Network Working Group
Internet-Draft
Intended status: Informational
Expires: May 24, 2014

B. Kaduk
MIT

November 20, 2013

Structure of the GSS Negotiation Loop
draft-kaduk-kitten-gss-loop-01

Abstract

This document specifies the generic structure of the negotiation loop to establish a GSS security context between initiator and acceptor. The control flow of the loop is indicated for both parties, including error conditions, and indications are given for where application-specific behavior must be specified.

Status of This Memo

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

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

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

This Internet-Draft will expire on May 24, 2014.

1. Introduction

The Generic Security Service Application Program Intervace version 2 [RFC2743] provides a generic interface for security services, in the form of an abstraction layer over the underlying security mechanisms that an application may use. A GSS initiator and acceptor exchange messages, called tokens, until a security context is established. Such a security context allows for mutual authentication of the two parties, the passing of confidential or integrity-protected messages between the initiator and acceptor, the generation of identical pseudo-random bit strings by both participants [RFC4401], and more. The number of tokens which must be exchanged between initiator and acceptor in order to establish the security context is dependent on the underlying mechanism as well as the desired properties of the security context, and is in general not known to the application. Accordingly, the application's control flow must include a loop within which GSS security context tokens are exchanged, which terminates upon successful establishment of a security context (or an error condition).

The GSS-API C bindings [RFC2744] provide some example code for such a negotiation loop, but this code does not specify the application's behavior on unexpected or error conditions. As such, individual application protocol specifications have had to specify the structure of their GSS negotiation loops, including error handling, on a per-protocol basis. [RFC4462], [RFC3645], [RFC5801], [RFC4752], [RFC2203] This represents a substantial duplication of effort, and the various specifications go into different levels of detail and describe different possible error conditions. It is therefore preferable to have the structure of the GSS negotiation loop, including error conditions and token passing, described in a single specification, which can then be referred to from other documents in lieu of repeating the structure of the loop each time. This document will perform that role.

The necessary requirements for correctly performing a GSS negotiation loop are essentially all included in [RFC2743], but they are scattered in many different places. This document brings all the requirements together into one place for the convenience of implementors, even though the normative requirements remain in [RFC2743]. In a few places, this document notes additional behavior which is useful for applications but is not mandated by [RFC2743].

2. Loop Structure

The loop is begun by the appropriately named initiator, which calls GSS_Init_sec_context() with an empty (zero-length) input_token and a fixed set of input flags containing the desired attributes for the security context. The initiator should not change any of the input parameters to GSS_Init_sec_context() between calls to it during the loop, with the exception of the input_token parameter, which will contain a message from the acceptor after the initial call, and the input_context_handle, which must be the result returned in the output_context_handle of the previous call to GSS_Init_sec_context() (or GSS_C_NO_CONTEXT for the first call). (In the C bindings, there is only a single read/modify context_handle argument.) RFC 2743 only requires that the claimant_cred_handle argument remain constant over all calls in the loop, but the other non-excepted arguments should also remain fixed for reliable operation.

The following subsections will describe the various steps of the loop, without special consideration to whether a call to GSS_Init_sec_context() or GSS_Accept_sec_context() is the first such call in the loop. For the first call to each routine in the loop, the major status code from the previous call to GSS_Init_sec_context() or GSS_Accept_sec_context() should be taken as GSS_S_CONTINUE_NEEDED.

2.1. Anonymous Initiators

If the initiator is requesting anonymity by setting the anon_req_flag input to GSS_Init_sec_context(), then on non-error returns from GSS_Init_sec_context() (that is, the major status is GSS_S_COMPLETE or GSS_S_CONTINUE_NEEDED), the initiator must verify that the output value of anon_state from GSS_Init_sec_context() is true before sending the security context token to the acceptor. Failing to perform this check could cause the initiator to lose anonymity.

2.2. GSS_Init_sec_context

The initiator calls GSS_Init_sec_context(), using the input_context_handle for the current proto-security-context and its fixed set of input parameters, and the input_token received from the acceptor (if not the first iteration of the loop). The presence of a nonempty output_token and the value of the major status code are the indicators for how to proceed:

If the major status code is GSS_S_COMPLETE and the output_token is empty, then the context negotiation is fully complete and ready for use by the initiator with no further actions.

If the major status code is GSS_S_COMPLETE and the output_token is nonempty, then the initiator's portion of the security context negotiation is complete but the acceptor's is not. The initiator must send the output_token to the acceptor so that the acceptor can establish its half of the security context.

If the major status code is GSS_S_CONTINUE_NEEDED and the output_token is nonempty, the context negotiation is incomplete. The initiator must send the output_token to the acceptor and await another input_token from the acceptor.

If the major status code is GSS_S_CONTINUE_NEEDED and the output_token is empty, the mechanism has produced an output which is not compliant with [RFC2743]. However, there are some known implementations of certain mechanisms which do produce empty context negotiation tokens. For maximum interoperability, applications should be prepared to accept such tokens, and should transmit them to the acceptor if they are generated.

If the major status code is any other value, the context negotiation has failed. If the output_token is nonempty, it is an error token, and the initiator should send it to the acceptor. If the output_token is empty, then the initiator should indicate the failure to the acceptor if an appropriate channel to do so is available.

2.3. Sending from Initiator to Acceptor

The establishment of a GSS security context between initiator and acceptor requires some communication channel by which to exchange the context negotiation tokens. The nature of this channel is not specified by the GSS specification -- it could be a synchronous TCP channel, a UDP-based RPC protocol, or any other sort of channel. In many cases, the channel will be multiplexed with non-GSS application data; the application protocol must provide some means by which the GSS context tokens can be identified and passed through to the mechanism accordingly. It is in such cases where the application protocol has a means to indicate error conditions that the initiator could indicate a failure to the acceptor, as mentioned in some of the above cases conditional on "an appropriate channel to do so".

However, even the presence of a communication channel does not necessarily indicate that it is appropriate for the initiator to indicate such errors. For example, if the acceptor is a stateless or near-stateless UDP server, there is probably no need for the initiator to explicitly indicate its failure to the acceptor. Conditions such as this can be treated in individual application protocol specifications.

If a regular security context output_token is produced by the call to GSS_Init_sec_context(), the initiator must transmit this token to the acceptor over the application's communication channel. If the call to GSS_Init_sec_context() returns an error token as output_token, it is recommended that the intiator transmit this token to the acceptor over the application's communication channel.

2.4. Acceptor Sanity Checking

The acceptor's half of the negotiation loop is triggered by the receipt of a context token from the initiator. Before calling GSS_Accept_sec_context(), the acceptor may find it useful to perform some sanity checks on the state of the negotiation loop.

If the acceptor receives a context token but was not expecting such a token (for example, if the acceptor's previous call to GSS_Accept_sec_context() returned GSS_S_COMPLETE), this is probably an error condition indicating that the initiator's state is invalid. See Section 3.2 for some exceptional cases. It is likely appropriate for the acceptor to report this error condition to the acceptor via the application's communication channel.

If the acceptor is expecting a context token (e.g., if the previous call to GSS_Accept_sec_context() returned GSS_S_CONTINUE_NEEDED), but does not receive such a token within a reasonable amount of time after transmitting the previous output_token to the initiator, the acceptor should assume that the initiator's state is invalid and fail the GSS negotiation. Again, it is likely appropriate for the acceptor to report this error condition to the initiator via the application's communication channel.

[Are there other checks to perform here?]

2.5. GSS_Accept_sec_context

The GSS acceptor responds to the actions of an initiator; as such, there should always be a nonempty input_token to calls to GSS_Accept_sec_context(). The input_context_handle parameter will always be given as the output_context_handle from the previous call to GSS_Accept_sec_context() in a given negotiation loop (or GSS_C_NO_CONTEXT on the first call), but the acceptor_cred_handle and chan_bindings arguments should remain fixed over the course of a given GSS negotiation loop. [RFC2743] only requires that the acceptor_cred_handle remain fixed throughout the loop, but the chan_bindings argument should also remain fixed for reliable operation.

The GSS acceptor calls GSS_Accept_sec_context(), using the input_context_handle for the current proto-security-context and the input_token received from the initiator. The presence of a nonempty output_token and the value of the major status code are the indicators for how to proceed:

If the major status code is GSS_S_COMPLETE and the output_token is empty, then the context negotiation is fully complete and ready for use by the acceptor with no further actions.

If the major status code is GSS_S_COMPLETE and the output_token is nonempty, then the acceptor's portion of the security context negotiation is complete but the initiator's is not. The acceptor must send the output_token to the initiator so that the initiator can establish its half of the security context.

If the major status code is GSS_S_CONTINUE_NEEDED and the output_token is nonempty, the context negotiation is incomplete. The acceptor must send the output_token to the initiator and await another input_token from the initiator.

If the major status code is GSS_S_CONTINUE_NEEDED and the output_token is empty, the mechanism has produced an output which is not compliant with [RFC2743]. output which is not compliant with [RFC2743]. However, there are some known implementations of certain mechanisms which do produce empty context negotiation tokens. For maximum interoperability, applications should be prepared to accept such tokens, and should transmit them to the initiator if they are generated.

If the major status code is any other value, the context negotiation has failed. If the output_token is nonempty, it is an error token, and the acceptor should send it to the initiator. If the output_token is empty, then the acceptor should indicate the failure to the initiator if an appropriate channel to do so is available.

2.6. Sending from Acceptor to Initiator

The mechanism for sending the context token from acceptor to initiator will depend on the nature of the communication channel between the two parties. For a synchronous bidirectional channel, it can be just another piece of data sent over the link, but for a stateless UDP RPC acceptor, the token will probably end up being sent as an RPC output parameter. Application protocol specifications will need to specify the nature of this behavior.

If the application protocol has the initiator driving the application's control flow (with the acceptor just responding to actions from the initiator), it is particularly helpful for the acceptor to indicate a failure to the initiator, as mentioned in some of the above cases conditional on "an appropriate channel to do so".

If a regular security context output_token is produced by the call to GSS_Accept_sec_context(), the acceptor must transmit this token to the initiator over the application's communication channel. If the call to GSS_Accept_sec_context() returns an error token as output_token, it is recommended that the acceptor transmit this token to the initiator over the application's communication channel.

2.7. Initiator input validation

The initiator's half of the negotiation loop is triggered (after the first call) by receipt of a context token from the acceptor. Before calling GSS_Init_sec_context(), the initiator may find it useful to perform some sanity checks on the state of the negotiation loop.

If the initiator receives a context token but was not expecting such a token (for example, if the initiator's previous call to GSS_Init_sec_context() returned GSS_S_COMPLETE), this is probably an error condition indicating that the acceptor's state is invalid. See Section 3.2 for some exceptional cases. It may be appropriate for the initiator to report this error condition to the acceptor via the application's communication channel.

If the initiator is expecting a context token (that is, the previous call to GSS_Init_sec_context() returned GSS_S_CONTINUE_NEEDED), but does not receive such a token within a reasonable amount of time after transmitting the previous output_token to the acceptor, the initiator should assume that the acceptor's state is invalid and fail the GSS negotiation. Again, it may be appropriate for the initiator to report this error condition to the acceptor via the application's communication channel.

[Are there other checks to perform here?]

2.8. Continue the Loop

If the loop is in neither a success or failure condition, then the loop must continue. Control flow returns to Section 2.2.

3. After Security Context Negotiation

Once a party has completed its half of the security context and fulfilled its obligations to the other party, the context is

complete, but it is not necessarily ready and appropriate for use. (In some cases the context may be ready for use earlier than this, see Section 3.1.) In particular, the security context flags may not be appropriate for the given application's use.

The initiator specifies as part of its fixed set of inputs to GSS_Init_sec_context() values for the following booleans: deleg_req_flag, mutual_req_flag, replay_det_req_flag, sequence_req_flag, conf_req_flag, and integ_req_flag. Upon completion of security context negotiation, the initiator must verify the values of the deleg_state, mutual_state, replay_det_state, sequence_state, conf_avail, and integ_avail flags from the last call to GSS_Init_sec_context() corresponding to the requested flags. If a flag was requested but is not available, and that feature is necessary for the appplication protocol, the initiator must destroy the security context and not use the security context for application traffic.

Application protocol specifications citing this document should indicate which context flags are required for the application protocol.

The acceptor receives as output the following booleans: deleg_state, mutual_state, replay_det_state, sequence_state, anon_state, trans_state, conf_avail, and integ_avail. The acceptor must verify that any flags necessary for the application protocol are set. If a necessary flag is not set, the acceptor must destroy the security context and not use the security context for application traffic.

3.1. Using Partially Complete Security Contexts

For mechanism/flag combinations that require multiple token exchanges, an application protocol may find it desirable to begin sending application data protected with GSS per-message operations while continuing to exchange security context tokens to complete the security context negotiation. The prot_ready_state output value from GSS_Init_sec_context() and GSS_Accept_sec_context() indicates when per-message operations are avaialble.

Applications requiring confidentiality and/or integrity protection from such messages must check the value of the conf_avail and/or integ_avail output flags from GSS_Init_sec_context()/ GSS_Accept_sec_context() as well as the conf_state output of GSS_Wrap() (if GSS_Wrap() is used).

3.2. Additional Context Tokens

Under some (rare) conditions, a context token will be received by a party to a security context negotiation after that party has completed the negotiation (i.e., after GSS_Init_sec_context() or GSS_Accept_sec_context() has returned GSS_S_COMPLETE). Such tokens must be passed to GSS_Process_context_token() for processing.

The most common cause of such tokens is security context deletion tokens, emitted when the remote party called GSS_Delete_sec_context() with a non-null output_context_token parameter. With the GSS-API version 2, it is not recommended to use security context deletion tokens.

Extra security context tokens can also be emitted if the selected mechanism specifies some functionality (such as per-message confidentiality protection) as optional-to-implement, and the acceptor's implementation does not implement the optional functionality, but the functionality was requested by the initiator. In this case, the acceptor's GSS implementation is required to emit at least one context token (even when one would not otherwise be needed to complete the context negotiation), and this can result in an "extra" token.

In the rare case when an application receives an extra context token, GSS_Inquire_context() should be used after processing the extra token to re-verify that the context does support the features necessary for the application protocol. This will also indicate whether the token was a deletion token, in which case the major status will be GSS_S_NO_CONTEXT.

4. Sample Code

This section gives sample code for the GSS negotiation loop, both for a regular application and for an application where the initiator wishes to remain anonymous. Since the code for the two cases is very similar, the anonymous-specific additions are wrapped in preprocessor conditionals which may be ignored if anonymous processing is not needed.

Since the communication channel between the initiator and acceptor is a matter for individual application protocols, it is inherently unspecified at the GSS-API level, which can lead to examples that are less satisfying than may be desired. For example, the sample code in [RFC2744] uses an unspecified send_token_to_peer() routine. In the interest of concreteness, this sample code uses pipes for communication between initiator and acceptor, so that explicit read() and write() may be used.

This sample code is written in C, using the GSS-API C bindings [RFC2744]. It uses the macro GSS_ERROR() to help unpack the various sorts of information which can be stored in the major status field; supplementary information does not necessarily indicate an error. Applications written in other languages will need to exercise care that checks against the major status value are written correctly.

This sample code should be compilable as a standalone program, linked against a GSS-API library. With most implementations, in order for it to successfully run, the initiator will need to specify an explicit target name for the acceptor (which must match the credentials available to the acceptor). A skeleton for how this may be done is provided, in a disabled block of code.

This sample code assumes v2 of the GSS-API. Applications wishing to remain compatible with v1 of the GSS-API may need to perform additional checks in some locations.

4.1. GSS Application Sample Code

```
#include <unistd.h>
#include <assert.h>
#include <err.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <gssapi/gssapi.h>

/*
 * Pipes for communication between initiator and acceptor.
 * We use a very simple communication protocol, that can only ever
 * send context negotiation tokens and no other application data.
 * The framing is that we write a 32-bit unsigned integer which is
 * the byte count of the following token, followed by the token.
 */
int pipefds_itoa[2];
int pipefds_atoi[2];

/*
 * This helper is used only on buffers that we allocate ourselves (e.g.,
 * from receive_buffer()).  Buffers allocated by GSS routines must use
 * gss_release_buffer().
 */
static void
release_buffer(gss_buffer_t buf)
{
    free(buf->value);
    buf->value = NULL;
```

```
        buf->length = 0;
    }

    /*
     * Helper to send a token on the specified fd, using our simple protocol.
     * We must warnx() instead of errx() because compliant GSS applications must
     * release resources allocated by the GSS library before exiting.  (These
     * resources may be non-local to the current process.)
     */
    static int
    send_token(int fd, gss_buffer_t token)
    {
        int ret;
        OM_uint32 length;

        assert(sizeof(length) == 4);
        length = token->length;
        ret = write(fd, &length, 4);
        if (ret != 4) {
            warnx("send_token could not write length\n");
            return 1;
        }
        ret = write(fd, token->value, length);
        if (ret != length) {
            warnx("send_token could not write token\n");
            return 1;
        }
        return 0;
    }

    /*
     * Helper to receive a token on the specified fd, using our simple protocol.
     * We must warnx() instead of errx() because compliant GSS applications must
     * release resources allocated by the GSS library before exiting.  (These
     * resources may be non-local to the current process.)
     */
    static int
    receive_token(int fd, gss_buffer_t token)
    {
        int ret;
        OM_uint32 length;

        assert(sizeof(length) == 4);
        ret = read(fd, &length, 4);
        if (ret != 4) {
            warnx("receive_token could not read length, ret %u\n", length);
            return 1;
        }
```

```
       /* Do a little sanity checking. */
       if (length > 64 * 1024*1024) {
           warnx("Attempting to receive token larger than 64M\n");
           return 1;
       }
       token->value = malloc(length);
       if (token->value == NULL) {
           warnx("Could not allocate memory to receive token\n");
           return 1;
       }
       ret = read(fd, token->value, length);
       if (ret != length) {
           warnx("Could not receive token\n");
           return 1;
       }
       token->length = length;
       return 0;
   }

   static void
   do_initiator(int readfd, int writefd)
   {
       int context_established = 0;
       gss_ctx_id_t ctx = GSS_C_NO_CONTEXT;
       OM_uint32 major, minor, req_flags, ret_flags;
       gss_buffer_desc input_token, output_token;
       gss_name_t target_name = GSS_C_NO_NAME;
       OM_uint32 ret;

       memset(&input_token, 0, sizeof(input_token));
       memset(&output_token, 0, sizeof(output_token));

       /* Applications should set target_name to a real value. */
   #if 0
       gss_buffer_desc name_buf;
       name_buf.value = "<service>@<hostname.domain>";
       name_buf.length = strlen(name_buf.value);
       major = gss_import_name(&minor, &name_buf, GSS_C_NT_HOSTBASED_SERVICE,
                               &target_name);
       /* target_name must be released with gss_release_name() at cleanup. */
   #endif

       /* Mutual authentication will require a token from acceptor to
initiator,
        * and thus a second call to gss_init_sec_context(). */
       req_flags = GSS_C_MUTUAL_FLAG | GSS_C_CONF_FLAG | GSS_C_INTEG_FLAG;
   #ifdef ANONYMOUS
       req_flags |= GSS_C_ANON_FLAG;
```

```
#endif
```

```
       while (!context_established) {
           /* The initiator_cred_handle, mech_type, time_req,
input_chan_bindings,
            * actual_mech_type, and time_rec parameters are not needed in many
            * cases.  We pass GSS_C_NO_CREDENTIAL, GSS_C_NO_OID, 0, NULL, NULL,
            * and NULL for them, respectively. */
           major = gss_init_sec_context(&minor, GSS_C_NO_CREDENTIAL, &ctx,
                                        target_name, GSS_C_NO_OID, req_flags,
0,
                                        NULL, &input_token, NULL,
&output_token,
                                        &ret_flags, NULL);
           /* This memory is no longer needed. */
           release_buffer(&input_token);
   #ifdef ANONYMOUS
           /* Initiators which wish to remain anonymous must check whether
            * their request has been honored before sending each context token.
*/
           if ((ret_flags & GSS_C_ANON_FLAG) != GSS_C_ANON_FLAG) {
               warnx("Anonymous processing not available\n");
               goto cleanup;
           }
   #endif
           /* Always send a token if we are expecting another input token
            * (GSS_S_CONTINUE_NEEDED) or if it is nonempty. */
           if ((major & GSS_S_CONTINUE_NEEDED) != 0 ||
               output_token.length > 0) {
               ret = send_token(writefd, &output_token);
               if (ret != 0)
                   goto cleanup;
           }
           /* Check for errors after sending the token so that we will send
            * error tokens. */
           if (GSS_ERROR(major)) {
               warnx("gss_init_sec_context() error major 0x%x\n", major);
               goto cleanup;
           }
           /* Having sent any output_token, release the storage for it. */
           (void)gss_release_buffer(&minor, &output_token);

           if ((major & GSS_S_CONTINUE_NEEDED) != 0) {
               ret = receive_token(readfd, &input_token);
               if (ret != 0)
                   goto cleanup;
           } else if (major == GSS_S_COMPLETE) {
               context_established = 1;
           } else {
               /* This situation is forbidden by RFC 2743.  Bail out. */
```

```
            warnx("major not complete or continue-needed but not error\n");
            goto cleanup;
        }
    }   /* while(!context_established) */
```

```
        if ((ret_flags & req_flags) != req_flags) {
            warnx("Negotiated context does not support requested flags\n");
            goto cleanup;
        }
        printf("Initiator's context negotiation successful\n");
    cleanup:
        /* It is safe to call gss_release_buffer twice on the same buffer. */
        (void)gss_release_buffer(&minor, &output_token);
        /* Do not request a context deletion token; pass NULL. */
        (void)gss_delete_sec_context(&minor, &ctx, NULL);
    }

    static void
    do_acceptor(int readfd, int writefd)
    {
        int context_established = 0, ret;
        gss_ctx_id_t ctx = GSS_C_NO_CONTEXT;
        OM_uint32 major, minor, ret_flags;
        gss_buffer_desc input_token, output_token;
        gss_name_t client_name;

        memset(&input_token, 0, sizeof(input_token));
        memset(&output_token, 0, sizeof(output_token));

        context_established = 0;
        major = GSS_S_CONTINUE_NEEDED;

        while(!context_established) {
            if ((major & GSS_S_CONTINUE_NEEDED) != 0) {
                ret = receive_token(readfd, &input_token);
                if (ret != 0)
                    goto cleanup;
            } else if (major == GSS_S_COMPLETE) {
                context_established = 1;
                break;
            } else {
                /* This situation is forbidden by RFC 2743.  Bail out. */
                warnx("major not complete or continue-needed but not error\n");
                goto cleanup;
            }
            /* We can use the default behavior or do not need the returned
             * information for the parameters acceptor_cred_handle,
             * input_chan_bindings, mech_type, time_rec, and
delegated_cred_handle
             * and pass the values GSS_C_NO_CREDENTIAL, NULL, NULL, NULL, and
NULL,
             * respectively.  In some cases the src_name will not be needed, but
             * most likely it will be needed for some authorization or logging
```

```
         * functionality. */
        major = gss_accept_sec_context(&minor, &ctx, GSS_C_NO_CREDENTIAL,
```

```
                                        &input_token, NULL, &client_name,
NULL,
                                        &output_token, &ret_flags, NULL,
NULL);
           /* Release memory no longer needed. */
           release_buffer(&input_token);
           /* Always send a token if we are expecting another input token
            * (GSS_S_CONTINUE_NEEDED) or if it is nonempty. */
           if ((major & GSS_S_CONTINUE_NEEDED) != 0 ||
               output_token.length > 0) {
               ret = send_token(writefd, &output_token);
               if (ret != 0)
                   goto cleanup;
           }
           /* Check for errors after sending the token so that we will send
            * error tokens. */
           if (GSS_ERROR(major)) {
               warnx("gss_accept_sec_context() error major 0x%x\n", major);
               goto cleanup;
           }
           /* Release the output token's storage; we don't need it anymore. */
           (void)gss_release_buffer(&minor, &output_token);
       }   /* while(!context_established) */
       if ((ret_flags & GSS_C_INTEG_FLAG) != GSS_C_INTEG_FLAG) {
           warnx("Negotiated context does not support integrity\n");
           goto cleanup;
       }
       printf("Acceptor's context negotiation successful\n");
   cleanup:
       /* It is safe to call gss_release_buffer twice on the same buffer. */
       release_buffer(&input_token);
       /* Do not request a context deletion token, pass NULL. */
       (void)gss_delete_sec_context(&minor, &ctx, NULL);
       (void)gss_release_name(&minor, &client_name);
   }

   int main(int argc, char **argv)
   {
       pid_t pid;

       if (pipe(pipefds_itoa) != 0)
           err(1, "pipe failed for itoa\n");
       if (pipe(pipefds_atoi) != 0)
           err(1, "pipe failed for atoi\n");
       pid = fork();
       if (pid == 0)
           do_initiator(pipefds_atoi[0], pipefds_itoa[1]);
       else if (pid > 0)
```

```
            do_acceptor(pipefds_itoa[0], pipefds_atoi[1]);
        else
```

```
            err(1, "fork() failed\n");
        exit(0);
    }
```

5. Security Considerations

This document provides a (reasonably) concise description and example for correct construction of the GSS-API security context negotiation loop. Since everything relating to the construction and use of a GSS security context is security-related, there are security-relevant considerations throughout the document. It is useul to call out a few things in this section, though.

The GSS-API uses a request-and-check model for features. An application using the GSS-API requests that certain features (confidentiality protection for messages, or anonynimity), but such a request does not require the GSS implementation to provide the feature. The application must check the returned flags to verify whether a requested feature is present; if the feature was non-optional for the application, the application must generate an error. Phrased differently, the GSS-API will not generate an error if it is unable to satisfy the features requested by the application.

6. References

6.1. Normative References

[RFC2743] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, January 2000.

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

6.2. Informational References

[RFC4462] Hutzelman, J., Salowey, J., Galbraith, J., and V. Welch, "Generic Security Service Application Program Interface (GSS-API) Authentication and Key Exchange for the Secure Shell (SSH) Protocol", RFC 4462, May 2006.

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

[RFC3645] Kwan, S., Garg, P., Gilroy, J., Esibov, L., Westhead, J., and R. Hall, "Generic Security Service Algorithm for Secret Key Transaction Authentication for DNS (GSS-TSIG)", RFC 3645, October 2003.

[RFC5801] Josefsson, S. and N. Williams, "Using Generic Security Service Application Program Interface (GSS-API) Mechanisms in Simple Authentication and Security Layer (SASL): The GS2 Mechanism Family", RFC 5801, July 2010.

[RFC4752] Melnikov, A., "The Kerberos V5 ("GSSAPI") Simple Authentication and Security Layer (SASL) Mechanism", RFC 4752, November 2006.

[RFC2203] Eisler, M., Chiu, A., and L. Ling, "RPCSEC_GSS Protocol Specification", RFC 2203, September 1997.

Appendix A. Acknowledgements

Thanks to Nico Williams and Jeff Hutzleman for prompting me to write this document.

Author's Address

Benjamin Kaduk
MIT Kerberos Consortium

Email: kaduk@mit.edu