Network Working Group
Internet-Draft
Intended status: Informational
Expires: August 7, 2014

S. Wilkinson
YFS
B. Kaduk
MIT

February 3, 2014

Integrating rxgk with AFS
draft-wilkinson-afs3-rxgk-afs-04

Abstract

This document describes how the new GSSAPI-based rxgk security class for RX is integrated with the AFS application protocol. It describes a number of extensions to the basic rxgk protocol, clarifies a number of implementation issues, and provides values for the application-specific elements of rxgk.

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 August 7, 2014.

Copyright Notice

Table of Contents

1. Introduction

rxgk [I-D.wilkinson-afs3-rxgk] is a new GSSAPI-based [RFC2743] security layer for the RX [RX] remote procedure call system. The rxgk specification details how it may be used with a generic RX application. This document provides additional detail specific to integrating rxgk with the AFS-3 distributed file system.

1.1. The AFS-3 Distributed File System

AFS-3 is a global distributed network file system. The system is split into a number of cells, with a cell being the administrative boundary. Typically an organisation will have one (or more) cells, but a cell will not span organisations. Each cell contains a number of fileservers which contain collections of files ("volumes") which they make available to clients using the AFS-3 protocol. Clients access these files using a service known as the cache manager.

In order to determine which server a particular file is located upon, the cache manager looks up the location in the volume location database (vldb) by contacting the vlserver. Each cell has one or more vlservers, which are synchronised using an out-of-band mechanism.

1.2. rxgk and AFS

This document describes the special integration steps needed to use rxgk with AFS-3 database servers (the PR and VL rx services) and file servers (the RXAFS, RXAFSCB, and AFSVol rx services), as well as specifying application-specific portions of the rxgk specification for use by these services. Other AFS-3 services are not covered by this document; the generic rxgk document applies to them.

AFS-3 differs from a standard rxgk deployment in that it does not require GSSAPI negotiation with each server. Instead, a client performs GSSAPI negotiation just once, with the vlserver, receiving a token usable with any database server in the cell. Traditional AFS rxkad authentication required that the cell-wide key be distributed to all servers in the cell, both database servers and file servers, making no distinction between tokens used for database servers and file servers. rxgk can operate in such a fashion, with a cell-wide key shared amongst all servers, but is not limited to doing so.

For more complex cell topologies, rxgk also supports configurations

where (some) file servers do not have the cell-wide key. Tokens for these file servers are obtained by means of an extended version of the CombineTokens RPC, AFSCombineTokens, which takes a server identifier and will return a token encrypted in the key for a specific file server. AFSCombineTokens also provides a mechanism for indicating whether a specific server is rxgk capable, allowing cells to securely migrate to rxgk from other security mechanisms.

We also define mechanisms for securing the callback channel that is created between fileserver and client.

1.3. 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. Security Index

When used within the AFS protocol, rxgk has an RX securityIndex value of 4.

3. Key Negotiation

An AFS cell wishing to support rxgk MUST run an rxgk key negotiation service, as specified in [I-D.wilkinson-afs3-rxgk], on each of its vlservers. The service MUST listen on the same port as the vlserver.

The GSS identity afs-rxgk@_afs.<cellname> of nametype GSS_C_NT_HOSTBASED_SERVICE is the acceptor identity for this service. Where multiple vlservers exist for a single cell, all of these servers must have access to the key material for this identity, which MUST be identical across the cell. Clients MAY use the presence of this identity as an indicator of rxgk support for a particular cell. Clients that wish to support cells using other rx security objects MAY downgrade if this identity is not available.

Tokens returned from the GSSNegotiate and CombineTokens calls MUST only be used with database servers. Tokens for fileservers MUST be obtained by calling AFSCombineTokens before each server is contacted.

rxgk tokens are in general only usable with the particular rx service that produced them. For the AFS-3 protocol, the database server services are grouped together to accept a common type of token, and the file server services are grouped together to accept a different common type of token, but it is important to emphasize that a token for a database server will not in general be useful against a file server, and vice versa. Tokens for database servers are obtained from the standard rxgk negotiation services, but tokens for file servers are obtained through a new procedure, the AFSCombineTokens RPC described in Section 3.1.

3.1. The AFSCombineTokens Operation

AFS extends the existing CombineTokens operation to provide a more featured token manipulation and conversion service. This operation takes a user token, an optional cache manager token, options for enctype and security level negotiation with the server, and a destination file server identifier. It returns a token specific to the specified destination, and a structure containing some

information describing the returned token. AFSCombineTokens is the only way to obtain a valid file server token (other than printing a token, see Section 7.1).

```
AFSCombineTokens(IN RXGK_Data *token0,
                 IN RXGK_Data *token1,
                 IN RXGK_CombineOptions *options,
                 IN afsUUID destination,
                 OUT RXGK_Data *new_token,
                 OUT RXGK_TokenInfo *token_info) = 3;
```

token0: An rxgk token for the vlserver.

token1: Either an rxgk token for the vlserver, or empty (zero-length).

options: An RXGK_CombineOptions structure containing a list of enctypes acceptable to the client and a list of security levels acceptable to the client.

destination: The UUID of the server new_token is intended for. Fileserver UUIDs may be obtained from the VLDB in the same call

that returns their addresses.

new_token: The output rxgk token, or empty (zero-length).

token_info: Information describing the returned token.

The AFSCombineTokens call MUST only be performed over a secured rxgk connection. AFSCombineTokens MUST NOT be offered over an RXGK_LEVEL_CLEAR connection. Servers MUST reject all attempts to perform this operation over channels that do not offer integrity protection. This integrity guarantee protects the returned token information (token_info) as well as the options and destination arguments submitted to the server.

Clients which are caching the results of RPCs on behalf of multiple users (such as a traditional AFS Cache Manager), SHOULD provide both the user's token (as token0) and a token generated from an identity that is private to the cache manager (as token1). This prevents a user from poisoning the cache for other users. Recommendations on keying cache managers are contained in Section 6.1.

Clients which are working on behalf of a single user can provide an empty token1, but MUST use AFSCombineTokens to obtain a destination-specific token for each fileserver they contact.

The output token from AFSCombineTokens is a token specific to the fileserver indicated by the destination argument. As such, it is not a valid input token for a successor AFSCombineTokens operation, as the input tokens for AFSCombineTokens must be tokens for the vlserver.

Clients using a printed token (see Section 7.1) MUST provide that token as token0. token1 MUST be empty. Printed tokens cannot be combined with any other token, and servers MUST reject attempts to do so, whether via the CombineTokens RPC, the AFSCombineTokens RPC, or any other token-combining procedure.

If the server is unable to perform the AFSCombineTokens operation with the given arguments, a nonzero value is returned. Clients MUST NOT use such an error as an indication to fall back to to a different

security class.

If the returned token is zero-length, then the destination does not support rxgk, and the client MAY fall back to using a different authentication mechanism for that server. An rxgk capable client operating within an rxgk enabled cell MUST NOT downgrade its choice of security layer in any other situation.

Other aspects of the operation of AFSCombineTokens, including the combination of keys and tokens, are largely the same as the CombineTokens RPC, documented in CombineTokens call, documented in [I-D.wilkinson-afs3-rxgk]. The only differences pertain to the case where the supplied token1 is empty. In this case, there is only one input token master key, so the KRB-FX-CF2() algorithm is not applicable; in these cases the master key K0 from token0 is reused unchanged for the output token. The case where token1 is empty also requires special handling for the identities list in new_token, which is discussed in Section 4.3.

The combination of identity information into the new_token is not specified in [I-D.wilkinson-afs3-rxgk], being left as application-specific function. For the AFS-3 protocols, identity information in tokens is stored as a list of PrAuthNames, and the identity combination algorithm is order-preserving concatenation.

4. Tokens

4.1. Container

rxgk tokens for AFS take the form of some key management data, followed by an encrypted data blob. The key management data (a version number, followed by an [RFC3961] encryption type) allows the

server receiving a token to identify which key has been used to encrypt the core token data.

```
struct RXGK_TokenContainer {
    afs_int32 kvno;
    afs_int32 enctype;
    opaque    encrypted_token<>;
};
```

The RXGK_TokenContainer structure is XDR encoded and transported within the 'token' field of the RXGK_ClientInfo structure specified in [I-D.wilkinson-afs3-rxgk].

4.2. Token Encryption

rxgk supports encrypting tokens both with a single cell-wide keys and with per-file-server keys. The cell-wide key must be installed on all database servers in the cell, and may additionally be installed on non-database file servers if the use of per-file-server keys is not desired. Cell-wide keys should be for a selected RFC3961 encryption mechanism that is supported by all servers within the cell that will use that key. Per-server keys should be for an encryption mechanism that is supported by both the destination server and the negotiation service. The management of per-server keys is discussed in more detail in Section 13.2.

Key rollover is permitted by means of a key version number. When a key is changed, whether cell-wide or per-server, a different (larger) key version number MUST be selected. Servers SHOULD accept tokens using old keys until the lifetime of all non-printed (see Section 7.1 existing tokens has elapsed. Services using printed tokens should be prepared to regenerate those tokens in the case of key rollover.

Encryption is performed over the XDR encoded RXGK_Token structure, using the RFC3961 encrypt operation, with a key usage value of RXGK_SERVER_ENC_TOKEN (defined in [I-D.wilkinson-afs3-rxgk]). The enrypted data is stored in the encryted_token field of the RXGK_TokenContainer structure described in Section 4.1.

4.3. Token Contents

The token itself contains the information expressed by the following RPC-L:


```
struct RXGK_Token {
```

```
        afs_int32 enctype;
        opaque K0<>;
        RXGK_Level level;
        rxgkTime start_time;
        afs_int32 lifetime;
        afs_int32 bytelife;
        rxgkTime expirationtime;
        struct PrAuthName identities<>;
    };
```

enctype: The RFC3961 encryption type of the session key contained within this ticket.

K0: The session key (see [I-D.wilkinson-afs3-rxgk] for details of how this key is negotiated between client and negotiation service).

level: The security level, as defined in [I-D.wilkinson-afs3-rxgk], that MUST be used for this connection.

start_time: The time at which the token's validity begins. Servers MUST reject attempts to use tokens with a start_time value later than the current time.

lifetime: The maximum number of seconds that a key derived from K0 may be used for, before the connection is rekeyed. If 0, keys have no time-based limit.

bytelife: The maximum amount of data (expressed as the log base 2 of the number of bytes) that may be transferred using a key derived from K0 before the connection is rekeyed. If 0, there is no data-based limit on key usage.

expirationtime: The time (expressed as an rxgkTime) beyond which this token may no longer be used. Servers MUST reject attempts to use connections secured with this token after this time. A value of 0 indicates that this token never expires.

identities: A list of identities represented by this token. struct PrAuthName is the identity structure defined in [I-D.brashear-afs3-pts-extended-names].

The above token structure is used for both database server tokens and file server tokens, however, there is a key distinction between the two: in file server tokens, the last identity in the list of identities is special, and corresponds to the "cache manager" identity that was presented to AFSCombineTokens in token1. File

server tokens must always have a cache manager identity present; if the token1 presented to AFSCombineTokens is empty, a placeholder identity is introduced into the identity list of the output token from AFSCombineTokens to indicate that fact. This placeholder identity is of a new kind, PRAUTHTYPE_EMPTY:

```
#define PRAUTHTYPE_EMPTY XXX
```

PrAuthNames of kind PRAUTHTYPE_EMPTY MUST have empty (zero-length) data and display components.

5. Authenticator Data

The appdata opaque within the RXGK_Authenticator structure contains the results of XDR encoding the RXGK_Authenticator_AFSAppData structure. The uuid field contains the UUID of the client.

```
struct RXGK_Authenticator_AFSAppData {
    afsUUID uuid;
};
```

6. Client Tokens

In order to protect users of a multi-user cache manager from each other, it must be impossible for an individual user to determine the key used to protect operations which affect the cache. This requires that the cache manager have key material of its own which can be combined with that of the user. This functionality is provided by the AFSCombineTokens call specified earlier in this document. However, this call requires that a cache manager have access to a token for this purpose.

6.1. Keyed Clients

When a host already has key material for a GSSAPI mechanism supported by rxgk, that material MAY be used to key the client. The client simply calls the rxgk negotiation service using the relevant material, and obtains a token. Combined tokens may not be used as cache manager tokens. The client SHOULD frequently regenerate this token, to avoid combined tokens having unnecessarily close expiration times. The client SHOULD NOT regenerate this token so often so as to place excessive load on the vlservers.

It is recommended that identities created specifically for use by a cache manager have the name afs3-callback@<hostname> where <hostname> is the fully qualified domain name of the cache manager.

6.2. Unkeyed Clients

When a client has no key material, it is possible that an anonymous GSSAPI connection may succeed. Clients MAY attempt to negotiate such a connection by calling GSS_Init_sec_context() with the anon_req_flag [RFC2743] and the default credentials set.

7. Server to Server Communication

A number of portions of the AFS protocol require that servers communicate amongst themselves. To secure this with rxgk we require both a mechanism of generating tokens for these servers to use, and a definition of which identities are permitted for authorisation purposes. We refer to the process of forging tokens for local use, given access to the relevant key, as "token printing".

7.1. Token Printing

A server with access to the cell-wide pre-shared key may print its own tokens for server-to-server access. To do so, it should construct a database server token with suitable values. The list of identities in such a token MUST be empty. It can then encrypt this token using the pre-shared key, and use it in the same way as a normal rxgk token. The receiving server can identify it is a printed token by the empty identity list.

The session key within a printed database server token MUST use the same encryption type as the pre-shared key. When connecting to a fileserver, a client SHOULD use the AFSCombineTokens service as discussed above to ensure that they are using the correct key for the fileserver.

File servers with per-server keys may also print tokens, though these tokens are in general of limited utility. (Being file server tokens, they are not valid inputs to AFSCombineTokens, etc..) In this case, the printed token should have a single identity in the list, of kind PRAUTHTYPE_EMPTY.

8. Declaring rxgk Support for a Fileserver

The AFSCombineTokens call has specific behaviour when a destination endpoint does not support rxgk. Implementing this behaviour requires that the vlserver be aware of whether a fileserver supports rxgk.

Fileservers currently register with the vlserver using the VL_RegisterAddrs RPC. This document introduces an extended version, VL_RegisterAddrsAndKey (Section 9), and either one may be used to indicate that a fileserver supports rxgk. Fileservers which support

rxgk MUST call these RPCs over an rxgk protected connection. The vlserver then infers rxgk support from the rx security layer used in registration. To prevent downgrade attacks, once a fileserver has registered as being rxgk capable, the vlserver MUST NOT remove that registration without administrator intervention. When the first rxgk-enabling call from a fileserver is VL_RegisterAddrs, that call MUST be made using a printed token, and the implication is that the token-encrypting key for that file server is the cell-wide shared key.

Once a fileserver has been marked as supporting rxgk, VL_RegisterAddrs calls for that fileserver MUST only be accepted over an rxgk protected connection. vlservers MUST only accept calls to VL_RegisterAddrs from a printed token, an administrator, or the identity registered for the fileserver using a prior call to VL_RegisterAddrsandKey.

9. Per Server Keys

The provisioning of file servers with their own keys, rather than the cell-wide master key, requires the ability to maintain a directory of these keys on the vlserver, so that the AFSCombineTokens RPC can encrypt the outgoing token with the correct key. The manner in which this directory is maintained is left to the implementor, who MAY decide to use a manual, or out of band, key management system. Otherwise, the automated keying mechanism described as follows will be used.

Implementations supporting automatic key management through the AFS-3 protocol MUST provide the VL_RegisterAddrsAndKey RPC (similar to the VL_RegisterAddrs RPC). This RPC is called by a fileserver to register itself with the VLDB; it MUST be called over a secure

connection. In particular, it MUST NOT be called over an rxkad connection.

For the purpose of this RPC, the fileserver acts as the client and the vlserver as the server. Once the RPC completes, both peers of the RPC call can generate a key to be used as the fileserver's long-term server key.

vlservers MUST NOT permit calls to VL_RegisterAddrsAndKey for UUIDs which already exist within the vldb, unless that UUID already has a server-specific key registered. The identity information stored in the token used to authenticate and secure the VL_RegisterAddrsAndKey connection is used to provide continuity between multiple calls to VL_RegisterAddrsAndKey, so a printed token is not appropriate here, as it would only be useful for the first call and there would be no identity available for cross-checking that a second call was from the

same source. To support this cross-check, when a new fileserver first registers with the vldb using VL_RegisterAddrsAndKey, the vlserver MUST store the identity list from the token used to make this connection. The vlserver MUST only permit subsequent calls to VL_RegisterAddrsAndKey for this UUID when they come over a connection authenticated with that same identity list, an administrator's token, or a printed token. New fileserver UUIDs register themselves with the vldb in a "leap of faith", binding a GSSAPI identity to the fileserver UUID for future authenticated operations. Fileservers SHOULD use VL_RegisterAddrsAndKey to rekey themselves periodically, in accordance with key lifetime best practices.

The VL_RegisterAddrsAndKey RPC is described by the following RPC-L:

```
struct RXGK_ServerKeyDataRequest {
    afs_int32 enctypes<>;
    opaque nonce1[20];
};

struct RXGK_ServerKeyDataResponse {
    afs_int32 enctype;
    afs_int32 kvno;
    opaque nonce2[20];
};
```

```
const RXGK_MAXKEYDATAREQUEST = 16384;
const RXGK_MAXKEYDATARESPONSE = 16384;
typedef opaque keyDataRequest<RXGK_MAXKEYDATAREQUEST>;
typedef opaque keyDataResponse<RXGK_MAXKEYDATARESPONSE>;
VL_RegisterAddrsAndKey(
    IN afsUUID *uuidp,
    IN afs_int32 spare1,
    IN bulkaddrs *ipaddr,
    IN afs_int32 secIndex,
    IN keyDataRequest *request,
    OUT keyDataResponse *response) = XXX;
```

uuidp: The fileserver's UUID.

spare1: Unused. (Clients SHOULD pass zero.)

ipaddr: The list of addresses to register as belonging to this fileserver.

secIndex: The index of the security mechanism for which a key is being set. For rxgk, this value MUST be 4.

keyDataRequest: An opaque blob of data, specific to the security mechanism defined by secIndex. For rxgk, it is the XDR-encoded representation of an RXGK_ServerKeyDataRequest structure.

keyDataResponse: An opaque blob of data, specific to the security mechanism defined by secIndex. For rxgk, it is the XDR-encoded representation of an RXGK_ServerDataResponse structure.

The client provides, in the RXGK_ServerKeyDataRequest structure, a list of the RFC3961 encryption types that it will accept as a server key. It also provides a nonce containing 20 random data bytes.

The server selects an encryption type shared by it and the client, and returns that, along with 20 bytes of random data that it has generated, in RXGK_ServerKeyDataResponse. If there is no common encryption type, then the server MUST fail the request.

The file server key can then be derived by both client and server

using

```
random-to-key(PRF+(K0, K, nonce1 || nonce2));
```

random-to-key is the function specified by the RFC3961 profile of the encryption type chosen by the server and returned in enctype.

PRF+ is the function of that name specified by [RFC4402].

K0 is the master key of the current rxgk session, e.g., as originally determined by the GSSNegotiate call.

K is the key generation seed length as specified in enctype's RFC3961 profile.

|| is the concatenation operation.

10. Securing the Callback Channel

AFS has traditionally had an unprotected callback channel. However, extended callbacks [I-D.benjamin-extendedcallbackinfo] require a mechanism for ensuring that callback breaks and, critically, data updates, are protected. This requires that there is a strong connection between the key material used initially to perform the RPC, and that which is used to protect any resulting callback. We achieve this using the cache manager token discussed in Section 6.1, which is required in order for a client to accept secure callbacks.

10.1. The SetCallBackKey operation

A cache manager may set a key for secure callbacks by issuing the following RPC (in the RXAFS service):

```
RXAFS_SetCallBackKey(afs_int32 securityIndex,
                     opaque mech_data<>) = XXX;
```

securityIndex: The security index of the mechanism for which this key is being set. The security index for rxgk is 4.

mech_data: This contains the security object specific data. In rxgk's case this is an XDR encoded RXGK_CallBackKeyData structure.

```
struct RXGK_CallBackKeyData {
    afs_int32 enctype;
    opaque K0<16384>;
    RXGK_Level level;
    /* no rxgkTime start_time */
    afs_int32 lifetime;
    afs_int32 bytelife;
    /* no rxgkTime expirationtime */
    RXGK_Data token;
    /* no identities needed */
}
```

enctype The encryption type of K0.

K0 The raw key data for the callback connection's connection master key.

level The security level to be used for the callback connection.

lifetime The maximum number of seconds that a key derived from K0 may be used for, before the connection is rekeyed. If 0, keys have no time-based limit.

bytelife: The maximum amount of data (expressed as the log base 2 of the number of bytes) that may be transferred using a key derived from K0 before the connection is rekeyed. If 0, there is no data-based limit on key usage.

token An opaque token that permits the client to identify the key and connection parameters used for the callback connection. This token behaves as a generic rxgk token, as described in section 5 of [I-D.wilkinson-afs3-rxgk], and its contents and encoding are implementation-defined. In particular, a client

implementation might be able to leave this token empty and store the key and connection parameters in an internal per-fileserver storage location. A client implementation might also reuse wholesale the token format defined in this document,

filling the token field with an XDR-encoded RXGK_TokenContainer containing an encrypted encoded RXGK_Token, with a per-client secret key.

No expiration or start time need be transferred in this RPC (as are included in an RXGK_Token and RXGK_ClientInfo), because the callback connection is implicitly authorized to continue for as long as the client is interested in data from the fileserver.

10.2. Lifetime and scope of the callback channel

The RXAFS_SetCallBackKey RPC must be performed over a secure channel. When used to set callback keys for rxgk, this means that the RPC MUST be performed over an rxgk protected connection of security level RXGK_LEVEL_CRYPT. Additionally, the connection MUST have been established using solely the cache manager's token.

The callback channel key is inherently tied to the identity of the cache manager token used to establish it. A fileserver which is providing secure callbacks MUST store the cache manager identity used to establish each callback connection key and associate that identity with the cache manager UUID. In the abstract, the fileserver is storing triples of (UUID, identity, key). A cache manager may make multiple calls to RXAFS_SetCallBackKey, and the fileserver MAY store multiple cache manager identity/callback connection key pairs for a given cache manager UUID. If a fileserver receives an RXAFS_SetCallBackKey call which will cause it to stop storing an identity/key pair (whether because the fileserver only stores one such pair for a given cache manager, or some larger fixed limit is reached), it MUST break all secure callbacks held by that client that are using the old key before the RPC terminates.

Only RPCs issued over an rxgk protected connection should receive rxgk protected callbacks.

Since the callback connection key is tied to the cache manager identity, it should only be used to protect callbacks relating to data accessed using that identity. Since extended callbacks are only available when a cache manager token is in use, and the cache manager token's identity is always the last identity in the list, this means that the fileserver MUST only send rxgk protected extended callbacks when the cache manager identity in the token authenticating the RPC establishing the callback matches an identity associated with that cache manager's UUID. Additionally, the fileserver MUST use the

callback key (and token) registered to that identity for the extended callbacks established as a result of the RPC.

11. IANA Considerations

This memo includes no request to IANA.

12. AFS-3 Registry Considerations

This document requrests that the AFS-3 registry allocate code points for the new RPCs AFSCombineTokens (for the RXGK service), RegisterAddrsAndKey (for the VL service), and SetCallBackKey (for the RXAFS service). It also requests the allocation of a value for the PRAUTHTYPE_EMPTY kind of PrAuthName.

13. Security Considerations

13.1. Downgrade attacks

Using the presence of a GSSAPI key to determine a cell's ability to perform rxgk is vulnerable to a downgrade attack, as an attacker may forge error responses. Cells which no longer support rxkad should remove their afs@REALM and afs/cell@REALM Kerberos keys.

13.2. Per Server Keys

The mechanism for automatically registering per-server keys is potentially vulnerable, as it trades a short-lived key (the rxgk session key, which protects the key exchange) for a long-lived one (the server key). There is precedent for this sort of key exchange, such as when using kadmin to extract a new kerberos keytab.

13.3. Combined Key Materials

As described in Section 6, combined tokens are used to prevent cache poisoning attacks on multi-user systems. In order for this protection to be effective, cache managers MUST NOT provide user access to keys produced through the combine tokens operation, unless those keys will not be used by the cache manger itself.

14. References

14.1. Informational References

[RX] Zeldovich, N., "RX protocol specification", October 2002.

[I-D.benjamin-extendedcallbackinfo]
Benjamin, M., "AFS Callback Extensions (Draft 14)", draft-benjamin-extendedcallbackinfo-02 (work in progress), December 2011.

14.2. Normative References

[I-D.brashear-afs3-pts-extended-names]
Brashear, D., "Authentication Name Mapping extension for AFS-3 Protection Service", draft-brashear-afs3-pts-extended-names-09 (work in progress), March 2011.

[I-D.wilkinson-afs3-rxgk]
Wilkinson, S., "rxgk: GSSAPI based security class for RX", draft-wilkinson-afs3-rxgk-00 (work in progress), January 2010.

[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.

[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.

Appendix A. Acknowledgements

rxgk has been the work of many contributors over the years. A partial list is contained in the [I-D.wilkinson-afs3-rxgk]. All errors and omissions are, however, mine.

Appendix B. Changes

B.1. Since 00

Add references to RX and XDR specifications.

Add introductory material on AFS.

Change expirationTime to be expressed using the rxgkTime type.

Document how encryption types are chosen for printed tokens, and how they are used against fileservers.

Expand security considerations section to cover combined tokens.

Rename AFS_SetCallbackKey as RXAFS_SetCallbackKey.

B.2. Since 01

Rename RXAFS_SetCallbackKey to RXAFS_SetCallBackKey.

Add an AFS-3 Registry Considerations section.

Clarify the vlserver/dbserver/fileserver relationship.

AFSCombineTokens prototype changes.

Clarify the scope of the document.

Use a leap of faith for RegisterAddrsAndKey.

Specify the nametype of the acceptor identity.

B.3. Since 02

Deal with fallout of errorcode's removal from RXGK_TokenInfo.

Rework "securing the callback channel".

B.4. Since 03

Clarify the distinction between dbserver and fileserver tokens.

AFSCombineTokens is the only way to get file server tokens.

Add new kind of PrAuthName, PRAUTHTYPE_EMPTY.

Specify how cache manager token identities are stored in file server tokens.

Place bounds on some XDR opaque arrays.

Expound more about printed tokens, for dbservers and fileservers.

Authors' Addresses

Simon Wilkinson
Your File System Inc

Email: simon@sxw.org.uk

Benjamin Kaduk
MIT Kerberos Consortium

Email: kaduk@mit.edu