Network Working Group B. Kaduk
Internet-Draft MIT
Intended status: Informational December 31, 2014
Expires: July 4, 2015

Structure of the GSS Negotiation Loop
draft-ietf-kitten-gss-loop-03

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 July 4, 2015.

Copyright Notice

Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction 2
2. Application Protocol Requirements 3
3. Loop Structure 4
  3.1. Anonymous Initiators 4
  3.2. GSS_Init_sec_context 5
  3.3. Sending from Initiator to Acceptor 5
  3.4. Acceptor Sanity Checking 6
  3.5. GSS_Accept_sec_context 7
  3.6. Sending from Acceptor to Initiator 8
  3.7. Initiator input validation 8
  3.8. Continue the Loop 9
4. After Security Context Negotiation 9
  4.1. Authorization Checks 10
  4.2. Using Partially Complete Security Contexts 10
  4.3. Additional Context Tokens 10
5. Sample Code 11
  5.1. GSS Application Sample Code 12
6. Security Considerations 18
7. References 19
  7.1. Normative References 19
  7.2. Informational References 19
Appendix A. Acknowledgements 20
Author's Address 20

1. Introduction

The Generic Security Service Application Program Interface 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 each party to authenticate the other, the passing of confidential and/or integrity-protected messages between the initiator and acceptor, the generation of identical pseudo-random bit strings by both participants [RFC4401], and more.

During context establishment, security context tokens are exchanged synchronously, one at a time; the initiator sends the first context token. 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, together with its security mechanisms, specifies the format and encoding of the context tokens themselves, but the application protocol must specify the necessary framing for the application to determine what octet strings constitute GSS security context tokens and pass them into the GSS-API implementation as appropriate.

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. Application Protocol Requirements

Part of the purpose of this document is to guide the development of new application protocols using the GSS-API, as well as the development of new application software using such protocols. The following list is features which are necessary or useful in such an application protocol:

- A way to frame and identify security context negotiation tokens in the loop.

- Error tokens should generally also get special framing, as the recipient may have no other way to distinguish unexpected error context tokens from per-message tokens.

- Failing that, a way to indicate error status from one peer to the other, possibly accompanied by a descriptive string.

- A protocol may use the negotiated GSS security context for per-message operations; in such cases, the protocol will need a way to frame and identify those per-message tokens and the nature of their contents. For example, a protocol message may be accompanied by the output of GSS_GetMIC() over that message; the protocol must identify the location and size of that MIC token, and indicate that it is a MIC token and what cleartext it corresponds to.

- Applications are responsible for authorization of the authenticated peer principal names which are supplied by the GSS-API. Such names are mechanism-specific, and may come from a different portion of a federated identity scheme. Application protocols may need to supply additional information to support the authorization of access to a given resource, such as the SSHv2 "username" parameter.

3. 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() (GSS_C_NO_CONTEXT for the first call). (In the C bindings, there is only a single read/modify context_handle argument, so the same variable should be passed for each call in the loop.) 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.

3.1. 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, when 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.

3.2. 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 or absence 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 application-protocol channel to do so is available.

3.3. 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 dedicated 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 always provide some means by which the GSS context tokens can be identified (e.g., length and start location) and passed through to the mechanism accordingly. The application protocol may also include a facility for indicating errors from one party to the other, which can be used to convey errors resulting from GSS-API calls, when appropriate (such as when no error token was generated by the GSS-API implementation). Note that GSS major and minor status codes are specified by language bindings, not the abstract API; sending a major status code and optionally the display form of the two error codes may be the best that can be done in this case.

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 initiator transmit this token to the acceptor over the application's communication channel.

3.4. 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 4.3 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 (time out) 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.

3.5. 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 or absence 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]. 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 application-protocol channel to do so is available.

3.6. 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, 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 application-protocol 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.

3.7. 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 4.3 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.

3.8. Continue the Loop

If the loop is in neither a success or failure condition, then the loop must continue. Control flow returns to Section 3.2.

4. 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 particular, the security context flags may not be appropriate for the given application's use. In some cases the context may be ready for use before the negotiation is complete, see Section 4.2.

The initiator specifies as part of its fixed set of inputs to GSS_Init_sec_context() values for all defined request flag booleans, among them: deleg_req_flag, mutual_req_flag, replay_det_req_flag, sequence_req_flag, conf_req_flag, and integ_req_flag. Upon completion of the security context negotiation, the initiator must verify that the values of the deleg_state, mutual_state, replay_det_state, sequence_state, conf_avail, and integ_avail (and any additional flags added by extensions) from the last call to GSS_Init_sec_context() correspond 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 their 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, and any additional flags added by extensions to the GSS-API. 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.

4.1. Authorization Checks

The acceptor receives as one of the outputs of GSS_Accept_sec_context() the name of the initiator which has authenticated during the security context negotiation. Applications need to implement authorization checks on this received name ('client_name' in the sample code) before providing access to restricted resources. In particular, security context negotiation can be successful when the client is anonymous or is from a different identity realm than the acceptor, depending on the details of the mechanism used by the GSS-API to establish the security context. Acceptor applications can check which target name was used by the initiator, but the details are out of scope for this document. See [RFC2743] sections 2.2.6 and 1.1.5. Additional information can be available in GSS-API Naming Extensions, [RFC6680].

4.2. Using Partially Complete Security Contexts

For mechanism/flag combinations that require multiple token exchanges, the GSS-API specification [RFC2743] provides a prot_ready_state output value from GSS_Init_sec_context() and GSS_Accept_sec_context(), which indicates when per-message operations are available. However, many mechanism implementations do not provide this functionality, and the analysis of the security consequences of its use is rather complicated, so it is not expected to be useful in most application protocols.

In particular, mutual authentication, replay protection, and other services (if requested) are services which will be active when GSS_S_COMPLETE is returned, but which are not necessarily active before the security context is fully established.

4.3. Additional Context Tokens

Under some 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. It may not always be necessary for a mechanism implementation to generate an error token on the initiator side, or for an initiator application to transmit that token to the acceptor; such decisions are out of scope for this document. Both peers should always be prepared to process such tokens, and application protocols should provide means by which they can be transmitted.

Such tokens can be security context deletion tokens, emitted when the remote party called GSS_Delete_sec_context() with a non-null

output_context_token parameter, or error tokens, emitted when the remote party experiences an error processing the last token in a security context negotiation exchange. Errors experienced when processing tokens earlier in the negotiation would be transmitted as normal security context tokens and processed by GSS_Init_sec_context() or GSS_Accept_sec_context(), as appropriate. With the GSS-API version 2, it is not recommended to use security context deletion tokens, so error tokens are expected to be the most common form of additional context token for new application protocols.

GSS_Process_context_token() may indicate an error in its major_status field if an error is encountered locally during token processing, or to indicate that an error was encountered on the peer and conveyed in an error token. [RFC2743E4151] Regardless of the major_status output of GSS_Process_context_token(), GSS_Inquire_context() should be used after processing the extra token, to query the status of the security context and whether it can supply the features necessary for the application protocol.

At present, all tokens which should be handled by GSS_Process_context_token() will lead to the security context being effectively unusable. Future extensions to the GSS-API may allow for applications to continue to function after a call to GSS_Process_context_token(), and it is expected that the outputs of GSS_Inquire_context() will indicate whether it is safe to do so. However, since there are no such extensions at present (error tokens and deletion tokens both result in the security context being essentially unusable), there is no guidance to give to applications regarding this possibility at this time.

Even if GSS_Process_context_token() processes an error or deletion token which renders the context essentially unusable, the resources associated with the context must eventually be freed with a call to GSS_Delete_sec_context(), just as would be needed if GSS_Init_sec_context() or GSS_Accept_sec_context() had returned an error while processing an input context token and the input_context_handle was not GSS_C_NO_CONTEXT. RFC 2743 has some text which is slightly ambiguous in this regard, but the best practice is to always call GSS_Delete_sec_context().

5. 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 a

conditional check; that check and the conditional code 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. Fully correct and general code to frame and transmit tokens requires a substantial amount of error checking and would detract from the core purpose of this document, so we only present the function signature for one example of what such functions might be, and leave some comments in the otherwise-empty function bodies.

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. In addition to supplying implementations for the token transmission/receipt routines, in order for the program to successfully run when linked against most GSS-API libraries, 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, using a dummy name.

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.

5.1. GSS Application Sample Code

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

/*
 * This helper is used only on buffers that we allocate ourselves (e.g.,
 * from receive_token()).  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.
 *
 * If errors are encountered, this routine must not directly cause
 * termination of the process, because compliant GSS applications
 * must release resources allocated by the GSS library before
 * exiting.
 */
static int
send_token(int fd, gss_buffer_t token)
{
    /*
     * Supply token framing and transmission code here.
     *
     * It is advisable for the application protocol to specify the
     * length of the token being transmitted, unless the underlying
     * transit does so implicitly.
     *
     * In addition to checking for error returns from whichever
     * syscall(s) are used to send data, applications should have
     * a loop to handle EINTR returns.
     */
    return 1;
}

/*
 * Helper to receive a token on the specified fd.
 *
 * If errors are encountered, this routine must not directly cause
 * termination of the process, because compliant GSS applications
 * must release resources allocated by the GSS library before
 * exiting.
 */
static int
receive_token(int fd, gss_buffer_t token)
{
    /*
     * Supply token framing and transmission code here.
     *
     * In addition to checking for error returns from whichever
     * syscall(s) are used to receive data, applications should have
```

```
     * a loop to handle EINTR returns.
     *
     * This routine is assumed to allocate memory for the local copy
     * of the received token, which must be freed with release_buffer().
     */
    return 1;
}

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

    /* Applications should set target_name to a real value. */
    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);
    if (GSS_ERROR(major)) {
        warnx(1, "Could not import name\n");
        goto cleanup;
    }

    /* 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;
    if (anon)
        req_flags |= GSS_C_ANON_FLAG;

    while (!initiator_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 was allocated by receive_token() and is no longer
         * needed.  Free it now to avoid leaks if the loop continues. */
```

```
        release_buffer(&input_token);
        if (anon) {
            /* Initiators which wish to remain anonymous must check
             * whether their request has been honored before sending
             * each token. */
            if (!(ret_flags & GSS_C_ANON_FLAG)) {
                warnx("Anonymous requested but not available\n");
                goto cleanup;
            }
        }
        /* Always send a token if we are expecting another input token
         * (GSS_S_CONTINUE_NEEDED is set) or if it is nonempty. */
        if ((major & GSS_S_CONTINUE_NEEDED) ||
            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;
        }
        /* Free the output token's storage; we don't need it anymore.
         * gss_release_buffer() is safe to call on the output buffer
         * from gss_int_sec_context(), even if there is no storage
         * associated with that buffer. */
        (void)gss_release_buffer(&minor, &output_token);

        if (major & GSS_S_CONTINUE_NEEDED) {
            ret = receive_token(readfd, &input_token);
            if (ret != 0)
                goto cleanup;
        } else if (major == GSS_S_COMPLETE) {
            initiator_established = 1;
        } else {
            /* This situation is forbidden by RFC 2743.  Bail out. */
            warnx("major not complete or continue but not error\n");
            goto cleanup;
        }
    }   /* while (!initiator_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:
```

```
    /* We are required to release storage for nonzero-length output
     * tokens.  gss_release_buffer() zeros the length, so we are
     * will not attempt to release the same buffer twice. */
    if (output_token.length > 0)
        (void)gss_release_buffer(&minor, &output_token);
    /* Do not request a context deletion token; pass NULL. */
    (void)gss_delete_sec_context(&minor, &ctx, NULL);
    (void)gss_release_name(&minor, &target_name);
}

/*
 * Perform authorization checks on the initiator's GSS name object.
 *
 * Returns 0 on success (the initiator is authorized) and nonzero
 * when the initiator is not authorized.
 */
static int
check_authz(gss_name_t client_name)
{
    /*
     * Supply authorization checking code here.
     *
     * Options include bitwise comparison of the exported name against
     * a local database, and introspection against name attributes.
     */
    return 0;
}

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

    major = GSS_S_CONTINUE_NEEDED;

    while (!acceptor_established) {
        if (major & GSS_S_CONTINUE_NEEDED) {
            ret = receive_token(readfd, &input_token);
            if (ret != 0)
                goto cleanup;
        } else if (major == GSS_S_COMPLETE) {
            acceptor_established = 1;
            break;
```

```
        } else {
            /* This situation is forbidden by RFC 2743.  Bail out. */
            warnx("major not complete or continue 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);
        /* This was allocated by receive_token() and is no longer
         * needed.  Free it now to avoid leaks if the loop continues. */
        release_buffer(&input_token);
        /* Always send a token if we are expecting another input token
         * (GSS_S_CONTINUE_NEEDED is set) or if it is nonempty. */
        if ((major & GSS_S_CONTINUE_NEEDED) ||
            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;
        }
        /* Free the output token's storage; we don't need it anymore.
         * gss_release_buffer() is safe to call on the output buffer
         * from gss_accept_sec_context(), even if there is no storage
         * associated with that buffer. */
        (void)gss_release_buffer(&minor, &output_token);
    }   /* while (!acceptor_established) */
    if (!(ret_flags & GSS_C_INTEG_FLAG)) {
        warnx("Negotiated context does not support integrity\n");
        goto cleanup;
    }
    printf("Acceptor's context negotiation successful\n");
    ret = check_authz(client_name);
```

```
    if (ret != 0)
        printf("Client is not authorized; rejecting access\n");
cleanup:
    release_buffer(&input_token);
    /* We are required to release storage for nonzero-length output
     * tokens.  gss_release_buffer() zeros the length, so we are
     * will not attempt to release the same buffer twice. */
    if (output_token.length > 0)
        (void)gss_release_buffer(&minor, &output_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(void)
{
    pid_t pid;
    int fd1 = -1, fd2 = -1;

    /* Create fds for reading/writing here. */
    pid = fork();
    if (pid == 0)
        do_initiator(fd1, fd2, 0);
    else if (pid > 0)
        do_acceptor(fd2, fd1);
    else
        err(1, "fork() failed\n");
    exit(0);
}
```

6. 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 useful 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 certain features (confidentiality protection for messages, or anonymity), but such a request does not require the GSS implementation to provide that 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.

In many cases it is convenient for GSS acceptors to accept security contexts using multiple acceptor names (such as by using the default credential set, as happens when GSS_C_NO_CREDENTIAL is passed to GSS_Accept_sec_context()). This allows acceptors to use any credentials to which it has access for accepting security contexts, which may not be the desired behavior for a given application. (For example, sshd may only wish to accept only using GSS_C_NT_HOSTBASED credentials of the form host@<hostname>, and not nfs@<hostname>.) Acceptor applications can check which target name was used by the initiator, but the details are out of scope for this document. See [RFC2743] sections 2.2.6 and 1.1.5.

The C sample code uses the macro GSS_ERROR() to assess the return value of gss_init_sec_context() and gss_accept_sec_context(). This is done to indicate where checks are needed in writing code for other languages and what the nature of those checks might be. The C code could be made simpler by omitting that macro. In applications expecting to receive protected octet streams, this macro should not be used on the result of per-message operations, as it omits checking for supplementary status values such as GSS_S_DUPLICATE_TOKEN, GSS_S_OLD_TOKEN, etc.. Use of the GSS_ERROR() macro on the results of GSS-API per-message operations has resulted in security vulnerabilities in existing software!

The security considerations from RFCs 2743 and 2744 remain applicable to consumers of this document.

7. References

7.1. 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.

7.2. 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.

[RFC6680] Williams, N., Johansson, L., Hartman, S., and S. Josefsson, "Generic Security Service Application Programming Interface (GSS-API) Naming Extensions", RFC 6680, August 2012.

[RFC2743E4151]
Williams, N., "RFC 2743 Errata 4151", November 2014.

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