Network Working Group
Request for Comments: 2203
Category: Standards Track

M. Eisler
A. Chiu
L. Ling

September 1997

RPCSEC_GSS Protocol Specification

Status of this Memo

Abstract

This memo describes an ONC/RPC security flavor that allows RPC protocols to access the Generic Security Services Application Programming Interface (referred to henceforth as GSS-API).

Table of Contents

1. Introduction

This document describes the protocol used by the RPCSEC_GSS security flavor. Security flavors have been called authentication flavors for historical reasons. This memo recognizes that there are two other security services besides authentication, integrity, and privacy, and so defines a new RPCSEC_GSS security flavor.

The protocol is described using the XDR language [Srinivasan-xdr]. The reader is assumed to be familiar with ONC RPC and the security flavor mechanism [Srinivasan-rpc]. The reader is also assumed to be familiar with the GSS-API framework [Linn]. The RPCSEC_GSS security flavor uses GSS-API interfaces to provide security services that are independent of the underlying security mechanism.

2. The ONC RPC Message Protocol

This memo refers to the following XDR types of the ONC RPC protocol, which are described in the document entitled Remote Procedure Call Protocol Specification Version 2 [Srinivasan-rpc]:

```
msg_type
reply_stat
auth_flavor
accept_stat
reject_stat
auth_stat
opaque_auth
rpc_msg
call_body
reply_body
```

```
      accepted_reply
      rejected_reply
```

3. Flavor Number Assignment

The RPCSEC_GSS security flavor has been assigned the value of 6:

```
   enum auth_flavor {
       ...
       RPCSEC_GSS = 6      /* RPCSEC_GSS security flavor */
   };
```

4. New auth_stat Values

RPCSEC_GSS requires the addition of two new values to the auth_stat enumerated type definition:

```
   enum auth_stat {
           ...
           /*
            * RPCSEC_GSS errors
            */
           RPCSEC_GSS_CREDPROBLEM = 13,
           RPCSEC_GSS_CTXPROBLEM = 14
   };
```

The descriptions of these two new values are defined later in this memo.

5. Elements of the RPCSEC_GSS Security Protocol

An RPC session based on the RPCSEC_GSS security flavor consists of three phases: context creation, RPC data exchange, and context destruction. In the following discussion, protocol elements for these three phases are described.

The following description of the RPCSEC_GSS protocol uses some of the definitions within XDR language description of the RPC protocol.

Context creation and destruction use control messages that are not dispatched to service procedures registered by an RPC server. The program and version numbers used in these control messages are the same as the RPC service’s program and version numbers. The procedure number used is NULLPROC (zero). A field in the credential information (the gss_proc field which is defined in the rpc_gss_cred_t structure below) specifies whether a message is to be interpreted as a control message or a regular RPC message. If this field is set to RPCSEC_GSS_DATA, no control action is implied; in

this case, it is a regular data message. If this field is set to any other value, a control action is implied. This is described in the following sections.

Just as with normal RPC data exchange messages, the transaction identifier (the xid field in struct rpc_msg), should be set to unique values on each call for context creation and context destruction.

The following definitions are used for describing the protocol.

```
   /* RPCSEC_GSS control procedures */


   enum rpc_gss_proc_t {
           RPCSEC_GSS_DATA = 0,
           RPCSEC_GSS_INIT = 1,
           RPCSEC_GSS_CONTINUE_INIT = 2,
           RPCSEC_GSS_DESTROY = 3
   };

   /* RPCSEC_GSS services */

   enum rpc_gss_service_t {
       /* Note: the enumerated value for 0 is reserved. */
       rpc_gss_svc_none = 1,
       rpc_gss_svc_integrity = 2,
       rpc_gss_svc_privacy = 3
   };

   /* Credential */

   /*
    * Note: version 0 is reserved for possible future
    * definition of a version negotiation protocol
    *
    */
   #define RPCSEC_GSS_VERS_1 1

   struct rpc_gss_cred_t {
       union switch (unsigned int version) { /* version of
                                                RPCSEC_GSS */
       case RPCSEC_GSS_VERS_1:
           struct {
               rpc_gss_proc_t gss_proc;  /* control procedure */
               unsigned int seq_num;   /* sequence number */
               rpc_gss_service_t service; /* service used */
               opaque handle<>;       /* context handle */
           } rpc_gss_cred_vers_1_t;
```

```
        }
    };

    /* Maximum sequence number value */

    #define MAXSEQ 0x80000000
```

5.1. Version Selection

This document defines just one protocol version (RPCSEC_GSS_VERS_1). The client should assume that the server supports RPCSEC_GSS_VERS_1 and issue a Context Creation message (as described in the section RPCSEC_GSS_VERS_1, the RPC response will have a reply_stat of MSG_DENIED, a rejection status of AUTH_ERROR, and an auth_stat of AUTH_REJECTED_CRED.

5.2. Context Creation

Before RPC data is exchanged on a session using the RPCSEC_GSS flavor, a context must be set up between the client and the server. Context creation may involve zero or more RPC exchanges. The number of exchanges depends on the security mechanism.

5.2.1. Mechanism and QOP Selection

There is no facility in the RPCSEC_GSS protocol to negotiate GSS-API mechanism identifiers or QOP values. At minimum, it is expected that implementations of the RPCSEC_GSS protocol provide a means to:

* specify mechanism identifiers, QOP values, and RPCSEC_GSS service values on the client side, and to

* enforce mechanism identifiers, QOP values, and RPCSEC_GSS service values on a per-request basis on the server side.

It is necessary that above capabilities exist so that applications have the means to conform the required set of required set of <mechanism, QOP, service> tuples (See the section entitled Set of GSS-API Mechanisms). An application may negotiate <mechanism, QOP, service> selection within its protocol or via an out of band protocol. Hence it may be necessary for RPCSEC_GSS implementations to provide programming interfaces for the specification and enforcement of <mechanism, QOP, service>.

Additionally, implementations may depend on negotiation schemes constructed as pseudo-mechanisms under the GSS-API. Because such schemes are below the GSS-API layer, the RPCSEC_GSS protocol, as specified in this document, can make use of them.

5.2.2. Context Creation Requests

The first RPC request from the client to the server initiates context creation. Within the RPC message protocol's call_body structure, rpcvers is set to 2. prog and vers are always those for the service being accessed. The proc is always set to NULLPROC (zero).

Within the RPC message protocol's cred structure, flavor is set to RPCSEC_GSS (6). The opaque data of the cred structure (the body field) constituting the credential encodes the rpc_gss_cred_t structure defined previously.

The values of the fields contained in the rpc_gss_cred_t structure are set as follows. The version field is set to the version of the RPCSEC_GSS protocol the client wants to use. The remainder of this memo documents version RPCSEC_GSS_VERS_1 of RPCSEC_GSS, and so the version field would be set to RPCSEC_GSS_VERS_1. The gss_proc field must be set to RPCSEC_GSS_INIT for the first creation request. In subsequent creation requests, the gss_proc field must be set to RPCSEC_GSS_CONTINUE_INIT. In a creation request, the seq_num and service fields are undefined and both must be ignored by the server. In the first creation request, the handle field is NULL (opaque data of zero length). In subsequent creation requests, handle must be equal to the value returned by the server. The handle field serves as the identifier for the context, and will not change for the duration of the context, including responses to RPCSEC_GSS_CONTINUE_INIT.

The verifier field in the RPC message header is also described by the opaque_auth structure. All creation requests have the NULL verifier (AUTH_NONE flavor with zero length opaque data).

Following the verifier are the call data (procedure specific parameters). Note that the proc field of the call_body structure is set to NULLPROC, and thus normally there would be zero octets following the verifier. However, since there is no RPC data exchange during a context creation, it is safe to transfer information following the verifier. It is necessary to "overload" the call data in this way, rather than pack the GSS-API token into the RPC header, because RPC Version 2 restricts the amount of data that can be sent in the header. The opaque body of the credential and verifier fields can be each at most 400 octets long, and GSS tokens can be longer than 800 octets.

The call data for a context creation request is described by the following structure for all creation requests:

```
struct rpc_gss_init_arg {
    opaque gss_token<>;
};
```

Here, gss_token is the token returned by the call to GSS-API's GSS_Init_sec_context() routine, opaquely encoded. The value of this field will likely be different in each creation request, if there is more than one creation request. If no token is returned by the call to GSS_Init_sec_context(), the context must have been created (assuming no errors), and there will not be any more creation requests.

When GSS_Init_sec_context() is called, the parameters replay_det_req_flag and sequence_req_flag must be turned off. The reasons for this are:

* ONC RPC can be used over unreliable transports and provides no layer to reliably re-assemble messages. Thus it is possible for gaps in message sequencing to occur, as well as out of order messages.

* RPC servers can be multi-threaded, and thus the order in which GSS-API messages are signed or wrapped can be different from the order in which the messages are verified or unwrapped, even if the requests are sent on reliable transports.

* To maximize convenience of implementation, the order in which an ONC RPC entity will verify the header and verify/unwrap the body of an RPC call or reply is left unspecified.

The RPCSEC_GSS protocol provides for protection from replay attack, yet tolerates out-of-order delivery or processing of messages and tolerates dropped requests.

5.2.3. Context Creation Responses

5.2.3.1. Context Creation Response - Successful Acceptance

The response to a successful creation request has an MSG_ACCEPTED response with a status of SUCCESS. The results field encodes a response with the following structure:

```
struct rpc_gss_init_res {
        opaque handle<>;
        unsigned int gss_major;
        unsigned int gss_minor;
        unsigned int seq_window;
        opaque gss_token<>;
};
```

Here, handle is non-NULL opaque data that serves as the context identifier. The client must use this value in all subsequent requests whether control messages or otherwise). The gss_major and gss_minor fields contain the results of the call to GSS_Accept_sec_context() executed by the server. The values for the gss_major field are defined in Appendix A of this document. The values for the gss_minor field are GSS-API mechanism specific and are defined in the mechanism's specification. If gss_major is not one of GSS_S_COMPLETE or GSS_S_CONTINUE_NEEDED, the context setup has failed; in this case handle and gss_token must be set to NULL by the server. The value of gss_minor is dependent on the value of gss_major and the security mechanism used. The gss_token field contains any token returned by the GSS_Accept_sec_context() call executed by the server. A token may be returned for both successful values of gss_major. If the value is GSS_S_COMPLETE, it indicates that the server is not expecting any more tokens, and the RPC Data Exchange phase must begin on the subsequent request from the client. If the value is GSS_S_CONTINUE_NEEDED, the server is expecting another token. Hence the client must send at least one more creation request (with gss_proc set to RPCSEC_GSS_CONTINUE_INIT in the request's credential) carrying the required token.

In a successful response, the seq_window field is set to the sequence window length supported by the server for this context. This window specifies the maximum number of client requests that may be outstanding for this context. The server will accept "seq_window" requests at a time, and these may be out of order. The client may use this number to determine the number of threads that can simultaneously send requests on this context.

If gss_major is GSS_S_COMPLETE, the verifier's (the verf element in the response) flavor field is set to RPCSEC_GSS, and the body field set to the checksum of the seq_window (in network order). The QOP used for this checksum is 0 (zero), which is the default QOP. For all other values of gss_major, a NULL verifier (AUTH_NONE flavor with zero-length opaque data) is used.

5.2.3.1.1. Client Processing of Successful Context Creation Responses

If the value of gss_major in the response is GSS_S_CONTINUE_NEEDED, then the client, per the GSS-API specification, must invoke GSS_Init_sec_context() using the token returned in gss_token in the context creation response. The client must then generate a context creation request, with gss_proc set to RPCSEC_GSS_CONTINUE_INIT.

If the value of gss_major in the response is GSS_S_COMPLETE, and if the client's previous invocation of GSS_Init_sec_context() returned a gss_major value of GSS_S_CONTINUE_NEEDED, then the client, per the GSS-API specification, must invoke GSS_Init_sec_context() using the token returned in gss_token in the context creation response. If GSS_Init_sec_context() returns GSS_S_COMPLETE, the context is successfully set up, and the RPC data exchange phase must begin on the subsequent request from the client.

5.2.3.2. Context Creation Response - Unsuccessful Cases

An MSG_ACCEPTED reply (to a creation request) with an acceptance status of other than SUCCESS has a NULL verifier (flavor set to AUTH_NONE, and zero length opaque data in the body field), and is formulated as usual for different status values.

An MSG_DENIED reply (to a creation request) is also formulated as usual. Note that MSG_DENIED could be returned because the server's RPC implementation does not recognize the RPCSEC_GSS security flavor. RFC 1831 does not specify the appropriate reply status in this instance, but common implementation practice appears to be to return a rejection status of AUTH_ERROR with an auth_stat of AUTH_REJECTEDCRED. Even though two new values (RPCSEC_GSS_CREDPROBLEM and RPCSEC_GSS_CTXPROBLEM) have been defined for the auth_stat type, neither of these two can be returned in responses to context creation requests. The auth_stat new values can be used for responses to normal (data) requests. This is described later.

MSG_DENIED might also be returned if the RPCSEC_GSS version number in the credential is not supported on the server. In that case, the server returns a rejection status of AUTH_ERROR, with an auth_stat of

AUTH_REJECTED_CRED.

5.3. RPC Data Exchange

The data exchange phase is entered after a context has been successfully set up. The format of the data exchanged depends on the security service used for the request. Although clients can change the security service and QOP used on a per-request basis, this may not be acceptable to all RPC services; some RPC services may "lock" the data exchange phase into using the QOP and service used on the first data exchange message. For all three modes of service (no data integrity, data integrity, data privacy), the RPC request header has the same format.

5.3.1. RPC Request Header

The credential has the opaque_auth structure described earlier. The flavor field is set to RPCSEC_GSS. The credential body is created by XDR encoding the rpc_gss_cred_t structure listed earlier into an octet stream, and then opaquely encoding this octet stream as the body field.

Values of the fields contained in the rpc_gss_cred_t structure are set as follows. The version field is set to same version value that was used to create the context, which within the scope of this memo will always be RPCSEC_GSS_VERS_1. The gss_proc field is set to RPCSEC_GSS_DATA. The service field is set to indicate the desired service (one of rpc_gss_svc_none, rpc_gss_svc_integrity, or rpc_gss_svc_privacy). The handle field is set to the context handle value received from the RPC server during context creation. The seq_num field can start at any value below MAXSEQ, and must be incremented (by one or more) for successive requests. Use of sequence numbers is described in detail when server processing of the request is discussed.

The verifier has the opaque_auth structure described earlier. The flavor field is set to RPCSEC_GSS. The body field is set as follows. The checksum of the RPC header (up to and including the credential) is computed using the GSS_GetMIC() call with the desired QOP. This returns the checksum as an opaque octet stream and its length. This is encoded into the body field. Note that the QOP is not explicitly specified anywhere in the request. It is implicit in the checksum or encrypted data. The same QOP value as is used for the header checksum must also be used for the data (for checksumming or encrypting), unless the service used for the request is rpc_gss_svc_none.

5.3.2. RPC Request Data

5.3.2.1. RPC Request Data - No Data Integrity

If the service specified is rpc_gss_svc_none, the data (procedure arguments) are not integrity or privacy protected. They are sent in exactly the same way as they would be if the AUTH_NONE flavor were used (following the verifier). Note, however, that since the RPC header is integrity protected, the sender will still be authenticated in this case.

5.3.2.2. RPC Request Data - With Data Integrity

When data integrity is used, the request data is represented as follows:

```
struct rpc_gss_integ_data {
    opaque databody_integ<>;
    opaque checksum<>;
};
```

The databody_integ field is created as follows. A structure consisting of a sequence number followed by the procedure arguments is constructed. This is shown below as the type rpc_gss_data_t:

```
struct rpc_gss_data_t {
    unsigned int seq_num;
    proc_req_arg_t arg;
};
```

Here, seq_num must have the same value as in the credential. The type proc_req_arg_t is the procedure specific XDR type describing the procedure arguments (and so is not specified here). The octet stream corresponding to the XDR encoded rpc_gss_data_t structure and its length are placed in the databody_integ field. Note that because the XDR type of databody_integ is opaque, the XDR encoding of databody_integ will include an initial four octet length field, followed by the XDR encoded octet stream of rpc_gss_data_t.

The checksum field represents the checksum of the XDR encoded octet stream corresponding to the XDR encoded rpc_gss_data_t structure (note, this is not the checksum of the databody_integ field). This is obtained using the GSS_GetMIC() call, with the same QOP as was used to compute the header checksum (in the verifier). The

GSS_GetMIC() call returns the checksum as an opaque octet stream and its length. The checksum field of struct rpc_gss_integ_data has an XDR type of opaque. Thus the checksum length from GSS_GetMIC() is encoded as a four octet length field, followed by the checksum, padded to a multiple of four octets.

5.3.2.3. RPC Request Data - With Data Privacy

When data privacy is used, the request data is represented as follows:

```
struct rpc_gss_priv_data {
    opaque databody_priv<>
};
```

The databody_priv field is created as follows. The rpc_gss_data_t structure described earlier is constructed again in the same way as for the case of data integrity. Next, the GSS_Wrap() call is invoked to encrypt the octet stream corresponding to the rpc_gss_data_t structure, using the same value for QOP (argument qop_req to GSS_Wrap()) as was used for the header checksum (in the verifier) and conf_req_flag (an argument to GSS_Wrap()) of TRUE. The GSS_Wrap() call returns an opaque octet stream (representing the encrypted rpc_gss_data_t structure) and its length, and this is encoded as the databody_priv field. Since databody_priv has an XDR type of opaque, the length returned by GSS_Wrap() is encoded as the four octet length, followed by the encrypted octet stream (padded to a multiple of four octets).

5.3.3. Server Processing of RPC Data Requests

5.3.3.1. Context Management

When a request is received by the server, the following are verified to be acceptable:

* the version number in the credential

* the service specified in the credential

* the context handle specified in the credential

* the header checksum in the verifier (via GSS_VerifyMIC())

* the sequence number (seq_num) specified in the credential (more on this follows)

The gss_proc field in the credential must be set to RPCSEC_GSS_DATA for data requests (otherwise, the message will be interpreted as a control message).

The server maintains a window of "seq_window" sequence numbers, starting with the last sequence number seen and extending backwards. If a sequence number higher than the last number seen is received (AND if GSS_VerifyMIC() on the header checksum from the verifier returns GSS_S_COMPLETE), the window is moved forward to the new sequence number. If the last sequence number seen is N, the server is prepared to receive requests with sequence numbers in the range N through (N - seq_window + 1), both inclusive. If the sequence number received falls below this range, it is silently discarded. If the sequence number is within this range, and the server has not seen it, the request is accepted, and the server turns on a bit to "remember" that this sequence number has been seen. If the server determines that it has already seen a sequence number within the window, the request is silently discarded. The server should select a seq_window value based on the number requests it expects to process simultaneously. For example, in a threaded implementation seq_window might be equal to the number of server threads. There are no known security issues with selecting a large window. The primary issue is how much space the server is willing to allocate to keep track of requests received within the window.

The reason for discarding requests silently is that the server is unable to determine if the duplicate or out of range request was due to a sequencing problem in the client, network, or the operating system, or due to some quirk in routing, or a replay attack by an intruder. Discarding the request allows the client to recover after timing out, if indeed the duplication was unintentional or well intended. Note that a consequence of the silent discard is that clients may increment the seq_num by more than one. The effect of this is that the window will move forward more quickly. It is not believed that there is any benefit to doing this.

Note that the sequence number algorithm requires that the client increment the sequence number even if it is retrying a request with the same RPC transaction identifier. It is not infrequent for clients to get into a situation where they send two or more attempts and a slow server sends the reply for the first attempt. With RPCSEC_GSS, each request and reply will have a unique sequence number. If the client wishes to improve turn around time on the RPC call, it can cache the RPCSEC_GSS sequence number of each request it sends. Then when it receives a response with a matching RPC transaction identifier, it can compute the checksum of each sequence number in the cache to try to match the checksum in the reply’s verifier.

The data is decoded according to the service specified in the credential. In the case of integrity or privacy, the server ensures that the QOP value is acceptable, and that it is the same as that used for the header checksum in the verifier. Also, in the case of integrity or privacy, the server will reject the message (with a reply status of MSG_ACCEPTED, and an acceptance status of GARBAGE_ARGS) if the sequence number embedded in the request body is different from the sequence number in the credential.

5.3.3.2. Server Reply - Request Accepted

An MSG_ACCEPTED reply to a request in the data exchange phase will have the verifier's (the verf element in the response) flavor field set to RPCSEC_GSS, and the body field set to the checksum (the output of GSS_GetMIC()) of the sequence number (in network order) of the corresponding request. The QOP used is the same as the QOP used for the corresponding request.

If the status of the reply is not SUCCESS, the rest of the message is formatted as usual.

If the status of the message is SUCCESS, the format of the rest of the message depends on the service specified in the corresponding request message. Basically, what follows the verifier in this case are the procedure results, formatted in different ways depending on the requested service.

If no data integrity was requested, the procedure results are formatted as for the AUTH_NONE security flavor.

If data integrity was requested, the results are encoded in exactly the same way as the procedure arguments were in the corresponding request. See the section 'RPC Request Data - With Data Integrity.' The only difference is that the structure representing the procedure's result - proc_res_arg_t - must be substituted in place of the request argument structure proc_req_arg_t. The QOP used for the checksum must be the same as that used for constructing the reply verifier.

If data privacy was requested, the results are encoded in exactly the same way as the procedure arguments were in the corresponding request. See the section 'RPC Request Data - With Data Privacy.' The QOP used for encryption must be the same as that used for constructing the reply verifier.

5.3.3.3. Server Reply - Request Denied

An MSG_DENIED reply (to a data request) is formulated as usual. Two new values (RPCSEC_GSS_CREDPROBLEM and RPCSEC_GSS_CTXPROBLEM) have been defined for the auth_stat type. When the reason for denial of the request is a reject_stat of AUTH_ERROR, one of the two new auth_stat values could be returned in addition to the existing values. These two new values have special significance from the existing reasons for denial of a request.

The server maintains a list of contexts for the clients that are currently in session with it. Normally, a context is destroyed when the client ends the session corresponding to it. However, due to resource constraints, the server may destroy a context prematurely (on an LRU basis, or if the server machine is rebooted, for example). In this case, when a client request comes in, there may not be a context corresponding to its handle. The server rejects the request, with the reason RPCSEC_GSS_CREDPROBLEM in this case. Upon receiving this error, the client must refresh the context - that is, reestablish it after destroying the old one - and try the request again. This error is also returned if the context handle matches that of a different context that was allocated after the client's context was destroyed (this will be detected by a failure in verifying the header checksum).

If the GSS_VerifyMIC() call on the header checksum (contained in the verifier) fails to return GSS_S_COMPLETE, the server rejects the request and returns an auth_stat of RPCSEC_GSS_CREDPROBLEM.

When the client's sequence number exceeds the maximum the server will allow, the server will reject the request with the reason RPCSEC_GSS_CTXPROBLEM. Also, if security credentials become stale while in use (due to ticket expiry in the case of the Kerberos V5 mechanism, for example), the failures which result cause the RPCSEC_GSS_CTXPROBLEM reason to be returned. In these cases also, the client must refresh the context, and retry the request.

For other errors, retrying will not rectify the problem and the client must not refresh the context until the problem causing the client request to be denied is rectified.

If the version field in the credential does not match the version of RPCSEC_GSS that was used when the context was created, the AUTH_BADCRED value is returned.

If there is a problem with the credential, such a bad length, illegal control procedure, or an illegal service, the appropriate auth_stat status is AUTH_BADCRED.

Other errors can be returned as appropriate.

5.3.3.4. Mapping of GSS-API Errors to Server Responses

During the data exchange phase, the server may invoke GSS_GetMIC(), GSS_VerifyMIC(), GSS_Unwrap(), and GSS_Wrap(). If any of these routines fail to return GSS_S_COMPLETE, then various unsuccessful responses can be returned. The are described as follows for each of the aforementioned four interfaces.

5.3.3.4.1. GSS_GetMIC() Failure

When GSS_GetMIC() is called to generate the verifier in the response, a failure results in an RPC response with a reply status of MSG_DENIED, reject status of AUTH_ERROR and an auth status of RPCSEC_GSS_CTXPROBLEM.

When GSS_GetMIC() is called to sign the call results (service is rpc_gss_svc_integrity), a failure results in no RPC response being sent. Since ONC RPC server applications will typically control when a response is sent, the failure indication will be returned to the server application and it can take appropriate action (such as logging the error).

5.3.3.4.2. GSS_VerifyMIC() Failure

When GSS_VerifyMIC() is called to verify the verifier in request, a failure results in an RPC response with a reply status of MSG_DENIED, reject status of AUTH_ERROR and an auth status of RPCSEC_GSS_CREDPROBLEM.

When GSS_VerifyMIC() is called to verify the call arguments (service is rpc_gss_svc_integrity), a failure results in an RPC response with a reply status of MSG_ACCEPTED, and an acceptance status of GARBAGE_ARGS.

5.3.3.4.3. GSS_Unwrap() Failure

When GSS_Unwrap() is called to decrypt the call arguments (service is rpc_gss_svc_privacy), a failure results in an RPC response with a reply status of MSG_ACCEPTED, and an acceptance status of GARBAGE_ARGS.

5.3.3.4.4. GSS_Wrap() Failure

When GSS_Wrap() is called to encrypt the call results (service is rpc_gss_svc_privacy), a failure results in no RPC response being sent. Since ONC RPC server applications will typically control when a

response is sent, the failure indication will be returned to the application and it can take appropriate action (such as logging the error).

5.4. Context Destruction

When the client is done using the session, it must send a control message informing the server that it no longer requires the context. This message is formulated just like a data request packet, with the following differences: the credential has gss_proc set to RPCSEC_GSS_DESTROY, the procedure specified in the header is NULLPROC, and there are no procedure arguments. The sequence number in the request must be valid, and the header checksum in the verifier must be valid, for the server to accept the message. The server sends a response as it would to a data request. The client and server must then destroy the context for the session.

If the request to destroy the context fails for some reason, the client need not take any special action. The server must be prepared to deal with situations where clients never inform the server that they no longer are in session and so don't need the server to maintain a context. An LRU mechanism or an aging mechanism should be employed by the server to clean up in such cases.

6. Set of GSS-API Mechanisms

RPCSEC_GSS is effectively a "pass-through" to the GSS-API layer, and as such it is inappropriate for the RPCSEC_GSS specification to enumerate a minimum set of required security mechanisms and/or quality of protections.

If an application protocol specification references RPCSEC_GSS, the protocol specification must list a mandatory set of { mechanism, QOP, service } triples, such that an implementation cannot claim conformance to the protocol specification unless it implements the set of triples. Within each triple, mechanism is a GSS-API security mechanism, QOP is a valid quality-of-protection within the mechanism, and service is either rpc_gss_svc_integrity or rpc_gss_svc_privacy.

For example, a network filing protocol built on RPC that depends on RPCSEC_GSS for security, might require that Kerberos V5 with the default QOP using the rpc_gss_svc_integrity service be supported by implementations conforming to the network filing protocol specification.

7. Security Considerations

7.1. Privacy of Call Header

The reader will note that for the privacy option, only the call arguments and results are encrypted. Information about the application in the form of RPC program number, program version number, and program procedure number is transmitted in the clear. Encrypting these fields in the RPC call header would have changed the size and format of the call header. This would have required revising the RPC protocol which was beyond the scope of this proposal. Storing the encrypted numbers in the credential would have obviated a protocol change, but would have introduced more overloading of fields and would have made implementations of RPC more complex. Even if the fields were encrypted somehow, in most cases an attacker can determine the program number and version number by examining the destination address of the request and querying the rpcbind service on the destination host [Srinivasan-bind]. In any case, even by not encrypting the three numbers, RPCSEC_GSS still improves the state of security over what existing RPC services have had available previously. Implementors of new RPC services that are concerned about this risk may opt to design in a "sub-procedure" field that is included in the service specific call arguments.

7.2. Sequence Number Attacks

7.2.1. Sequence Numbers Above the Window

An attacker cannot coax the server into raising the sequence number beyond the range the legitimate client is aware of (and thus engineer a denial of server attack) without constructing an RPC request that will pass the header checksum. If the cost of verifying the header checksum is sufficiently large (depending on the speed of the processor doing the checksum and the cost of checksum algorithm), it is possible to envision a denial of service attack (vandalism, in the form of wasting processing resources) whereby the attacker sends requests that are above the window. The simplest method might be for the attacker to monitor the network traffic and then choose a sequence number that is far above the current sequence number. Then the attacker can send bogus requests using the above window sequence number.

7.2.2. Sequence Numbers Within or Below the Window

If the attacker sends requests that are within or below the window, then even if the header checksum is successfully verified, the server will silently discard the requests because the server assumes it has already processed the request. In this case, a server can optimize by

skipping the header checksum verification if the sequence number is below the window, or if it is within the window, not attempt the checksum verification if the sequence number has already been seen.

7.3. Message Stealing Attacks

This proposal does not address attacks where an attacker can block or steal messages without being detected by the server. To implement such protection would be tantamount to assuming a state in the RPC service. RPCSEC_GSS does not worsen this situation.

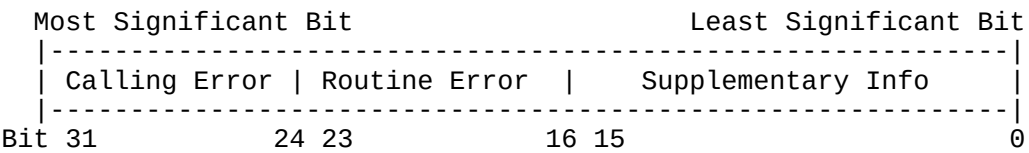
Appendix A. GSS-API Major Status Codes

The GSS-API definition [Linn] does not include numerical values for the various GSS-API major status codes. It is expected that this will be addressed in future RFC. Until then, this appendix defines the values for each GSS-API major status code listed in the GSS-API definition. If in the future, the GSS-API definition defines values for the codes that are different than what follows, then implementors of RPCSEC_GSS will be obliged to map them into the values defined below. If in the future, the GSS-API definition defines additional status codes not defined below, then the RPCSEC_GSS definition will subsume those additional values.

Here are the definitions of each GSS_S_* major status that the implementor of RPCSEC_GSS can expect in the gss_major major field of rpc_gss_init_res. These definitions are not in RPC description language form. The numbers are in base 16 (hexadecimal):

GSS_S_COMPLETE	0x00000000
GSS_S_CONTINUE_NEEDED	0x00000001
GSS_S_DUPLICATE_TOKEN	0x00000002
GSS_S_OLD_TOKEN	0x00000004
GSS_S_UNSEQ_TOKEN	0x00000008
GSS_S_GAP_TOKEN	0x00000010
GSS_S_BAD_MECH	0x00010000
GSS_S_BAD_NAME	0x00020000
GSS_S_BAD_NAMETYPE	0x00030000
GSS_S_BAD_BINDINGS	0x00040000
GSS_S_BAD_STATUS	0x00050000
GSS_S_BAD_MIC	0x00060000
GSS_S_BAD_SIG	0x00060000
GSS_S_NO_CRED	0x00070000
GSS_S_NO_CONTEXT	0x00080000
GSS_S_DEFECTIVE_TOKEN	0x00090000
GSS_S_DEFECTIVE_CREDENTIAL	0x000a0000
GSS_S_CREDENTIALS_EXPIRED	0x000b0000
GSS_S_CONTEXT_EXPIRED	0x000c0000
GSS_S_FAILURE	0x000d0000
GSS_S_BAD_QOP	0x000e0000
GSS_S_UNAUTHORIZED	0x000f0000
GSS_S_UNAVAILABLE	0x00100000
GSS_S_DUPLICATE_ELEMENT	0x00110000
GSS_S_NAME_NOT_MN	0x00120000
GSS_S_CALL_INACCESSIBLE_READ	0x01000000
GSS_S_CALL_INACCESSIBLE_WRITE	0x02000000
GSS_S_CALL_BAD_STRUCTURE	0x03000000

Note that the GSS-API major status is split into three fields as follows:



Up to one status in the Calling Error field can be logically ORed with up to one status in the Routine Error field which in turn can be logically ORed with zero or more statuses in the Supplementary Info field. If the resulting major status has a non-zero Calling Error and/or a non-zero Routine Error, then the applicable GSS-API operation has failed. For purposes of RPCSEC_GSS, this means that the GSS_Accept_sec_context() call executed by the server has failed.

If the major status is equal GSS_S_COMPLETE, then this indicates the absence of any Errors or Supplementary Info.

The meanings of most of the GSS_S_* status are defined in the GSS-API definition, which the exceptions of:

GSS_S_BAD_MIC This code has the same meaning as GSS_S_BAD_SIG.

GSS_S_CALL_INACCESSIBLE_READ
A required input parameter could not be read.

GSS_S_CALL_INACCESSIBLE_WRITE
A required input parameter could not be written.

GSS_S_CALL_BAD_STRUCTURE
A parameter was malformed.

Acknowledgements

Much of the protocol was based on the AUTH_GSSAPI security flavor developed by Open Vision Technologies [Jaspan]. In particular, we acknowledge Barry Jaspan, Marc Horowitz, John Linn, and Ellen McDermott.

Raj Srinivasan designed RPCSEC_GSS [Eisler] with input from Mike Eisler. Raj, Roland Schemers, Lin Ling, and Alex Chiu contributed to Sun Microsystems' implementation of RPCSEC_GSS.

Brent Callaghan, Marc Horowitz, Barry Jaspan, John Linn, Hilarie Orman, Martin Rex, Ted Ts'o, and John Wroclawski analyzed the specification and gave valuable feedback.

Steve Nahm and Kathy Slattery reviewed various drafts of this specification.

Much of content of Appendix A was excerpted from John Wray's Work in Progress on GSS-API Version 2 C-bindings.

References

[Eisler] Eisler, M., Schemers, R., and Srinivasan, R. (1996). "Security Mechanism Independence in ONC RPC," Proceedings of the Sixth Annual USENIX Security Symposium, pp. 51-65.

[Jaspan] Jaspan, B. (1995). "GSS-API Security for ONC RPC," '95 Proceedings of The Internet Society Symposium on Network and Distributed System Security, pp. 144- 151.

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

[Srinivasan-bind] Srinivasan, R., "Binding Protocols for ONC RPC Version 2", RFC 1833, August 1995.

[Srinivasan-rpc] Srinivasan, R., "RPC: Remote Procedure Call Protocol Specification Version 2", RFC 1831, August 1995.

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

Authors' Addresses

Michael Eisler
Sun Microsystems, Inc.
M/S UCOS03
2550 Garcia Avenue
Mountain View, CA 94043

Phone: +1 (719) 599-9026
EMail: mre@eng.sun.com

Alex Chiu
Sun Microsystems, Inc.
M/S UMPK17-203
2550 Garcia Avenue
Mountain View, CA 94043

Phone: +1 (415) 786-6465
EMail: hacker@eng.sun.com

Lin Ling
Sun Microsystems, Inc.
M/S UMPK17-201
2550 Garcia Avenue
Mountain View, CA 94043

Phone: +1 (415) 786-5084
EMail: lling@eng.sun.com