEALED_INCON, "Sealed data inconsistent"
ec RXGK_DATA_LEN, "User data too long"
end
```

The error table base should be 1233242880, with codes within the table assigned relative numbers starting from 0 in the order appearing above.

11. IANA Considerations

This memo includes no request to IANA.

12. Security Considerations

12.1. Abort Packets

RX Abort packets are not protected by the RX security layer. Therefore caution should be exercised when relying on their results. In particular, clients MUST NOT use an error from GSSNegotiate or

CombineTokens to determine whether to downgrade to another security class

12.2. Token Expiry

This document permits tokens to be issued with expiration times after the expiration time of the underlying GSSAPI credential, though implementations SHOULD NOT do so. Allowing the expiration time of a credential to be artificially increased can break the invariants assumed by a security system, with potentially disastrous consequences. For example, with the krb5 GSSAPI mechanism, access revocation may be implemented by refusing to issue new tickets (or renew existing tickets) for a principal; all access is assumed to be revoked once the maximum ticket lifetime has passed. If an rxgk token is created with a longer lifetime than the kerberos ticket, this assumption is invalid, and the user whose access has supposedly been revoked may gain access to sensitive materials. An application should only allow token expiration times to be extended after a security review of the assumptions made about credential expiration for the GSSAPI mechanism(s) in use with that application. Such a review is needed to confirm that allowing token expiration times to be extended will not introduce vulnerabilities into the security eocsystem in which the application operates.

13. References

13.1. Informational References

[RX] Zeldovich, N., "RX protocol specification".

[COMERR] Raeburn, K., "A Common Error Description Library for UNIX".

This paper is available as com_err.texinfo within com_err.tar.Z.

13.2. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2743] Linn, J., "Generic Security Service Application Program

Interface Version 2, Update 1", RFC 2743, January 2000.

[RFC2744] Wray, J., "Generic Security Service API Version 2 : C-bindings", RFC 2744, January 2000.

[RFC3961] Raeburn, K., "Encryption and Checksum Specifications for Kerberos 5", RFC 3961, February 2005.

[RFC4120] Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The Kerberos Network Authentication Service (V5)", RFC 4120, July 2005.

[RFC4401] Williams, N., "A Pseudo-Random Function (PRF) API Extension for the Generic Security Service Application Program Interface (GSS-API)", RFC 4401, February 2006.

[RFC4402] Williams, N., "A Pseudo-Random Function (PRF) for the Kerberos V Generic Security Service Application Program Interface (GSS-API) Mechanism", RFC 4402, February 2006.

[RFC4506] Eisler, M., "XDR: External Data Representation Standard", STD 67, RFC 4506, May 2006.

[RFC6113] Hartman, S. and L. Zhu, "A Generalized Framework for Kerberos Pre-Authentication", RFC 6113, April 2011.

Appendix A. Acknowledgements

rxgk was originally developed over a number of AFS Hackathons. The editor of this document has assembled the protocol description from a number of notes taken at these meetings, and from a partial implementation in the Arla AFS client.

Thanks to Derrick Brashear, Jeffrey Hutzelman, Love Hornquist Astrand and Chaskiel Grundman for their original design work, and comments on this document, and apologies for any omissions or misconceptions in my archaeological work.

Marcus Watts and Jeffrey Altman provided invaluable feedback on an earlier version of this document at the 2009 Edinburgh AFS Hackathon.

The text describing the rxgkTime type is based on language from Andrew Deason.

Appendix B. Changes

B.1. Since 00

Add a reference to RFC4402, which describes the PRF+ algorithm we are using.

Change references to RXGK_Token to RXGK_Data for clarity, and add a definition of that type.

Rename the 'ticket' member of RXGK_ClientInfo to 'token', for consistency, and make it a simple opaque.

Add a length field to the packet header, so that we can remove padding.

Remove versioning in the challenge and the response.

Clarify that both bytelife and lifetime are advisory.

Remove the RXGK_CLIENT_COMBINE_ORIG and RXGK_SERVER_COMBINE_NEW key derivations, as these are no longer used.

Update the reference to draft-ietf-krb-wg-preauth-framework.

Require that CombineTokens be offered over an rxgk authenticated connection.

Pull our time definition out into its own section and define a type for it.

Define an enum for the security level, and use that throughout.

B.2. Since 01

Spell check.

Remove a couple of stray references to afs_ types.

Update start_time text to clarify that it uses rxgkTime.

Make expiration also be an rxgkTime.

Add a definition for RXGK_LEVEL_BIND.

Add reference to RX.

Add reference to XDR.

Rename the gss_status output parameter from the GSSNegotiate RPC to gss_major_status, and update the supporting text.

Add a new gss_minor_status output paramter to the GSSNegotiate RPC, but make clear that it is there for informational use only.

B.3. Since 02

Edit for grammar and punctuation.

Remove RXGK_LEVEL_BIND.

Make CombineTokens negotiate level and enctype.

Allow key version rollover at 16 bits when rekeying.

Add Security Considerations for increasing token expiry.

Clarify behavior at RXGK_LEVEL_AUTH.

Add RXGK com_err table and descriptions.

Clean up call number vector and maxcalls support.

Improve the description of the GSS negotiation loop.

Give suggestions for acceptor principal names.

Author's Address

Simon Wilkinson
Your File System Inc

Email: simon@sxw.org.uk