Network Working Group
Internet-Draft
Intended status: Informational
Expires: July 13, 2012

Wilkinson
YFS

January 10, 2012

rxgk: GSSAPI based security class for RX
draft-wilkinson-afs3-rxgk-02

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 July 13, 2012.

Table of Contents

1. Introduction

rxgk is a GSSAPI [RFC2743] based security class for the rx protocol. It provides authentication, confidentiality and integrity protection for rx [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 in 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_TICKET          = 1036;
```

4. Security Levels

rxgk supports the negotiation of a range of different security levels. These, along with the protocol constant that represents 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' level.

Encryption (2) Provides both integrity and confidentiality protection, corresponding to 'crypt'

Bind (3) Connection security is provided by channel bindings with another layer. This mode of operation is experimental, and this value is reserved for future expansion.

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


```
enum RXGK_Level {
    RXGK_LEVEL_CLEAR = 0;
    RXGK_LEVEL_AUTH = 1;
    RXGK_LEVEL_CRYPT = 2;
    RXGK_LEVEL_BIND = 3;
};
```

5. Token Format

An rxgk token is an opaque identifier which is specific to an 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 hash table on the server.

The token MUST NOT expose the session key on the wire. It 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 service running on every server, on the same transport endpoints, but using a separate, dedicated, rx service id.

The key negotiation RPC is defined by the following XDR


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

struct RXGK_StartParams {
    RXGK_Enctypes enctypes;
    RXGK_Level levels<>;
    int lifetime;
    int bytelife;
    opaque client_nonce<>;
};

struct RXGK_ClientInfo {
    int errorcode;
    int flags;
    int enctype;
```

```
        RXGK_Level level;
        int lifetime;
        int bytelife;
        rxgkTime expiration;
        opaque mic<>;
        opaque token<>;
        opaque server_nonce<>;
    };

    package RXGK_

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

The client populates RXGK_StartParams with lists of its preferred options. These should be ordered from best to worst, with the clients 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.

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 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

clientparms 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, this should be NULL.

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 negotiate. 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.

Upon receiving the server's response, the client checks the contents of gss_major_status. If this is GSS_S_CONTINUE_NEEDED, the client should call gss_init_sec_context again with the token provided by the server in output_token_buffer, followed by a further call to GSSNegotiate, including the server's previous opaque_out as this call's opaque_in

This process continues until the either the server, or client, encounters an error, or the server returns GSS_S_COMPLETE in gss_major_status.

Upon 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.

flags

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

level The rxgk security level selected by the server.

lifetime The connection lifetime, in seconds, as determined by the server (this must be less than or equal to the lifetime proposed by the client)

bytelife The maximum amount of data (in log 2 bytes) that may be transfered using this key. This must be less than or equal to the bytelife proposed by the client

expiration The time, expressed as an rxgkTime, at which this token expires

mic The result of calling gss_get_mic over the XDR encoded representation of the StartParams request received by the server.

token An rxgk token. This is an opaque blob, as detailed earlier

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). It should also verify that the server's selected connection properties match those it proposed.

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

7. The combine tokens operation

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. It

locally combines the two keys using a defined combination algorithm to produce Kn. It then calls the CombineTokens RPC with T0 and T1, to receive a new token, Tn, which has embedded within it Kn, as computed by the server.

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 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

```
    CombineTokens(IN opaque token0,
                  IN opaque token1,
                  OUT opaque new_token) = 2;
```

7.4. Server operation

The server receives both token0 and token1 from the RPC call, and decrypts these tokens using its private key. Providing this decryption is successful, it now has copies of the initial key (K0) from both tokens. It then performs the key combination algorithm detailed above to obtain a new key, Kn. The server constructs a new token, where each of the numerical fields are set to the minimum of the values of each of the original tokens, and the list of identities is the union of those in the original tokens. 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.

To reduce the potential for denial of service attacks, servers SHOULD only offer the CombineTokens operation to clients connecting over an rxgk secured 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. It then receives a new token, Tn from this call. 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, it stores the current timestamp (as 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.

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. rxgk challenges simply contain some versioning information and 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 a

number of 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 endpoint performing the rekey 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;
    opaque token<>
    opaque authenticator<>
};
```


start_time: the time since the Unix epoch (1970-01-01 00:00:00Z), expressed as an rxgkTime

authenticator: the XDR encoded representation of 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<>
    unsigned int epoch;
    unsigned int cid;
    unsigned int maxcalls;
    unsigned int call_numbers<>;
};
```

nonce: a copy of the nonce from the challenge

appdata: an application specific opaque blob

epoch: the rx connection epoch

cid: the rx connection ID

maxcalls: the highest rx call number in use

call_numbers: the set of current rx call numbers

8.6. Checking the Response

To check the validity of an rxgk response, the authenticator should be decrypted, the nonce compared with that sent in the challenge, and the connection ID and epoch compared with that of the current connection. Failure of any of these steps MUST result in the failure of the security context.

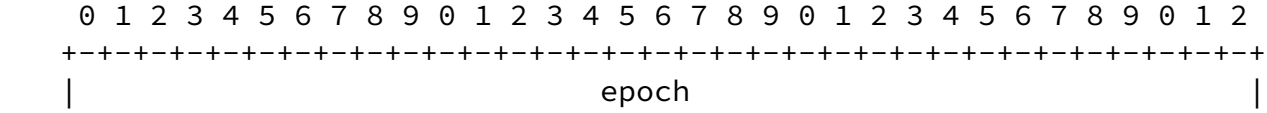
8.7. Packet handling

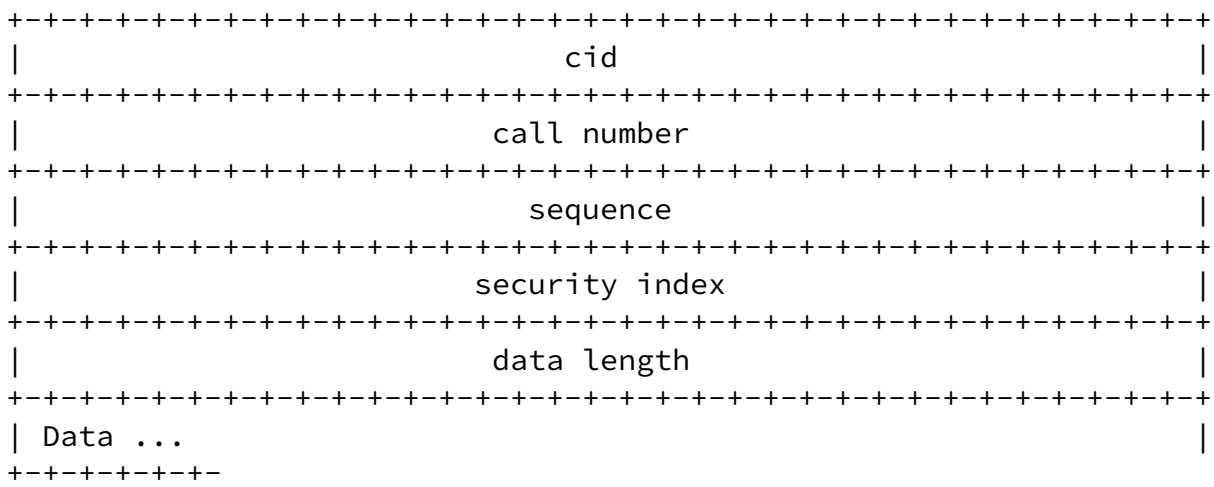
The way in which the rxgk security class handles packets depends upon the requested security level. As noted earlier, 3 levels are currently defined - authentication only, integrity protection and encryption

8.7.1. Encryption

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

The existing payload is prefixed with a psuedo header, to produce the following data for encryption.





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.

8.7.2. Integrity protection

The rxgk_auth security level prepends the packet with the same data block as crypt (as detailed above), and then calls the RFC3961 get_mic operation over the result, using key usage RXGK_CLIENT_MIC_PACKET for messages from client to server, and RXGK_SERVER_MIC_PACKET for messages from server to client.

The peer is sent the output from the MIC operation, followed by the original payload (excluding the additional header which was added for the MIC step).

Upon receiving a protected packet, the receiver should consult the RFC3961 profile for the encryption algorithm in use to determine how many bytes of checksum are contained within the packet. Having split the data into checksum and payload using this information, the

checksum should be verified using the encryption profile's verify_mic() operation with the appropriate key derivation.

Note that the checksum field within the rx packet header itself is not used, as it is too small to hold a collision proof checksum value.

8.7.3. Authentication only

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

9. IANA Considerations

This memo includes no request to IANA.

10. Security Considerations

10.1. Abort Packets

RX Abort packets are not protected by the 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

11. References

11.1. Informational References

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

11.2. 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.

[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

Author's Address

Simon Wilkinson
Your File System Inc

Email: simon@sxw.org.uk