RXK5 kerberos 5 authentication for rx

Marcus Watts

September 18, 2009

Abstract

Rxk5 is a replacement for rxkad that does kerberos 5 natively, supports strong encryption, and is designed to be as simple and obvious as possible. This document describes the design philosophy and the basic protocol. Integration into OpenAFS is also described.

1 Why Do Rxk5

Rx[1] has been around since 1990. Rx proper contains a flexible scheme whereby multiple different authentication modules can be plugged in. The original set of modules included rxnull, which handled unauthenticated access, and rxkad, which handled kerberos 4 style tickets. These modules remain the only supported choice today, even though kerberos 4 is extremely dated. Some security improvements were made in rxkad, but the level of security provided by rxkad is still less than would be desirable for many purposes.

Rxk5 was originally implemented in 2005. The initial focus were certain administrative services at the University of Michigan. These services were based on kerberos 4 and rxkad; the goal was to replace that with high quality kerberos 5 support. A specific initial goal was to eliminate the dependency on des, and to support all the encryption types supported in kerberos 5. Some specific improvements were then made to the protocol at the behest of the afs community. The current version has been integrated with OpenAFS.

2 Basic Requirements

During the design of rxk5, the following design goals were identified.

drop-in replacement Using rxk5 should not require changes to kerberos or OpenAFS.

limit session key use The session key should not be used to encrypt data packets. Keys used to encrypt data packets should have a lifetime that is less than that of the kerberos ticket. Knowledge of the key used to encrypt one data packet should not compromise the session key, data packets encrypted with other keys on this rx connection, or other connections.

strong protection against replays It should not be possible to use previously sent packets to fool the server into redoing work.

enctype and level negotiation There should be a scheme whereby the server and client select a mutually acceptable enctype and integrity checking level.

level=none There should be support in the protocol for identification only use, that is, no integrity checking or privacy.[1]

kerberos library The security mechanism should allow standalone operation, that is, not require an external kerberos library. This requirement is obsolete today.

enctype negotiation The security mechanism must support some means of determining which enctypes are available in the environment.

3 Rxkad Defects

Rxkad has numerous faults, most of which are well known. Some obvious ones are the use of kerberos 4 tickets, and the use of fcrypt[7] which is far weaker than even des.

Integrity checking in rxkad has been substantially improved[6], but is still substandard. Kerberos 4 has similar problems, but can provide limited assurance of integrity since the server can verify that the decrypted data “makes sense”. With rxkad, this isn’t possible. Later versions of rxkad borrowed a header field to add checksum data. This improves what was present, but is still not up to modern standards. Modern cryptographic practice also includes the notion of a “confounder” prefixed to the encrypted data. This notion is completely lacking in kerberos 4 and in rxkad.

Rxkad has limited support for kerberos 5 tickets[8]. While it is technically possible to use service keys other than des, this requires a custom ticket decoder and is not commonly implemented. The session key given to fcrypt must still be des.

Rxkad uses the same session key without modification to encrypt and decrypt all data, on all connections. Modern cryptographic practice is to avoid extended use of session keys, to use different keys for different connections, and to use derived keys for different purposes to ensure that different key uses cannot be used against each other.

4 Rx Basics

Rx runs on top of udp. It implements logical connections on top of the udp protocol. Each connection is made to a host, port, and serviceid on a remote machine. Connections are uniquely identified by epoch || cid, which are assigned when the connection is made. Connections have 4 channels; each channel can run up to one call logically independent of the other channels. Data is sent in packet sized chunks. This matters for flow control and encryption. Packet boundaries are (usually) invisible to client programs, which assume they have a byte stream. A call starts by sending data from the client to the server. When the client is done, it ends the data, and waits for a response from the server. The server starts processing a call as soon as it has any data (and other resources). Once the server side logic has read all the data it needs, it can then return data to the user. The server may instead return an error packet, in which case the call is ended.

4.1 rx header

Figure 1 on the facing page is a picture of the rx packet header. The epoch is assigned once by the client and used for

[1]This is only useful for environments where security is not a big concern. For instance, a client/server environment in a restricted network where the use of kerberos for identification is a convenience, not a security need.

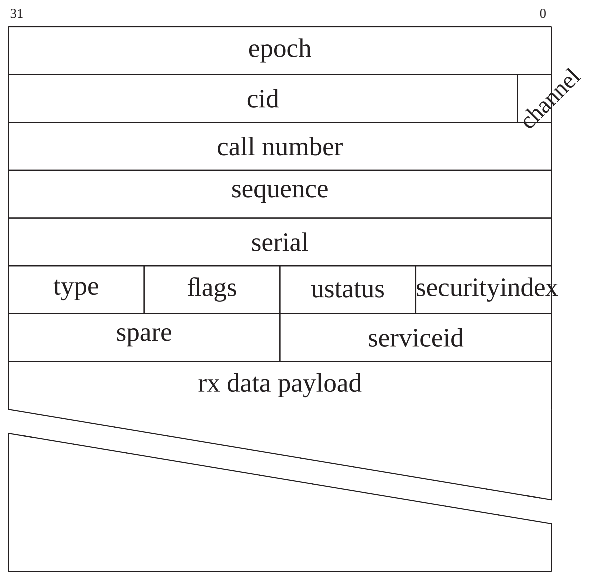

Figure 1: rx header
All values are big-endian.

all connections. The top bits of the epoch have arcane meaning to rx. The cid is assigned uniquely per connection. It is a 30-bit value. The channel is a 2-bit number, indicating which of 4 channels to use. The the server keeps the largest call number seen per connection channel, and require that call numbers on each channel be monotonically increasing.

The sequence number allows the other end to reconstruct the byte stream from packets that might have arrived out of order. The serial number is the value of a counter that is kept per connection; it is incremented each time a packet is sent or resent. Ustatus or user status can be used to “color” packet traffic on a connection. Ustatus is set “late”, just before the packet is sent or resent. No known AFS application uses this. The serviceid maps into an rx grammar and set of operations. It’s assigned by applications on a per grammar basis. Some services, such as ubik, make their own set of operations known on the same port that the application is using; since they have a different serviceid, there is no conflict.

The security index indicates which security class to use. There is no requirement that any particular index be used for a security class, and different applications could, in theory, use different mappings. Nevertheless, by convention, there is a fairly strict mapping, as follows:

index	**class**	**use**
0	rxnull	null security class
1	rxvab(bcrypt)	obselete, was used in AFS 3.0
2	rxkad(kerberos4,des)	current practice
3[2]	rxgss	(experimental, 2002)
4	rxgk	(experimental, 2003)
5	rxk5	(as described here)

[2]Some afs documentation suggests that the fileserver implements an encrypting-only (`rxkad_crypt`)) flavor of rxkad as security index

The “spare” field is in fact used by rxkad, to store a 16-bit checksum value. In rxk5, the checksum is larger, and must be stored elsewhere, so this field is not used.

4.2 packet flow

Figure 2 on the next page is a picture of the basic security packet flow sequence, as used by rx.

1. When the server starts up, for each security class it intends to support, it calls newserversecurityobject. These are then passed to `rx_NewService`
2. Before the client makes a secure connection, it gets a ticket and then calls the selected class’s newclientsecurityobject.
3. When the client makes a secure connection, it is passed in the client security object. Multiple connections to different servers may share the same security object, provided the servers share the same service key.
4. When the client makes a call, it will eventually cause queued rx write data to be flushed. This results creating one or more packets to send. Each data packet is passed to `RXS_PreparePacket` exactly once before being sent.
5. Just before each data packet is sent or resent, `RXS_SendPacket` is called.
6. When the server receives a packet on a connection it does not know about, it first calls newconnection, which allocates per-connection security object data.
7. The server then calls `RXS_CheckAuthentication`. If the security object indicates the connect is not yet authenticated[3], the server then saves the incoming data packet.
8. The server calls `RXS_CreateChallenge` once to allocate any fixed data for the challenge.
9. The server calls `RXS_GetChallenge` one or more times to create a challenge packet. If the client fails to return a response within a given time, the server will call `RXS_GetChallenge` a fixed number of additional times to request a response.
10. When the client receives a challenge, it calls `RXS_GetResponse` to create a response. The response should prove in some secure fashion that the client saw the challenge, and must contain sufficient information for the server to handle the data packet that was previously sent before the challenge was created.
11. When the server receives a response, it calls `RXS_CheckResponse`.
12. If checkresponse returned success, the server then calls `RXS_CheckPacket` to handle the saved data packet.
13. When the server has data to send back to the client, it writes data to be sent back to the client. `RXS_PreparePacket` is called exactly once on each data packet.

3. While the fileserver does in fact set up rxkad this way, it’s not useful: when determing authorization rights, security class 3 is always given anonymous access. There is almost certainly no client code using this.

[3]The official documentation[1] and the source make misleading statements about this. The documentation claims that `RXS_CheckAuthentication` should return 1 if authenticated calls are being made on this connection. The source claims that rx may invoke this every RX_AUTH_REQUEST_TIMEOUT seconds on active connections. Neither of these statements is true. There is no such periodic polling mechanism in rx. Instead, `RXS_CheckAuthentication` is invoked at the start of every server call. It should return non-zero whenever the security mechanism wants rx to issue a challenge on a connection. If there are other calls that have started execution on the same connection, they will proceed unimpacted. If `RXS_CheckAuthentication` returns non-zero, it must not then return 0 until `RXS_CheckResponse` is called with an acceptable response.

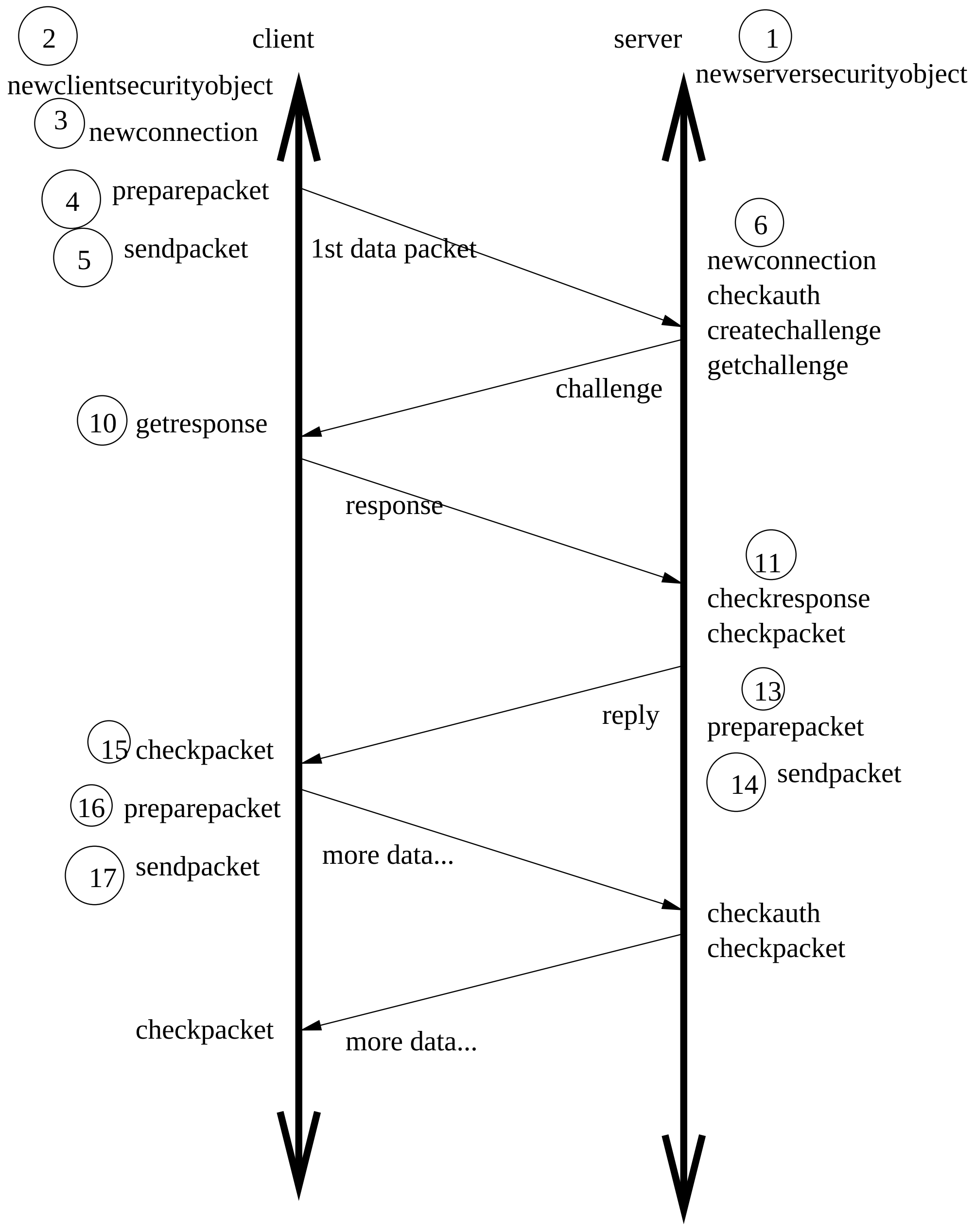

Figure 2: rx security packet sequence

14. Just before each data packet is sent or resent, `RXS_SendPacket` is called.

15. The client calls `RXS_CheckPacket` on each packet returned by the server.

16. The server and packet may exchange additional packets. The client and server call `RXS_PreparePacket` and checkpacket on each packet.

Note that the client must choose a key, encrypt a data packet, and send it to the server before it receives a challenge. Therefore, it can't change the key used for encrypting a particular packet based on receiving a challenge for that packet. The client could choose to encrypt future data packets differently. This requires some form of old key/new key logic to handle any in-flight packets.

The client has no way to cause the server to re-challenge the client, except to start a new connection. Starting a new connection requires application visible changes, significant reworking of rx, and/or hairy logic in the security object.

The server conventionally does not re-challenge a client once it has received a valid response. It might re-challenge the client if the connection is left idle long enough to be reclaimed on the server, or if the server crashes and has restarted. It appears to be possible for the security object on the server to cause a challenge to be issued on an active connection by returning false to `RXS_CheckAuthentication`. A connection in this state should refuse to service any further new calls, but should complete any calls already in progress. There is no existing security class that does this, so this logic might not perform as expected.

4.3 security operations

rx security operations:

	client	**server**
close	y	y
newconn	3	6
preparepkt	4	13
sendpacket	5	14
checkauth	.	7
cr_challenge	.	8
getchallenge	.	9
getresponse	10	.
checkresponse	.	11
checkpacket	15	12
destroycon	y	y
getstats	y	y

The numbers here can be sorted into call order, and correspond with the numbers in section 4.2 on page 4

pairs:	sender	receiver
	preparepacket	checkpacket
	cr_challenge+get_challenge	getresponse
	getresponse	checkresponse
	newconn	destroyconn

The security object may set "maxHeaderSize" and "maxTrailerSize". Data copied in and out of data buffers starts at offset "maxHeaderSize". preparepacket may increase packet size up to "maxTrailerSize". checkpacket may reduce packet size up to "maxTrailerSize". Rx data sizes are nearly always multiples of 4, because of the use of xdr[14].

-connection and security object-
newsecurityobject - create security object.

newconn - initialize for new connection.
destroyconn - free per-connection data.
close - discard security object data (session key).

-challenge/response-
checkauth
create_challenge
get_challenge
get_response
check_response
getstats

-per packet-
preparepkt
sendpacket
checkpacket

5 Rxk5

For rxk5, the client application is responsible for supplying the session key and ticket. On the server side, the default logic expects to decode a kerberos 5 ticket. Normally the service ticket will be gotten directly from kerberos 5 in the usual way. The session key is only used during the challenge response process (see section 4.2 on page 4) to generate the keys that will be used for the rest of the connection (see section 5.2 on page 9).

The actual acceptable ticket format is determined entirely by the ticket decoding logic in the server. The client side ticket logic ships whatever ticket is given it. Since both these can be customized by the application, specific applications could choose to either augment or substitute the ticket format.

There are 2 available hooks to protect outgoing data packets. They are preparepacket, and sendpacket. Preparepacket is called first, exactly once. Sendpacket may be called multiple times, once before each time the packet is sent. Rxk5, like rxkad, uses preparepacket. This has the consequence that several fields cannot be protected by the security class. These fields include: serial number, flags, ustatus, and spare.

5.1 security operations

Rxk5 implements the following security operations:

	client	**server**
close	`rxk5_c_Close`	`rxk5_s_Close`
newconn	`rxk5_c_NewConnection`	`rxk5_s_NewConnection`
preparepkt	`rxk5_c_PreparePacket`	`rxk5_s_PreparePacket`
sendpacket	`0`[4]	`0`[4]
checkauth	`0`	`rxk5_s_CheckAuthentication`
cr_challenge	`0`	`rxk5_s_CreateChallenge`
getchallenge	`0`	`rxk5_s_GetChallenge`
getresponse	`rxk5_c_GetResponse`	`0`
checkresponse	`0`	`rxk5_s_CheckResponse`
checkpacket	`rxk5_c_CheckPacket`	`rxk5_s_CheckPacket`
destroycon	`rxk5_c_DestroyConnection`	`rxk5_s_DestroyConnection`
getstats	`rxk5_c_GetStats`	`rxk5_s_GetStats`

Rxk5 security operations may be classified into several groups; in-memory connection operations, challenge/response, packet operations, and application visible entry points.

Connection and security objects. These only affect local memory; they do not cause any network on their own.

`rxk5_NewServerSecurityObject` (server) This is called by the application. It retains information about how to fetch the server's key. Multiple connections may be accepted using this security object.

`rxk5_NewClientSecurityObject` (client) This is called by the application. It retains the client credential for use with one or more connections.

`rxk5_s_NewConnection` (server) allocate per-connection storage.

`rxk5_c_NewConnection` (client) allocate per-connection storage, select checksum type.

`rxk5_c_DestroyConnection` (client) discard all connection keys, free storage.

`rxk5_s_DestroyConnection` discard all connection keys, free storage. also free name information from ticket.

`rxk5_s_Close` (server) When reference count drops to zero, free local storage.

`rxk5_c_Close` (client) When reference count drops to zero, free local storage. on client: free session key.

challenge/response These cause or respond to network I/O.

`rxk5_s_CheckAuthentication` (server) return non-zero if connection needs to have a challenge issued, zero if the connection is available for use.

`rxk5_s_CreateChallenge` (server) allocate per-connection local storage, allocate challengeid (random integer), retain level.

`rxk5_s_GetChallenge` (server) serialize vers+challengeid+level: construct challenge packet. See figure 5 on page 11 for layout of the challenge.

`rxk5_c_GetResponse` (client) Eat challenge, make response. Keys were previously selected when the 1st data packet was sent; so the job of this function is to construct a response that contains the ticket & the master key.

`rxk5_s_CheckResponse` (server) Unpack response. Decrypt ticket with server key. Decrypt authenticator with session key. Verify authenticator. Save client name for later authorization purposes. Based on the master key, compute and save client and server side keys. Validate encryption level, key types. For `rxk5_auth`, save the client's checksum type.

[4] sendpacket is called for both sending and receiving, but rxk5 does not use it.

`rxk5_s_GetStats` (server) return use counts on demand.

`rxk5_c_GetStats` (client) return use counts on demand.

Packet operations

`rxk5_c_PreparePacket` (client) generate keys. call prepareencrypt or preparesum.

`rxk5_s_PreparePacket` (server) call `rxk5i_PrepareEncrypt` or `rxk5i_PrepareSum`.

(sendpacket) Not used in rxk5. Note that this is invoked on every retransmit.

`rxk5_c_CheckPacket` (client) call `rxk5i_CheckEncrypt` or `rxk5i_CheckSum`.

`rxk5_s_CheckPacket` (client) call `rxk5i_CheckEncrypt` or `rxk5i_CheckSum`.

These aren't directly defined by the RXS framework, but are necessary in order to use it.

`rxk5_NewServerSecurityObject` Create a new server side security object. Holds key connection logic, primitive authorization, and kerberos context.

`rxk5_NewClientSecurityObject` Create a new client side security object. Holds ticket.

`rxk5_GetServerInfo` Return information about the authorization associated with this name.

`rxk5_GetServerInfo2` Return information about the authorization associated with this name. Also returns the server principal.

`rxk5_get_context` Return the k5 context being used by rxk5.

`rxk5_default_get_key` Given a context, server principal, return the key.

The function `rxk5_AllocCID` is responsible for allocating connection ids. On the first call, it sets up a local epoch and counter, and will also call `rx_SetEpoch`. On successive calls after that, it will return the "next" cid (counter$+ =$4). The high order bits of the epoch are forced to be 10. This logic precisely follows that in rxkad, except that the random number logic is different for rxk5.

5.2 Key Flow

Figure 3 on the following page is a picture of the basic key flow. The session key is only used to encrypt the challenge. The challenge contains a master key, randomly generated for each connection, which is used to derive all other keys used for that connection. The master key is then used to derive 2 more keys, one used to protect all client side packets, and one used to protect all server side packets. On the server, the connection's master key is of no further interest and is not kept once the response is processed. On the client side, the master key must be kept in case an additional challenge is issued. The server and client side keys are not used directly, but are further permuted to become per-packet keys, so every packet is encrypted by a unique key which is never reused.

5.3 Rxk5 Key Derivation

The common algorithm used to derive most keys in rxk5 is:

$$K_{b,d}(K_b, D) = N(H(K_b, D))$$

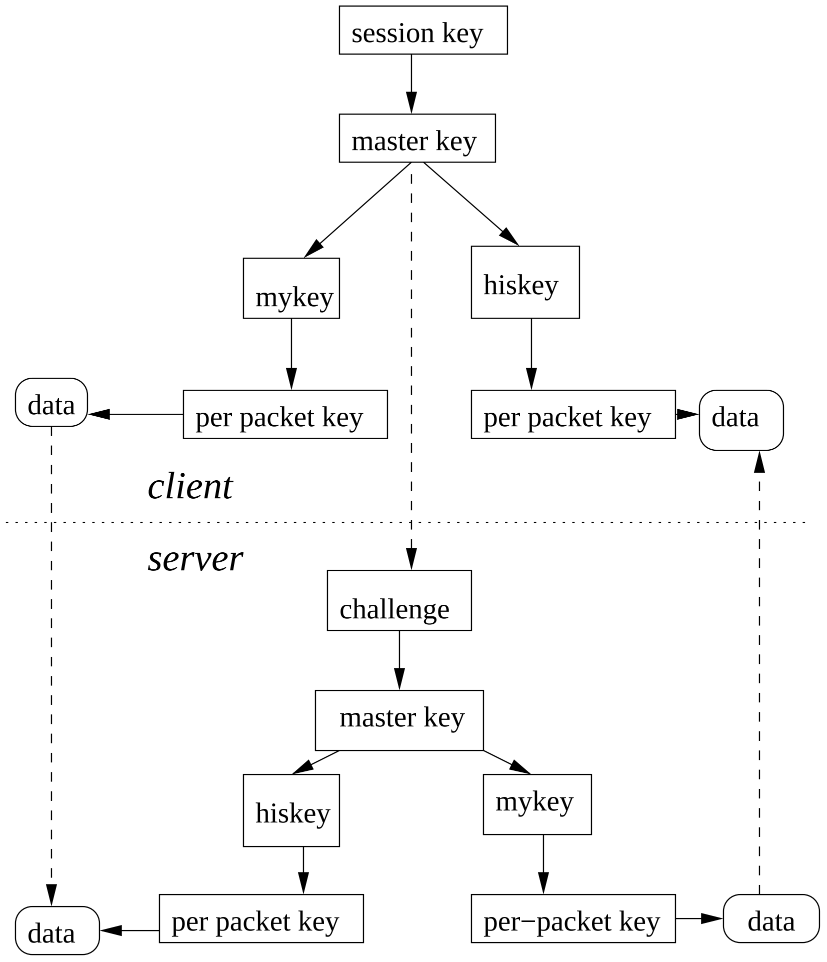

Figure 3: rxk5 key flow

where K_b is the base key, D is an octet array, H is md5[9], and N is the kerberos n-fold operation[11]. DES keys need to have the parity bits fixed up in the usual way. This is not a clever algorithm, it was merely the best available that was reasonably portable. It was intended that this be replaced with the Kerberos 5 PRF function described in [11]. Unfortunately, it turns out the PRF specification was incomplete, so useful implementations took much longer than expected.

The data D used to derived the client key is the integer 102, stored in 4 octets in network byte order. The data used to derive the server key is the integer 103, again as 4 bytes in network byte order. The data used to derive the per-packet keys is the pseudo-header.

5.4 Rxk5 data structures

Figure 4 on the next page is a picture of the pseudo-header. The pseudo-header is important, because it's what keeps

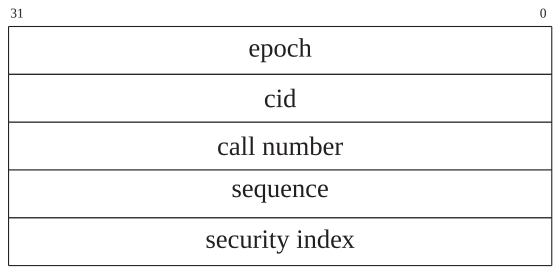

Figure 4: pseudo-header
All values are big-endian.

packets from being replayed later in the connection. In rxkad, the pseudo-header was prepended to the data, but was not actually transferred on the connection. Use of the pseudo-header in this fashion is not obviously compatible with the kerberos cryptographic suite[11]. The encrypt and decrypt routines there do not allow for "virtual data" – data that should be part of the checksum algorithm, but that should not be transported.[5] In rxk5, the pseudo-header is instead used to generate the per-packet key. Since each packet gets a unique pseudo-header, each packet also gets a unique key. In a pure software implementation, using a unique key per packet is not a significant performance obstacle. Hardware acceleration was not a consideration for rxk5.[6]

Figure 5 is a picture of the challenge. This is sent by the server to the client whenever it sees a new connection, and starts the process of transfering connection key information to the server. Here, vers is 0. The client will reject any

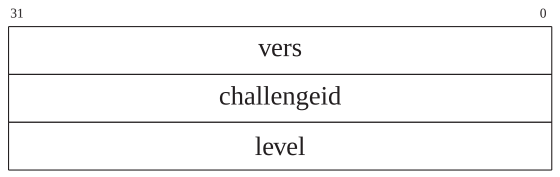

Figure 5: challenge
All values are big-endian.

value but 0. The challengeid is a random integer selected by the server. Level here is the min level that the server will accept. Normally that should be `rxk5_auth` (1). The client will return an error if the level it used is less.

Figure 6 on the next page is a picture of the response. This is sent by the client to the server whenever the client receives a challenge packet. It contains connection key information for the server. Here, vers is 0. The server will reject any value but 0. The field skeyidx is used when using the session key to ship data. The goal is to vary the usage given to the session key on successive uses. The field skeyidx is set to the connection serial counter, as a convenient strictly increasing small integer source. When calling the kerberos encryption functions, there is a "usage" parameter. This is set to `USAGE_RESPONSE`+*skeyidx*. Here, `USAGE_RESPONSE` is the value 104.

Figure 7 on page 13 is a picture of the encrypted part of the response. It's sent by the client to the server, and is

[5] One could argue the ivec does this, sort of.

[6] At the time of implementation, hardware acceleration was not common. Hardware acceleration that is key agile and supports some form of scatter-gather I/O or incremental encryption should nevertheless work reasonably.

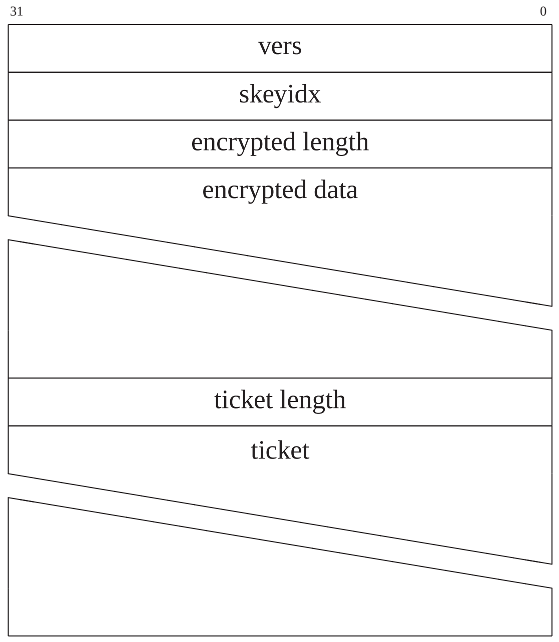

Figure 6: response

All values are big-endian. The encrypted data and ticket are padded on the right to the next word boundary with up to 3 NULs.

encrypted under a derived copy of the session key. It contains the actual master key generated by the client, plus additional information needed by the server to setup or recreate the server side connection state.

The rxk5 response is very similar to an rxkad response, The encrypted part is similar to an authenticator in kerberos 5, except additional data necessary to properly secure rx is included. This data closely follows that introduced with a security enhancement to rxkad, which was first introduced in AFS 3.1.[6] A few extra fields necessary to rxk5 are included, and xdr[14]is used to simplify data marshalling.

epoch and cuid The epoch and cuid uniquely identify one connection to rx.

securityindex This must be the same value as the security index in rx.

challengeid This must be the challenge id issued by the server incremented by one.

level This is the encryption level requested by the client application. It will be one of 3 values, `rxk5_clear (0)`, `rxk5_auth (1)`, or `rxk5_crypt (2)`. See section 5.5 on the next page.

cktype For level `rxk5_auth (1)`, this is the kerberos 5 checksum algorithm that the client selected. See section 5.5 on the facing page.

dktype The value `DK_CUSTOM (0)` indicates that a key derivation formula involving md5 is used. All other values are reserved. See section 5.3 on page 9.

callnumber This is a vector containing the number of calls the client has processed on each channel. The server initializes its state with these call numbers, and will not accept smaller call numbers. This is an important part

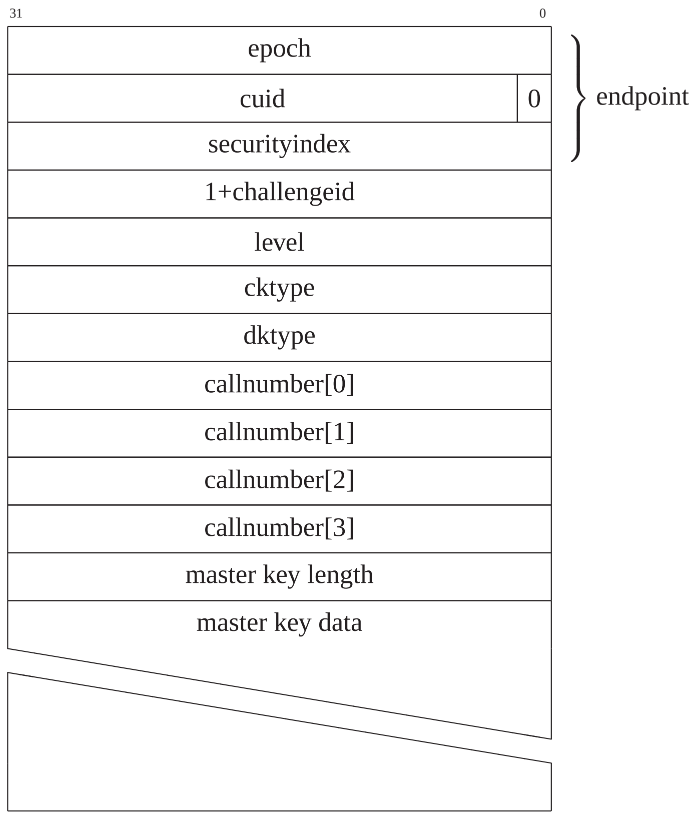

Figure 7: authenticator

All values are big-endian. The key is padded on the right to the next word boundary with up to 3 NULs.

of the replay protection.

master key This is similar to a kerberos subkey. All other keys used by rxk5 for this connection are derived from this key.

5.5 encryption levels

Encryption levels are entirely a property of the security class. As such they are set when the security object is created, and are not directly visible in the base rx connection in memory, or in the packet header during transit.

Rxk5 supports 3 encryption levels:

`rxk5_clear`	`0`	Identification only
`rxk5_auth`	`1`	Identification and integrity protection
`rxk5_crypt`	`2`	Identification, integrity checking, and privacy

Although rxk5_clear is defined, it has little advantage over rxk5_auth, and considerably less security. Therefore, rxk5_auth is recommended in all cases, and rxk5 must be compiled specially to even make rxk5_clear available.

For rxk5_auth, a keyed checksum algorithm is used. It turns out that encryption types and checksum algorithms do not have a simple 1-1 relation. There can be multiple checksum algorithms for any given key type, and the exact combinations available depend on the implementation. Therefore, rxk5 servers should be prepared to accept all checksum algorithms compatible with a given key. Clients should choose the "best" algorithm available.

Originally, rxk5 used a kerberos library function (MIT kerberos: `krb5_c_keyed_checksum_types`) to select a compatible checksum type, but there was no such equivalent function in other kerberos implementations. Worse yet, there was enough variation in encryption and checksum type availability that it was surprisingly difficult to make the right choice in all cases. Current logic uses a fixed switch table that should yield the same results, as follows

enctype		**cksumtype**		
DES_CBC_CRC	1	RSA_MD5_DES	8	des
DES_CBC_MD4	2	RSA_MD5_DES	8	
DES_CBC_MD5	3	RSA_MD5_DES	8	
ARCFOUR_HMAC_MD5	23	HMAC_SHA1_DES3	12	rc4
DES3_CBC_SHA1	16	HMAC_SHA1_DES3	12	triple des
AES128_CTS_HMAC_SHA1_96	17	HMAC_SHA1_DES3	12	aes 128
AES256_CTS_HMAC_SHA1_96	18	HMAC_SHA1_DES3	12	aes 256
DES3_CBC_MD5	5	HMAC_SHA1_DES3	12	(heimdal only)
DES_HMAC_SHA1	8	HMAC_SHA1_DES3	12	(mit only)

Note that cksum type `HMAC_SHA1_DES3 (12)` is in fact a plain hmac; it does not use triple des and will work with any keytype.

The client always computes and sets cktype in the response to the challenge. The server ignores cktype unless `rxk5_auth` is selected, in which case it validates the checksum type.

5.6 packet header and trailer sizes

Adding integrity or encryption increases the size of the data sent. Since rx packets are fixed size and filled in before the security layer sees it, it's important to tell rx how much space to reserve. Rx provides 2 calls for this, `rx_SetSecurityHeaderSize` and `rx_SetSecurityMaxTrailerSize`. For `rxk5_auth`, the overhead is quite simple: it's the checksum length. The checksum is inserted at the start of the packet, before the data, therefore the header length is also set to the checksum length.

For `rxk5_crypt`, calculating overhead is quite a bit more complicated. The various kerberos 5 encryption types add a confounder, checksum, and perhaps padding to the input data. The confounder and checksum do not necessarily end on a block boundary, so block jumps in padding needs are not necessarily block aligned. If padding is required, then the decrypted length will include the extra padding that kerberos added. If it turns out kerberos is going to add this padding, then rxk5 needs to supply intentional padding so that it can be safely removed at decrypt time. Therefore, computing overhead is messy: a series of trial encryption sizes must be tried to determine the confounder and checksum overhead, and padding requirements. The header and trailer sizes need to be set to the smallest and largest encrypted size increases seen. If padding was detected, then the input block offset must also be saved.

When encrypting and decrypting packets, if padding is needed, then the block size and block offset must be used. The block size can be computed as the difference between the header and trailer sizes + 1. The block offset that was previously saved must be used when computing the padding difference. Note that the header size set in rx has no actual connection to how the kerberos encryption functions might arrange the confounder, checksum, and data.

5.7 ticket size considerations

In the current implementation of rx, the challenge and response must each fit inside one packet. This is normally about 1400 bytes, but the exact limit depends on jumbogram support, also system and network constraints, the actual limit may be more like 16000 bytes. The maximum acceptable kerberos ticket is going to be slightly less than this size due to the need to include the encrypted authenticator.

In theory, this limit should be plenty generous. While the standard allows the construct very large kerberos tickets, in practice, the main items in a service ticket for rxk5 that are of variable size are the client and service principals, which should not be very large. The only item that is likely to cause problems is the authorization_data contained in the encrypted part of the ticket. There's no mechanism within OpenAFS or rxk5 today to make use of this data, however, there are some implementations of kerberos[7] that may put data here even though it is not useful[8]. This problem is not unique to rxk5.

Now that rxk5 is integrated with openafs, making small changes to rx is more feasible. A future version will probably include large ticket support, by extending the response logic in rx.

6 Afs Integration Notes

Implementing rxk5 in OpenAFS is straight-forward. On the client side, the main issue is managing multiple token types, the cache manager, and some kerberos integration issues. On the server side, the main issue is handling localauth. Most of the pieces closely reflect existing rxkad practice. One significant change: `rxk5_auth` is used wherever `rxkad_clear` was formerly used. Figure 8 on the following page is a picture of the major pieces of OpenAFS.

On the client side, the decision about whether to use rxk5 is made at "klog" time. At this time, the kerberos ticket granting ticket is used to acquire a kerberos 5 ticket for the cell. As such, it represents the easiest place to make decisions about what security mechanism to use. The existing logic in klog and aklog has an ad hoc algorithm to decide which service principal to use for rxkad. This is extended with rxk5 to also try for a rxk5 service ticket. If an rxk5 service ticket can be acquired, then the cell must support rxk5. Additional command line switches are provided for klog/aklog to disable the use of rxk5 or rxkad. Before klog and aklog request an rxk5 service ticket, they check the local cache manager for the set of encryption types supported in the kernel, and this is passed to the kdc. The kdc will select the "best" encryption type and return a session ticket of the appropriate type. No additional security negotiation is required to use rxk5 with cell services after that.

The cache manager already contained logic to select the use of rxkad or rxnull connections. This logic is enhanced to also support the use of rxk5 credentials. The in-kernel credentials logic is somewhat confused, and presents the illusion that it matters what viceid is assigned to the the client credential. This is in fact false; there's no requirement

[7] Microsoft Active Directory

[8] See Microsoft Knowledge Base article 832572 for how to disable PAC authorization data on a principal in Active Directory.

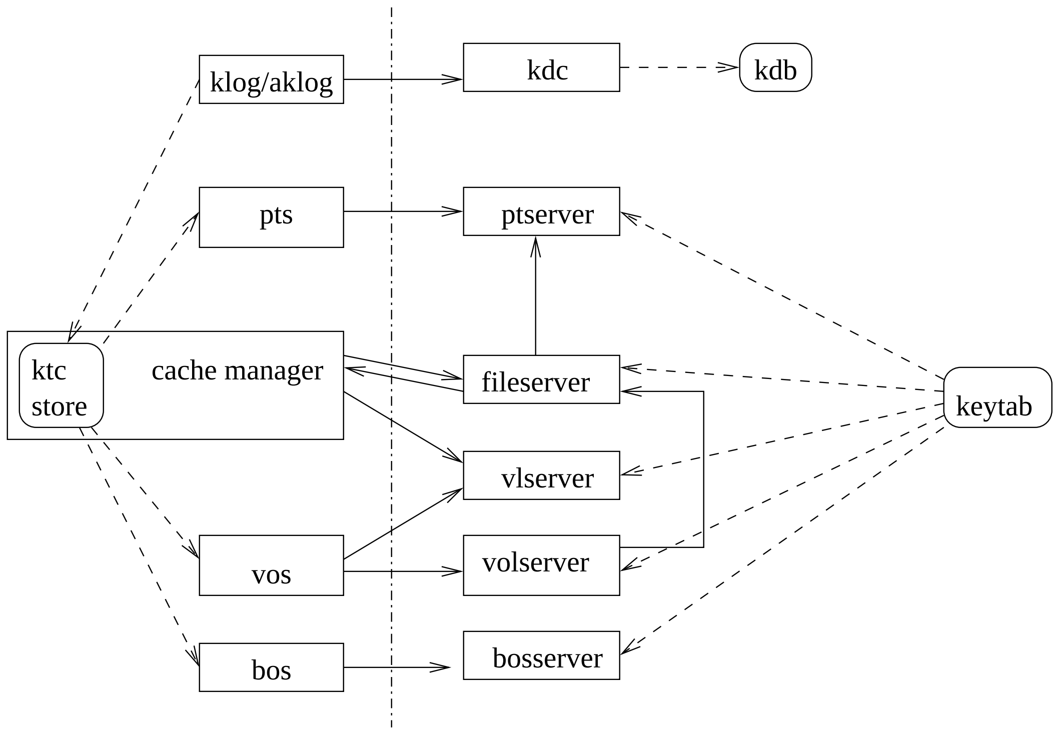

Figure 8:
afs the big picture

in the cache manager that the client know what viceid was assigned to the user. There is a reserved value of viceid (`UNDEFVID`, -1) that indicates that there is no authentication information for this user in this cell. For rxk5, an arbitrary viceid is assigned. The existing rxkad credential pioctl interface is not suitable for extension, so a new extensible mechanism was devised to pass credential types between the kernel and user code. The userland utilities klog/klog/tokens/unlog were extended to use the new interface.

The new interface provides the capability to pass in a list of credential types, and for a userland application to selectively skip credential types it doesn't understand. Because rxk5 and rxkad don't support the simultaneous use of multiple security mechanisms within one cell, there isn't any logic in the kernel that will do anything useful with a list of more than one credential type, therefore, these are simply rejected. A compatibility hack which falls back to using the old pioctl interface allows the use of new code on old cache managers.

Here is an example of how the kernel credentials interface could be extended in the future: currently the effects of "fs setcrypt" are system-wide. To make this per-authentication, an additional credential type could be supplied which says "do fs setcrypt for this user only". The credentials list is available to the connection logic, which can decide to implement the effects of "fs setcrypt" for just this one user, while not altering the credentials of other users. This example is for illustration purposes only; there are no current plans to make this part of rxk5.

On the server side, just like most kerberos 5 services, a conventional keytab file stores the key for `afs/rxk5@`*realm*. This credential is normally kept apart from the system keytab, to simplify machine management. At runtime, when each server process sets up server side connection logic, it passes an array of security objects to rx. For rxk5 this array is extended to include an rxk5 security object.

6.1 localauth

On the OpenAFS server, there is a notion of "localauth" access. This allows server side code to directly access other servers without going through the kdc. This is an intrinsic property of kerberos; since the server has access to its own keytab, it naturally has the ability to construct credentials on anyone's behalf. This is not a new invention of rxk5, rxkad has always had the same feature.

When using MIT kerberos, two functions are required for this, `krb5_encrypt_tkt_part()` and `encode_krb5_ticket()`. Recent versions should be exporting both these functions.

6.2 k5ssl

Provided with rxk5 is a custom implementation of a subset of kerberos, k5ssl. This is designed to solve 4 problems, which are: no external kerberos dependency, kerberos in the kernel, kerberos for refractory environments, and kerberos with hardware acceleration,

no external kerberos An initial requirement was operation of OpenAFS without linking against an external kerberos library. The kerberos 4 and 5 support within OpenAFS still conforms to this standard. This requirement is obsolete today.

kernel Kerberos support in the kernel is uncommon, and portable support is non-existent. Linux provides a very partial subset which (at least at the time) only included des support, so was not even minimally adequate for rxk5. Some versions of solaris support kerberos. The existing userland kerberos libraries were relatively large and would have required extensive customization to run inside the kernel.

refractory environments The Kerberos programming API is badly documented, not standardized, and much of the available documentation is out of date. Vendors who provide commercial support commonly seek to reduce support costs by simplifying the public API that is available. It is not clear if US export restrictions are also an issue here. The result is while kerberos compiled from open source usually provides the functions necessary for rxk5, vendor libraries usually don't. Vendors are understandably reluctant to provide support for algorithms they regard as experimental, which presents rxk5 with a "chicken-and-the-egg" conundrum.

A common answer to this problem is "why don't you just take so-and-so's open source implementation, extract the parts you need, and adapt them to do what you want?" Since when a necessary routine is not available, the routines it calls in turn are also usually not available, this amounts to a triangle of code, about the size of k5ssl, but without the advantages.

hardware acceleration At the time rxk5 was written, openssl[5] was the most widely installed open-source cryptography library, and it happens to include hardware acceleration support on many platforms. Using openssl provides a cheap way to gain access to those resources. It should be recognized that heimdal also supports openssl in certain configurations.

K5ssl as it currently exists builds in two forms: one form is inside the kernel and includes just the subset of operations necessary to act as an rxk5 client. For this form, k5ssl includes its own internal cryptography, and has no need to do asn.1 parsing.

The form outside the kernel includes support to act as either an rxk5 client or server, and also includes support to do initial authentication. In this form, k5ssl relies on openssl to provide cryptography and asn.1 parsing. The initial authentication portion is sufficient to support klog.

The cryptographic layer provided for the kernel is designed to look a lot like openssl, but this is just a convenience. It should be quite feasible to use kernel specific cryptographic services in place of the portable layer provided.

K5ssl is designed to look very much like MIT kerberos to applications. The rxk5 code can mostly ignore the differences between the two.

6.3 ubik

There are many pieces of afs which are driven more or less by human interaction. It is fairly straight-forward to attach rxk5 to these bits, and to provide controls to alter their behavior. And then there is ubik[4]. The interesting part of ubik is hidden inside the database servers, and must originate and manage connections quite separately from any human input. The ubik logic provides a mechanism whereby the client program may supply a security object and index, but this mechanism needs improvement to handle rxk5.

The hook that is supplied is declared this way: The variable

```
extern int
(*ubik_SRXSecurityProc) (void *rock,
    struct rx_securityClass **astr,
    afs_int32 *aindex);
```

points to the routine that is called when setting ubik security up. There is also a global that is not supposed to be known to the application server which is named

```
struct rx_securityClass *ubik_sc[3];
```

At runtime, the hook function is supposed to set `*astr` and return the security index in *aindex. Immediately upon returning, logic in ubik then copies `*astr` to `ubik_sc[*aindex]` and does not make any further reference to `*aindex`. The first and most obvious problem is that `ubik_sc` is too small. A less obvious problem is that there's no way for the application logic to know if the array is in fact large enough for the selected security index. One more design shortcoming is that the security hook cannot in fact choose to set more than one security class. While that's not necessary for rxk5, there's a way to both simplify the code and incidently gain this ability.

The revised hook function looks like this

```
extern int
(*ubik_SRXSecurityProc) (void *rock,
    struct rx_securityClass **sc,
    afs_int32 maxindex);
```

where `**sc` is now just `ubik_sc`, and maxindex is the size of that arrray. The hook function can check `maxindex` to determine that the ubik implementation supports rxk5, before filling in as many elements of `sc[]` as it pleases, and returning the max security index + 1 as its output value.

This revised implementation is more robust. Application server logic that is corrected will gain the ability to determine at runtime how many security classes may be set. Application server logic that uses the default security hook function `afsconf_ServerAuth` will continue to work without changes, because that function has been revised to match.[9]

[9] If there is any orgnization running their own custom security hook on top of ubik within an application, they will have the know-how to debug the resulting core dump if they attempt to use their unmodified code with this new interface.

6.4 client naming and authorization

Kerberos does not provide an authorization mechanism. It merely provides a secure channel and a client identity. The application must then provide some means of taking the client identity on the server and resolving authorization for that identity.

In AFS, there are two methods of authorization. The first is the protection server[2], or ptserver as it is more commonly known, which is used for file access. The second is the userlist, which is used for afs management purposes. The existing logic inside of AFS is kerberos 4 specific; it knows names consist of exactly 2 components, and are typically stored in fixed sized buffers. Kerberos 5 names may have one or more components, and are typically managed using dynamic memory.

Rxk5 provides `rxk5_GetServerInfo2`, which returns the client's identity and some other security information. The client identity is returned as a printable string stored in dynamic memory consisting of the principal name.

For fileserver access, the authorization logic must map the kerberos 5 name into a kerberos 4 name and then call the existing protection service interface to determine access. This logic is similar to that already used to handle kerberos 5 names with rxkad 2b[8]. It would be preferable if the protection service could handle kerberos 5 names directly.

For "userlist" name matching, the existing logic in `afsconf_SuperUser` glues together the name components together and does pattern matching against the whole string. For rxk5, exactly the same thing happens, except the component separator character is the forward slash (`/`).

6.5 afs_token

The existing token mechanism that the afs cache manager provided had a number of shortcomings. Beyond the obvious problem that it didn't support rxk5, less obvious problems include a library interface that is inconvenient to use, a kernel interface that does not closely match the library interface available to user programs, and evolutionary code to marshal and unmarshal data. A particularly odd feature of this mechanism is the use of even/odd time differences to flag "AFS ID" vs. "Unix UID".

For rxk5, some obvious improvements were to make the user calls more convenient to use, and make them match the kernel interface much more closely. A less obvious improvement was to use xdr[14] to marshal data. Since afs already makes extensive use of xdr, this is a natural fit. Xdr provides one bonus feature: it will byte-swap data to match network byte order. This is not necessary for rxk5, but harmless.

The most critical part of the interface turned out to be computing the lengths of items whose internal structure was not necessarily known. This shows up in two places: the pioctl interface, and the per-token data structure. The pioctl interface that OpenAFS implements takes a variable sized byte array, copies it into the kernel, and then copies another variable sized byte array out into a user provided buffer. The actual number of bytes copied out is not directly available to user code. Fortunately, xdr is very good at dealing with upper bounds on buffer sizes. For the per-token data structure, it's necessary to include the length so that logic that does not understand a particular token type can skip past it to read other token types.

The guts of the afs_token interface looks like this:

```
const AFSTOKEN_UNION_NOAUTH = 0;
const AFSTOKEN_UNION_K5 = 5;
union afstoken_soliton switch (int32_t at_type) {
case AFSTOKEN_UNION_K5:
        token_rxk5 at_rxk5;
};
const AFSTOKEN_LENGTH_MAX = 16384;
const AFSTOKEN_CELL_MAX = 64;
const AFSTOKEN_MAX = 8;
typedef opaque token_opaque<AFSTOKEN_LENGTH_MAX>;
struct pioctl_set_token {
        int32_t flags;
        string cell<AFSTOKEN_CELL_MAX>;
        token_opaque tokens<AFSTOKEN_MAX>;
};
```

Here, `afstoken_soliton` can contain one typed credential, and `pioctl_set_token` is a list of credentials for one cell. The data element `token_rxk5` is defined to look very much like a regular kerberos 5 ticket; this is more information than the kernel actually needs, but this reduces the chances that the token interface will need revision. Other token types can have additional cases in the union. Clients processing tokens types that they don't recognize can simply skip them. The rxk5 integration patch also includes code to handle rxkad tokens.

The library interface looks like this:

```
int
ktc_GetTokenEx(afs_int32 index,
    const char *cell,
    pioctl_set_token *a_token);
int
ktc_SetTokenEx(pioctl_set_token *a_token);
```

To support iterating the available cells with tokens, pass 0 for `cell` and iterate passing in `index` = 0, 1, 2, etc. To directly examine a particular cell, pass in the cell name and 0.

6.6 kernel calls

OpenAFS implements pioctl calls for AFS specific operations. For rxk5, the token interface obviously changes, as described in section 6.5 on the previous page. Less obvious changes include the need to get a list of encryption types supported by rxk5. Rather than invent rxk5 specific system calls, all these kernel calls were designed to be as generic as possible.

`VIOCGETTOK2`	`_CVICEIOCTL(7)`	fetch tokens
`VIOCSETTOK2`	`_CVICEIOCTL(8)`	set tokens
`VIOCGETPROP`	`_CVICEIOCTL(10)`	get cache manager capabilities
`VIOCSETPROP`	`_CVICEIOCTL(11)`	set cache manager capabilities

The routines `ktc_GetTokenEx` and `ktc_SetTokenEx` implement fetch and set tokens as previously described.

Capabilities are intended to provide a facility similar to sysctl, but specifically for OpenAFS. That is, a hierarchal name space of simple properties. The interface is specifically designed to make it easy to query multiple properties with one system call.

The one property defined for rxk5 is `"rxk5.enctypes"`, which should contain a read-only string similar to `"18 17 16 23 8 3 2 1 24"`. If the property does not exist, then rxk5 must not be available. The decimal numbers correspond to kerberos 5 encryption types in decreasing order of preference. It is possible to build the copy of k5ssl in the kernel without support for one or more encryption types, in which case those types will not appear in the returned list.

A library routine,

```
     int
     ktc_GetK5Enctypes(krb5_enctype *buf,
         int buflen);
```

is provided which invokes `VIOCGETPROP`, then converts the returned string to a list of kerberos encryption types.

6.7 force flags

In the existing code, there are various command line flags to control the use of authentication, such as `-noauth`, `-localauth`, and in recent versions of OpenAFS, `-encrypt`. These are passed to routines such as `vsu_ClientInit`, `afsconf_ClientAuthEx`. If these exist in the source as anything past an expression like `as->parms[16].items` then these typically show up as variables or arguments named `NoAuthFlag` and `localauth`.

With rxk5, the situation is much more complicated. There are more options, such as `-ktc`, `-k5`, and many of these options are additive. The environment variable `AFS_RXK5_DEFAULT` can also be supplied to alter default behavior. Passing these as many extra individual integers was not at all attractive.

The implementation choice made was to pass these as exactly one integer named `force_flags`, and to provide named manifests to represent the various flags. The correspondence between command flags and internal manifests is a bit ad hoc; some combinations don't make sense or are interpreted differently, and different commands behave slightly differently to preserve historical behavior. The following is a slight simplification of what the code actually does.

`-noauth`	`FORCE_NOAUTH`	unauthenticated access
`-auth`	`FORCE_AUTH`	authenticated access
`-localauth`	`FORCE_SECURE`	use keytab or keyfile
`(-k5)`	`FORCE_K5CC`	use rxk5 and ccache
`-ktc`	`FORCE_KTC`	use token from kernel (rxk5 or rxkad)
`-k4`	`FORCE_RXKAD`	use rxkad only
`-k5`	`FORCE_RXK5`	use rxk5 only
	`FORCE_FAILAUTH`	do not default to noauth if no credentials
`-encrypt`	`FORCE_ENCRYPT`	force the use of encryption

6.8 error codes

MIT implemented a Common Error Description library, otherwise known as comerr[3], way back in 1989. Its purpose is to allow multiple libraries to share a common space of error codes, and to provide a common code path to map error codes to messages. AFS uses comerr to manage error codes. MIT Kerberos 5 uses comerr to manage error codes. Heimdal uses comerr to manage error codes. Kerberos and AFS use disjoint ranges of error codes. In theory, it should be possible to share a single copy of comerr between the two. Sharing a single copy means higher quality error messages with less code.

In practice, it turns out each implementation has its own unique problems and differences. For OpenAFS, the problems include custom error codes with non-standard bases (not a multiple of 2^8). For rxk5, the comerr library was modified to better support custom bases, and extra routines were added to make it more compatible with the MIT library. Several versions were successfully linked with MIT providing a single error code handling path.

More recent versions of OpenAFS have renamed all of the entry points into comerr. Current versions of rxk5 retain the cleaned up custom base logic, but drop the MIT compatibility interfaces. Providing high quality error messages for kerberos 5 errors remains elusive.

7 Future Directions

Rxk5 as described here has been in production use for several years, for several rx-based administrative systems. Some of the items here are already possible or are minor improvements to rxk5, others would be major shifts in direction.

md5/key generation As described above, it would be nice to replace md5 with the kerberos 5 PRF function described in [11]. Doing this requires multiple interoperable implementations, with accessible APIs. A simpler fix would be to pick some newer popular cryptographic hash, and just use that.

kerberos ticket There have been several concerns about using kerberos 5 tickets directly.[10] Rxk5 provides a pluggable interface whereby the application can supply its own ticket decoder. Applications can choose to support their own style of tickets in addition to, or instead of kerberos 5. It would be quite feasible to support diffie-hellman, or other choices this way.

gssapi Gssapi[10] is commonly supposed to be the easiest, most portable, and best way to use kerberos 5. To make a long story short, if this were true for afs/rx, then a suitable security mechanism would have appeared long ago. Current proposals for using gssapi mostly involve the use of some “helper” ticket conversion protocol; this could in fact be done with rxk5 on a per-application basis, perhaps using a custom ticket decoder.

rekey frequency Using a unique key for every data packet is in some sense “over-kill”. The main reason to do this at all is to limit the exposure of the session key; the reason to rekey on every packet is because it allows 2 problems to be solved at the same time. Using a key for more than one packet means the pseudo-header will need to be handled separately. This has particular implications for the packet encrypt and decrypt functions, which will need to include data in the data checksum that is not actually being shipped. Using a key for more than one packet requires that the server and client somehow “agree” on which key is to be used. The obvious choice, ‘sequence number’, is a problem for data retransmits. Something based on call number or other data in the pseudo-header seems more likely. Since in rx, the client decides up front which key to use, there’s no simple way for the server to use the challenge response protocol to force the client to select a different key. It’s been proposed to instead just have ‘short-lived’ connections, but this requires application visible connection refresh logic.

more efficient cryptographic primitives Right now, rxk5 depends mainly on on cryptographic primitives from the underlying kerberos library. For data encryption/decryption, this means some extra data copying to marshal data. An api capable of doing some form of incremental processing, or processing using a list of buffers, could eliminate this need. There is apparently a proposal to add this to the kerberos 5 api, so it probably makes sense to wait to see what that looks like.

[10]Kerberos 5 ticket concerns include: size of the ticket, and cache poisoning on shared access machines.

additional cryptographic suites There are certainly many other cryptographic algorithms other than those used in kerberos 5. It would be nice if those could be used with rxk5. The current rxk5 algorithm is for the client to choose a cryptographic algorithm based directly on the server key types. This approach is simple, robust, and does not require extra packet round trips. Additional cryptographic types would require a security negotiation mechanism separate from kerberos. The extra complexity would present new opportunities for downgrade attacks and other security problems.

hardware cryptography This is in fact already possible with rxk5. There are several ways to layer kerberos on top of openssl, which supports hardware cryptography. Any of those ways will yield an rxk5 implementation that can use hardware cryptography. On a purely economic level: hardware cryptography won't make much practical sense until consumer grade equipment commonly supports it. That's never been true in the past but it may become true; some CPU designs now include hardware cryptographic support, and there's some evidence that cryptographic acceleration based on modern graphics hardware may become useful. In order to be of general use in rx, any such support must include key agility.

8 Thanks

I would like the to thank the following people for their ideas or work on rxk5.

Love Hörnquist Åstrand Work on rxgk, afs_token interface.

Jeffrey Hutzelman early critique/review of functional requirements

Michael Shiplett early rxk5 tester

Matt Benjamin OpenAFS integration

References

[1] Edward R. Zayas, *AFS-3 Programmer's Reference: Specification for the Rx Remote Procedure Call Facility*, Transarc Corporation, 1991

[2] Edward R. Zayas, *AFS-3 Programmer's Reference: Protection Server Interface*, Transarc Corporation, 1991

[3] Raeburn, K., Sommerfeld, B., *A Common Error Description Library for UNIX*, MIT Student Information Processing Board, 1989.

[4] Kazar, M. *Ubik – A Library for Managing Ubiquitous Data,* Information Technology Center, Carnegie-Mellon University, Pittsburgh, PA, 1988.

[5] The openssl home page.

[6] Honeyman, P., Huston, L.B., and Stolarchuk, M.T., *Hijacking AFS,* CITI Technical Report 91-4, 1991.

[7] Ted Anderson, *Specification of FCrypt: Encryption for AFS Remote Procedure Calls*

[8] Jeffrey Hutzelman. *Rx Protocol - rxkad security module "proposal 2b" extension,* 2003.

[9] Rivest, R. *The MD5 Message-Digest Algorithm* RFC 1321, April 1992.

[10] Linn, J., *Generic Security Service Application Program Interface*, RFC 2743, January 2000.

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

[12] Raeburn, K., *Advanced Encryption Standard (AES) Encryption for Kerberos 5* RFC 3962, February 2005.

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

[14] Eisler, M., *XDR: External Data Representation Standard*, RFC 4506, May 2006.