Network Working Group
Internet-Draft

M. Eisler
Network Appliance, Inc.
N. Williams
Sun Microsystems, Inc.
May 2003

The Channel Conjunction Mechanism (CCM) for GSS
draft-ietf-nfsv4-ccm-01.txt

Status of this Memo

ABSTRACT

This document describes a suite of new mechanisms under the GSS [RFC2743]. Some protocols, such as RPCSEC_GSS [RFC2203], use GSS to authenticate every message transfer, thereby incurring significant overhead due to the costs of cryptographic computation. While hardware-based cryptographic accelerators can mitigate such overhead, it is more likely that acceleration will be available for lower layer protocols, such as IPsec [RFC2401] than for upper layer protocols like RPCSEC_GSS. CCM can be used as a way to allow GSS mechanism-independent upper layer protocols to leverage the data stream protections of lower layer protocols, without the inconvenience of modifying the upper layer protocol to do so.

TABLE OF CONTENTS

1. Conventions Used in this Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Introduction

The GSS framework provides a general means for authenticating clients and servers, as well as providing a general means for encrypting and integrity protecting data exchanged during a session. GSS specifies formats for a set of tokens for authentication, integrity, and privacy. The formats consist of a mechanism independent form, and a mechanism dependent form. An example of a set of mechanism dependent forms is the Kerberos V5 mechanism definition [RFC1964].

It is possible for a protocol to use GSS for one time authentication, or for per message authentication. An example of the former is DAFS [DAFS]. An example of the latter is RPCSEC_GSS. Obviously, it is more secure to authenticate each message. On the other hand, it is also more expensive. However, suppose the data stream of the upper layer protocol (the layer using GSS) is protected at a lower layer protocol from tampering, such as via a cryptographic checksum. If so, it may not be necessary to additionally authenticate each message of the upper layer protocol. Instead, it may suffice to use GSS to authenticate at the beginning of the upper layer protocol's session.

To take advantage of one time authentication, existing consumers of GSS that authenticate exclusively on each message have to change. One way to change is to modify the protocol that is using GSS. This has disadvantages including, introducing a protocol incompatibility, and effectively introducing another authentication paradigm. Another way to change, is the basis of the proposal in this document: the Channel Conjunction Mechanism (CCM). CCM allows a GSS initiator and target to conjunct (bind) a secure session (or channel) at one protocol layer with (e.g. IPsec) a security context of a non-CCM GSS

mechanism. Since CCM is yet another mechanism under the GSS, the

effect is that there are no modifications to the protocol the GSS consumer is using.

3. Overview

CCM is a "wrapper" mechanism over the set of all other GSS mechanisms. When CCM creates a context, it invokes an underlying mechanism to create a child context. CCM determines the underlying mechanism by examining the mechanism object identifier (OID) that it is called with. The prefix will always be the OID of CCM, and the suffix will be the OID of the underlying mechanism. The context initiation and acceptance entry points of CCM wrap the resulting the context tokens with a CCM header.

3.1. Example Application of CCM

Let us use RPCSEC_GSS and NFSv4 [RFC3530] as our example. Basic understanding of the RPCSEC_GSS protocol is assumed. If an NFSv4 client uses the wrong security mechanism, the server returns the NFS4ERR_WRONGSEC error. The client can then use NFSv4's SECINFO operation to ask the server which GSS mechanism to use.

Let us say the client and server are using Kerberos V5 [RFC1964] to secure the traffic. Suppose the TCP connection NFSv4 uses is secured and encrypted with IPsec. It is therefore not necessary for NFSv4/RPCSEC_GSS to use integrity or privacy. Fortunately, RPCSEC_GSS has an authentication mode, whereby only the header of each remote procedure call and response is integrity protected. So, this minimizes the overhead somewhat, but there is still the cost of the headers being checksummed. Since IPsec is protecting the connection, incurring even that minimal per remote procedure call overhead may not be necessary.

Enter CCM. The server detects that the connection is protected with IPsec. Via SECINFO, the client is informed that it should use CCM/Kerberos V5. Via the RPCSEC_GSS protocol, the server authenticates the end-user on the client with Kerberos V5. The context tokens exchanged over RPCSEC_GSS are wrapped inside CCM tokens.

3.2. A Suite of CCM Mechanisms

CCM consists of a suite of GSS mechanisms. CCM-NULL, CCM-ADDR, and CCM-KEY bind a GSS mechanism context to a secure channel via GSS channel bindings (see section 1.1.6 of RFC2743). As noted in RFC2743, the purpose of channel bindings are to limit the scope within which an intercepted GSS context token can be used by an attacker. CCM-KEY requires the use of channel bindings that are

derived from the secure channel's encryption keys. CCM-ADDR requires the use of channel bindings that are derived from network addresses associated with the secure channel. For environments where it is not feasible to use key-based channel bindings (e.g., the programming interfaces to get them are not available) or address-based channel bindings (e.g., the secure channel may be constructed over a path that requires the use of Network Address Translation), CCM-NULL is also defined. CCM-NULL requires the use of null channel bindings.

As discussed later in this document CCM-MIC exists for the purpose of optimizing the use of CCM.

Implementations that claim compliance with this document are REQUIRED to implement CCM-KEY and CCM-MIC. CCM-NULL and CCM-ADDR implementation are OPTIONAL. Specifications that make normative references to CCM are free to mandate any subset of the suite CCM mechanisms.

Because the GSS channel bindings to IPsec [RFC2401, RFC2409, IKEv2] have not been previously defined, and to ensure the usefulness of CCM, they are defined in this document.

Also, the SPKM (1, 2 and 3) [RFC2025, RFC2847] mechanism is not clear on how channel bindings work with SPKM; a simple clarification is provided.

CCM-MIC is intended to reduce the instances of full GSS context establishment to a per- {initiator principal, target} tuple. CCM-MIC is used to establish a new context by proving that the initiator and target both have a previously established, unexpired GSS context; the proof is accomplished by exchanging MICs made with the previously established GSS context. The CCM-MIC context creation entry points utilize the CCM_REAL_QOP (discussed later Overview section) in the value to generate and verify the MICs. The type of channel bindings used when initiating CCM-MIC contexts MUST match that used when creating the previously established context.

3.3. QOPs

The CCM mechanisms provide two QOPs: the default QOP (0) that amounts to no protection, and a QOP (CCM_REAL_QOP, defined as value 1) that maps to the default QOP of the underlying GSS mechanism. The MIC tokens for CCM are zero length values. When qop_req is 0, the wrap output tokens for CCM are equal to the input tokens.

> [XXX - We assume that applications can cope with zero length MICs. We propose that implementations try and find out. We may revisit this by requiring a small (8-32 bits) MIC token. However, given that the C bindings of GSS allocates the MIC on

the heap, this could introduce an unnecessary and expensive allocation, we suggest applications be fixed to deal with zero length tokens.]

4. Token Formats

This section discusses the protocol visible tokens that GSS consumers exchange when using CCM.

4.1. Mechanism Object Identifier

There are two classes of Mechanism object identifiers (OIDs) for CCM. The first class consists of the channel binding specific OIDs, and will be referred to as the CCM-BIND mechanisms:

{iso(1)identified-organization(3)dod(6)internet(1)security(5) mechanisms(5)ccm-family(TBD1)ccm-bind(1)ccm-null(1)}

{iso(1)identified-organization(3)dod(6)internet(1)security(5) mechanisms(5)ccm-family(TBD1)ccm-bind(1)ccm-addr(2)}

{iso(1)identified-organization(3)dod(6)internet(1)security(5) mechanisms(5)ccm-family(TBD1)ccm-bind(1)ccm-key(3)}

The above three object identifiers are not complete mechanism OIDs. Complete CCM mechanism OIDs MUST consist of one of the above OIDs as prefix, followed by a real mechanism OID, such as that of Kerberos V5 as defined in [RFC1964]. The second class consists of a single OID for the CCM-MIC mechanism.

{iso(1)identified-organization(3)dod(6)internet(1)security(5) mechanisms(5)ccm-family(TBD1)ccm-mic(2)}

The CCM-MIC OID is a complete mechanism OIDs, and is not a prefix.

GSS defines the generic part of a token in ASN.1 encoding. GSS does not require ASN.1 for the mechanism specific part of a token.

4.2. Tokens for the CCM-BIND mechanisms

4.3. Context Establishment Tokens for CCM-BIND Mechanisms

The CCM-BIND context establishment tokens are simple wrappers around a real GSS mechanism's tokens. The CCM-BIND mechanisms use the same number context token exchanges as required by they underlying real mechanism.

4.3.1. Initial Context Token for CCM-BIND

GSS requires that the initial context token from the initiator to the target use the format as described in section 3.1 of RFC2743. The format consists of a mechanism independent prefix, and a mechanism dependent suffix. The mechanism independent token includes the MechType field. The MechType MUST be equal to the OID of CCM-NULL, CCM-ADDR, or CCM-KEY. The mechanism dependent portion of the Initial Context Token is always equal to the full InitialContextToken as returned by the underlying real mechanism. This will include yet another MechType, which will have the underlying mechanism's OID.

4.3.2. Subsequent Context Tokens for CCM-BIND

A subsequent context token can be any subsequent context token from the initiator context initialization entry point, or any response context from the target's context acceptance entry point. The GSS specification [RFC2743] does not prescribe any format.

4.3.2.1. Subsequent Initiator Context Initialization Token for CCM-BIND

A SubsequentContextToken for a CCM-BIND mechanism is equal to that returned by the initiator's context initialization routine of the underlying real mechanism.

4.3.2.2. Response Token for CCM-BIND

The response token for a CCM-BIND mechanism is equal to that returned by the target's context acceptance routine of the underlying real mechanism.

4.4. MIC Token for CCM-BIND

This token corresponds to the PerMsgToken type as defined in section 3.1 of RFC2743. When the qop_req is the default QOP (0), then the PerMsgToken is a quantity zero bits in length. A programming API that calls GSS_GetMIC() with the default QOP will thus produce an octet string of zero length.

When the qop_req is CCM_REAL_QOP (1), then PerMsgToken is whatever the underlying real mechanism returns from GSS_GetMIC() when passed the default QOP value (0).

4.5. Wrap Token for CCM-BIND

This token corresponds to the SealedMessage type as defined in section 3.1 of RFC2743. When the qop_req is the default QOP (0), then the SealedMessage token is equal to the unmodified input to GSS_Wrap().

When the qop_req is CCM_REAL_QOP (1), then SealedMessage is whatever the underlying real mechanism returns from GSS_Wrap(), when passed the default QOP value (0).

4.6. Other Tokens for CCM-BIND

All other tokens are what the real underlying mechanism returns as a token.

4.7. Tokens for CCM-MIC

4.8. Context Establishment Tokens for CCM-MIC

4.8.1. Initial Context Token for CCM-MIC

The initial context token from the initiator to the target uses the format as described in section 3.1 of RFC2743. The format consists of a mechanism independent prefix, and a mechanism dependent suffix. The mechanism independent token includes the MechType field. The MechType MUST be equal to the OID of CCM-MIC. RFC2743 refers to the mechanism dependent token as the innerContextToken. This is the CCM-MIC specific token and is XDR [RFC1832] encoded as follows, using XDR description language:

```
typedef struct {
        unsigned int ctx_sh_number;
        unsigned int rand;
} CCM_nonce_t;

typedef struct {
        CCM_nonce_t nonce;
        opaque gss_targ_ctx[20];
        opaque chan_bindings<>;
} CCM_MIC_unwrapped_init_token_t;

/*
 * The result of CCM_MIC_unwrapped_init_token_t after
 * Invoking GSS_GetMIC() on it.  qop_req is CCM_REAL_QOP, and
 * conf_flag is FALSE.
 */
typedef opaque CCM_MIC_wrapped_init_token_t<>;
```

Once an initiator has established an initial CCM context with a target via a CCM-BIND mechanism, the additional contexts can be established via the CCM-MIC mechanism. The disadvantage of re-establishing additional contexts via the CCM-BIND route is that the

underlying mechanism context set up must be repeated, which can be expensive. Whereas, the CCM-MIC mechanism route merely requires that the first CCM context's underlying mechanism context be available to produce an integrity checksum. The initial context token for CCM-MIC is computed as follows.

* The gss_targ_ctx is computed as the SHA-1 checksum of the concatenation of SHA-1 [FIPS] checksums of the context tokens exchanged by the CCM-BIND mechanism in the order in which they were processed. For example, the context handle identifier for a CCM-KEY context exchange over a Kerberos V5 context exchange would be: SHA-1({ SHA-1(CCM-KEY's initiator's token), SHA-1(CCM-KEY's target's token)) }. Since the SHA-1 standard mandates a 160 bit output, (20 octets), gss_targ_ctx is a fixed length, 20 octet string.

* The subfield nonce.rand is set a random or pseudo random value. It is provided so as to ensure more variability of the the mic that GSS will calculate when CCM_MIC_unwrapped_init_token_t is GSS_Wrap()ed into CCM_MIC_wrapped_init_token_t.

* The subfield nonce.ctx_sh_number is the identifier of the CCM-MIC context relative to the CCM-BIND context (as identified by gss_targ_ctx) that the initiator is assigning. The value for ctx_sh_number is selected by the initiator such that it is larger than any previous ctx_sh_number for the given gss_targ_ctx. This way, the target need only keep track of the largest ctx_sh_number received. Once ctx_sh_number has reached the maximum value for an unsigned 32 bit integer, the given gss_targ_ctx can no longer be used.

* Once the above fields are calculated, GSS_Wrap() is performed on the CCM_MIC_unwrapped_init_token_t value, to produce a CCM_MIC_wrapped_init_token_t value that becomes the initial context token to send to the target.

4.8.2. Subsequent Context Tokens for CCM-MIC

A subsequent context token can be any subsequent context token from the initiator context initialization entry point, or any response context from the target's context acceptance entry point. The GSS specification [RFC2743] does not prescribe any format.

4.8.2.1. Subsequent Initiator Context Initialization Token for CCM-MIC

As CCM-MIC has only one round trip for context token exchange, there are no subsequent initiator context tokens.

4.8.2.2. Response Token for CCM-MIC

The CCM response token, in XDR encoding is:

```
typedef enum {
        CCM_OK = 0,

        /*
         * gss_targ_ctx was malformed.
         */
        CCM_ERR_HANDLE_MALFORMED = 1,

        /*
         * GSS context corresponding to gss_targ_ctx expired.
         */

        CCM_ERR_HANDLE_EXPIRED = 2,

        /*
         * gss_targ_ctx was not found.
         */
        CCM_ERR_HANDLE_NOT_FOUND = 3,

        /*
         * The ctx_sh_number has already been received
         * by the target.  Or the maximum ctx_sh_number has
         * been previously received.
         */
        CCM_ERR_TKN_REPLAY = 4,

        /*
         * Channel binding type mismatch between CCM-BIND context
         * and the CCM-MIC initial context.
         */
        CCM_ERR_CHAN_MISMATCH = 5,

        /*
         * The GSS_Unwrap() failed on initial context token
         */
        CCM_ERR_TKN_UNWRAP = 6,

        /*
         * The GSS_GetMIC() called failed on the target().
         */

        CCM_ERR_TKN_GET_MIC = 7,

        /*
         * The GSS_Wrap() failed on the initiator.  Not reported
```

```
                * by target.
                */

               CCM_ERR_TKN_WRAP = 8,

               /*
                * The GSS_VerifyMIC() failed on the initiator.  Not
                * reported by target.
                */

               CCM_ERR_TKN_VER_MIC = 9

       } CCM_MIC_status_t;

       /*
        * GSS errors returned by the underlying mechanism
        */
       typedef struct {
               unsigned int gss_major;
               unsigned int gss_minor;
       } CCM_MIC_real_gss_err_t;

       /*
        * The response context token for CCM-MIC.
        */
       typedef union switch (CCM_MIC_status status) {
               case CCM_OK:
                       opaque mic_init_tkn<>;
               case CCM_ERR_TKN_UNWRAP:
               case CCM_ERR_TKN_GET_MIC:
                       CCM_real_gss_err_t gss_err;
               default:
                       void;
       } CCM_MIC_resp_t;
```

If a value of the status field is CCM_OK, then the CCM-MIC context has been established on the target. The field mic_init_tkn is equal to the output of GSS_GetMIC() (qop_req is CCM_REAL_QOP (1)) on the entire and original token that came from the initiator. In other words, the input_token value to GSS_Accept_sec_context(). This is necessary because the inner token from the initiator is wrapped with GSS_Wrap(), and thus contains a MIC. If we performed GSS_GetMIC() on the unwrapped inner token, then for some underlying mechanisms, we would end up with a mic_init_tkn in the response token equal to what was embedded in the request token.

If the status field is CCM_ERR_TKN_UNWRAP or CCM_ERR_TKN_GET_MIC, then gss_err.gss_major and gss_err.minor are set to the major and

minor GSS statuses as returned by GSS_Unwrap() or GSS_GetMIC(). The values for the gss_major field are as defined in [RFC2744]. The values for the gss_minor field are both mechanism dependent and mechanism implemented dependent. They are nonetheless potentially useful as debugging aids.

4.9. MIC Token for CCM-MIC

The MIC token for CCM-MIC is the same as the MIC token for CCM-BIND.

4.10. Wrap Token for CCM-MIC

The wrap token for CCM-MIC is the same as the wrap token for CCM-BIND.

4.11. Context Deletion Token

The context deletion token for CCM-MIC is a zero length token.

4.12. Exported Context Token

The Exported context token for CCM-MIC is implementation defined.

4.13. Other Tokens for CCM-MIC

All other tokens are the same as corresponding tokens for CCM-BIND.

5. GSS Channel Bindings for Common Secure Channel Protocols

For CCM-KEY to be useful and secure, CCM-KEY MUST be used in conjunction with channel bindings to bind GSS authentication at the application layer to a lower layer in the network that provides cryptographic session protection.

To date only network address type channel bindings have been defined for GSS [RFC2743]. But the GSS also allows for channel bindings of "transformations of encryption keys" [RFC2743]. The actual generic representation of channel bindings is defined in the C-Bindings of the GSS-API [RFC2744].

Modern secure transports generally define some quantity or quantities which are either derived from the session keys (or from key exchange material) or which are securely exchanged in such a way that both peers of any one connection or association can arrive at the same derived quantities, while a man-in-the-middle cannot make these quantities match for both peers. Signatures of these quantities can be exchanged to prove that there is no man-in-the-middle (because a man-in-the-middle cannot cause them to be the same for both peers). These quantities correspond to what the GSS terms "transformations of

encryption keys" that are referred to in [RFC2743].

Where a secure transport clearly defines a session identifier securely derived from session keys or key exchange material, that identifier MUST be used as the GSS channel bindings data when CCM-BIND is used to bind GSS to that transport.

This section defines four forms of "transformations of encryption keys," one for IKEv1, one for IKEv2, one for SSHv2 and one for TLS. All four forms are to be used as the value of the "application_data" field of the gss_channel_bindings_struct type defined in [RFC2744].

5.1. GSS Channel Bindings for IKEv1

IKEv1 does not define a single value which can be used -- by both the IPsec initiator and responder of an IPsec SA -- to identify a given SA. IKEv1 does, however, define public values derived from the IKEv1 key exchange: 'HASH_I' and 'HASH_R'.

For IKEv1, the GSS channel bindings data to use with CCM-KEY consists of the concatenation of HASH_I and HASH_R octet string values, in that order, from the underlying IPsec session being bound to [IKEv1].

5.2. GSS Channel Bindings for IKEv2

IKEv2 peers assign and exchange 8-octet "Security Parameters Index" (SPI) values, such that a pair of SPIs suffices to uniquely identify a given IPsec security association.

For IKEv2 the GSS channel bindings data to use with CCM-KEY is simply the concatenation of the SPIi and SPIr values, in that order, which identify the IPsec SA being bound to.

5.3. GSS Channel Bindings for SSHv2

SSHv2 defines a session ID derived from the initial key exchange of an SSHv2 connection; this value is not secret and is the same for both the client and the server for any given connection.

For SSHv2 the GSS channel bindings data for use with CCM-KEY consists of the SSHv2 session ID.

5.4. GSS Channel Bindings for TLS

XXX - This section is To Be Defined.

6. Use of Channel Bindings with CCM-BIND and SPKM

Whereas the Kerberos V5 mechanism specification [RFC1964] is quite

detailed with respect to the use of GSS channel bindings, the same is not true for SPKM, which merely provides a field named "channelId" for passing channel bindings data, as octet strings, from initiators to acceptors. No interpretation is given in RFC2025 for the value of the channelId field. Therefore SPKM requires some clarification to be usable with channel bindings and CCM-KEY: The channelId field of SPKM Context-Data ASN.1 structure MUST be set to the checksum of the channel bindings data that is defined for the Kerberos V5 mechanism [RFC1964], using SHA-1 instead of MD5 as the hash algorithm.

[Note: This checksum can be computed independently of the GSS language bindings used by the application, even though RFC1964 references the C-Bindings of the GSS-API [RFC2744] in the construction of this checksum (read the RFC1964 text carefully).]

7. CCM-KEY and Anonymous IPsec

For sites that do not use IPsec, but use Kerberos V5, SPKM, or LIPKEY, deploying IPsec, a PKI infrastructure and certificates for use with IKE may prove quite difficult to deploy just for secure application (e.g., NFS) performance improvements. Such sites could avoid the need to deploy a PKI and certificates to all clients and server by using "anonymous IPsec" for the application (e.g., NFS with/ RPCSEC_GSS) and CCM-KEY.

Though there is no such thing as "anonymous IPsec," the effect can be achieved by using self-signed certificates.

By using anonymous IPsec with the application and CCM-KEY, the full benefit of offloading session cryptography from upper layer protocol layer to the IP layer can be had without having to deploy an authentication infrastructure for IPsec.

8. Other Protocol Issues for CCM

CCM-BIND is a trivial mechanism, and normally will return the same major status code as the underlying real mechanism, including GSS_S_COMPLETE as returned by GSS_Init_sec_context(). However, the first time GSS_Init_sec_context is called on a CCM-BIND mechanism, if the underlying real mechanism returns GSS_S_COMPLETE, CCM-BIND's GSS_Init_sec_context() entry point MUST return GSS_S_CONTINUE_NEEDED to the caller. This way, the initiator will receive another context token from the target, even if the underlying real mechanism context set up is done. The CCM-BIND initiator will need to record state that indicates that the underlying mechanism has reached a completely established state (and so is uninterested in any token the target returns). This way, the initiator can process every token produced by the target's GSS_Accept_sec_context() routine and so calculate

gss_targ_ctx value that matches that of the target.

9. Implementation Issues

The "over the wire" aspects of CCM have been completely specified. However, GSS is usually implemented as an Application Programming Interface (the GSS-API), and security mechanisms are often implemented as modules that are plugged into the GSS-API. It is useful to discuss implementation issues and workable resolutions. The reader is cautioned that the authors have not implemented CCM, so what follows is at best a series of educated guesses.

9.1. Management of gss_targ_ctx

The gss_targ_ctx value is computed by the initiator and target based on SHA-1 computations of the CCM-BIND context tokens. There is a space/time trade off between the initiator and target storing the sequence of context tokens until needed by CCM-BIND, versus computing the SHA-1 checksums and then disposing of the context tokens when CCM-BIND no longer needs them. If it is likely there will be CCM-MIC contexts created for the CCM-BIND context, and if the sequence of context tokens requires more space than a 20 octet SHA-1 value, then the tradeoff is obvious.

Since the bit space of all possible sequences of CCM-BIND context tokens is larger than the 160 bit space of possible SHA-1 checksums, in theory two or more different CCM-BIND contexts will produce produce the same SHA-1 context, and thus for CCM-MIC context initiation, there will be ambiguity as to which CCM-BIND context the initiator is binding to. The target can resolve this ambiguity by attempting to unwrap the inner context token from the CCM-MIC initiator for each matching CCM-BIND context. In theory no more than one GSS_Unwrap() attempt for each matching CCM-BIND context will succeed. If multiple succeed, then clearly the underlying mechanism is doing poor job at generating "unique" session keys. CCM implementations that detect this SHOULD log it so that the problem in the underlying mechanism can be discovered and fixed.

9.2. CCM-BIND Versus CCM-MIC

The first time a CCM context is needed between an principal on the initiator and a principal on the target, the initiator has no choice but to create an underlying mechanism context via a CCM-BIND context token exchange. Once that is done, subsequent CCM contexts between the initiator and target can be created via CCM-MIC. CCM-MIC context establishment is better because no more than one round trip is necessary to establish a CCM context, and because the overhead of the establishing a real, underlying mechanism context is avoided.

9.3. Initiating CCM-MIC Contexts

The key issue is how to associate an CCM-BIND established security context with a new CCM-MIC context, There no existing interfaces defined in the GSS-API for associating one GSS context with another. This then is the key issue for implementations of CCM-MIC.

We will assume that GSS-API implementation is in the C programming language and therefore the GSS-API C bindings [RFC2744] are being used. The CCM mechanism implementation will have a table that maps gss_targ_ctx values to gss_ctx_id_t values (see section 5.19 of [RFC2744]). The latter are GSS-API context handles as returned by gss_init_sec_context(). The former are the context handles as returned in a response token from the CCM target. In addition, each CCM context has a reference to its underlying mechanism context.

Let us suppose the application decides it will use CCM-MIC. CCM-MIC has a well known mechanism OID which the application can check for. The point where the initiator calls GSS_Init_sec_context(), is a logical place to associate an existing CCM-BIND context with a new CCM-MIC context. Here is where special CCM handling is necessary in order to associate a security context with a CCM context. We discuss several approaches.

1. The first approach is for the CCM-MIC's GSS_Init_sec_context() entry point to pass as the claimant_cred_handle the output_context_handle as returned by GSS_Init_sec_context() for a previously created CCM-BIND context. Such an approach may work well with applications that normally pass GSS_C_NO_CREDENTIAL as the claimant_cred_handle.

2. The second approach derives from the observation that normally, the first time GSS_Init_sec_context() is called, the input_token field is NULL and the initial context_handle (type gss_ctx_id_t) is also NULL. The input_token is supposed to be the token received from the target's context acceptance routine, which has the XDR type CCM_MIC_resp_t. Overloading the input_token is one way. By passing in a non-null input_token, and a NULL pointer to the context_handle (using the C bindings calling conventions for gss_init_sec_context()), this will tell the CCM-MIC initiator that input_token containing information to to associate a new CCM-MIC context with an existing CCM-BIND context. In the C programming language, we could thus have have input_token containing:

```
typedef struct {
     gss_ctx_id_t context_ptr;
} CCM_MIC_initiator_bootstrap_t;
```

The CCM entry point for creating contexts on the initiator side would, if being called for the first time (*context_handle is NULL), interpret the presence of the input token with an invalid status as the CCM_MIC_initiator_bootstrap_t. It would use context_ptr to lookup the corresponding gss_targ_ctx in the aforementioned gss_ctx_id_t to gss_targ_ctx mapping table. It would then proceed to generate an output token encoded as XDR type CCM_MIC_init_t, described in the section entitled "Initial Context Token for CCM-MIC".

Regardless of the approach taken, the first time GSS_Init_sec_context is called, assuming success, it will return GSS_S_CONTINUE_NEEDED, because it will need to process the token returned by the target. The second time it is called, assuming success, it will return GSS_S_COMPLETE.

9.4. Accepting CCM-MIC Contexts

The CCM-MIC target receives an opaque gss_targ_ctx value as part of the mechanism dependent part of the initial context token. Originally, this opaque handle came from the target as a result of previously creating a context via a CCM-BIND context exchange. If the opaque handle is still valid, then the target can easily determine the original CCM-BIND context, and from that, the CCM-BIND mechanism's context. With the underlying context, GSS_VerifyMIC() can be invoked (with a qop_req of CCM_REAL_QOP (1)) to verify the mic_nonce of the input token, and GSS_GetMIC() can be used to generate the mic_init_tkn field of the output token. By comparing the ctx_sh_number in the initiator's token with highest value recorded by the target, the target takes care to ensure that initiator has not replayed a short token.

9.5. Non-Token Generating GSS-API Routines

Since the CCM module will record the underlying mechanism's context pointer in its internal data structures, this provides a simple answer to what to do when GSS-API is invoked on a CCM context that does not generate any tokens for the GSS peer. When CCM is called for such an operation, it simply re-invokes the GSS-API call, but on the recorded underlying context.

9.6. CCM-MIC and GSS_Delete_sec_context()

The CCM-MIC entry point for GSS_Delete_sec_context() should not call the underlying mechanism's GSS_Delete_sec_context() routine. If it did, this would effectively delete all CCM-MIC context's associating with the same underlying mechanism.

9.7. GSS Status Codes

9.7.1. Status Codes for CCM-BIND

CCM-BIND mechanisms define no minor status codes. If the underlying mechanism is not available, then a CCM-BIND mechanism will return GSS_S_BAD_MECH and minor status of zero. Otherwise, it will return whatever major and minor status codes the underlying mechanism returns.

9.7.2. Status Codes for CCM-MIC

Generally, major and minor status codes for will be whatever major and minor status codes the underlying CCM-BIND mechanism returns. However, for GSS_Init_sec_context() and GSS_Accept_sec_context(), this is not the case because the those operations are invoking routines (GSS_Wrap() and GSS_Unwrap()) that have major statuses that are not subsets of the legal status returns from GSS_Init_sec_context() and GSS_Accept_sec_context(). Moreover, in some cases for GSS_Init_sec_context(), the minor and major status are driven from the target, and the target's codes will not always be among the legal set for GSS_Init_sec_context().

9.7.2.1. CCM-MIC: GSS_Accept_sec_context() status codes

The minor status code for GSS_Accept_sec_context is always from the set defined in the CCM_MIC_status_t type. If GSS_Unwrap() reports a major status failure, then the minor status will be CCM_ERR_TKN_UNWRAP, and the reported major status will what GSS_Unwrap() reports, with exceptions as according to the following table:

major status code from GSS_Unwrap	major status code reported by GSS_Accept_sec_context to caller.
GSS_S_BAD_SIG	GSS_S_BAD_SIG
GSS_S_CONTEXT_EXPIRED	GSS_S_DEFECTIVE_TOKEN
GSS_S_GAP_TOKEN	GSS_S_DEFECTIVE_TOKEN
GSS_S_UNSEQ_TOKEN	GSS_S_DUPLICATE_TOKEN

If GSS_GetMIC() reports a major status failure, then the minor status will be CCM_ERR_TKN_GET_MIC, and the reported major status will be what GSS_GetMIC() reports, with exceptions as according to the following table:

major status code from GSS_GetMIC	major status code reported by GSS_Accept_sec_context() to caller.

GSS_S_BAD_QOP	GSS_S_FAILURE
GSS_S_CONTEXT_EXPIRED	GSS_S_DEFECTIVE_TOKEN

The target will always report the actual GSS major and minor codes to the initiator. The initiator will map the GSS major code as described in the next subsection.

9.7.2.2. CCM-MIC: GSS_Init_sec_context() status codes

The minor status code for GSS_Init_sec_context is always from the set defined in the CCM_MIC_status_t type.

If the minor status code came from the target, then that will always be what GSS_Init_sec_context() reports. The most of the minor codes from the target are to be mapped to the major status code as follows:

minor status code from target	major status code reported to caller of GSS_Init_sec_context()
CCM_OK	GSS_S_COMPLETE
CCM_ERR_HANDLE_MALFORMED	GSS_S_DEFECTIVE_TOKEN
CCM_ERR_HANDLE_EXPIRED	GSS_S_CREDENTIALS_EXPIRED
CCM_ERR_HANDLE_NOT_FOUND	GSS_S_CREDENTIALS_EXPIRED
CCM_ERR_TKN_REPLAY	GSS_S_DUPLICATE_TOKEN
CCM_ERR_CHAN_MISMATCH	GSS_S_BAD_BINDINGS
CCM_ERR_TKN_WRAP	GSS_S_FAILURE
CCM_ERR_TKN_VER_MIC	GSS_S_FAILURE

Note that in the above table CCM_ERR_TKN_WRAP and CCM_ERR_TKN_VER_MIC MUST not be returned by the target. But if they are, then the initiator reports GSS_S_FAILURE.

If the minor status code from the target is CCM_ERR_TKN_UNWRAP or CCM_ERR_TKN_GET_MIC, then the target will also report the major status code it got from GSS_Unwrap() or GSS_GetMIC(). The major status from the target will be be reported by GSS_Init_sec_context() to its caller with exceptions as according to the following table:

major status code from target	major status code reported by GSS_Init_sec_context() to caller
GSS_S_BAD_QOP	GSS_S_FAILURE
GSS_S_BAD_SIG	GSS_S_BAD_SIG
GSS_S_CONTEXT_EXPIRED	GSS_S_DEFECTIVE_TOKEN
GSS_S_GAP_TOKEN	GSS_S_DEFECTIVE_TOKEN
GSS_S_UNSEQ_TOKEN	GSS_S_DUPLICATE_TOKEN

If GSS_Wrap() fails on the initiator, then the minor status will be CCM_ERR_TKN_WRAP, and the major status will what GSS_Wrap() reports, with exceptions as according to the following table:

major status code from GSS_Wrap	major status code reported by GSS_Init_sec_context() to caller
GSS_S_CONTEXT_EXPIRED	GSS_S_DEFECTIVE_TOKEN or GSS_S_DEFECTIVE_CREDENTIAL
GSS_S_BAD_QOP	GSS_S_FAILURE

If GSS_VerifyMIC() fails on the initiator, then the minor status will be CCM_ERR_TKN_VER_MIC, and the major status will what GSS_VerifyMIC() reports, with exceptions as according to the following table:

major status code from GSS_VerifyMIC	major status code reported by GSS_Init_sec_context() to caller
GSS_S_CONTEXT_EXPIRED	GSS_S_DEFECTIVE_TOKEN
GSS_S_GAP_TOKEN	GSS_S_DEFECTIVE_TOKEN
GSS_S_UNSEQ_TOKEN	GSS_S_DUPLICATE_TOKEN

9.8. Channel Bindings on the Target

When an application invokes GSS_Accept_sec_context() on a CCM token, it won't know if channel bindings are required or not. Of course, it could inspect the OID of the input_token and determine the channel bindings directly if it is a CCM-BIND token, but normally applications will not parse the mechanism OID in an input token. And in any case, such inspection for a CCM-MIC token provides no information about channel bindings to the target application.

The application on the target will have to try GSS_Accept_sec_context() without channel bindings. If the target CCM mechanism requires channel bindings (as indicated by the GSS_S_BAD_BINDINGS), then the application will have to re-invoke GSS_Accept_sec_context() with the right channel bindings. If the channel bindings are the wrong type, then the CCM mechanism will indicate GSS_S_BAD_BINDINGS again. The application will have to iterate through all the valid types of bindings. The application can avoid this iteration if the bindings includes both, address and key bindings if at all possible. The CCM mechanisms should use only those parts of the application-provided bindings that they care for.

10. Advice for NFSv4 Implementors

The NFSv4.0 specification does not mandate CCM, so clients and servers should not insist on its use. When a server wants a client to try to use CCM, it can return a NFS4ERR_WRONGSEC error to the client. The client will then follow up with a SECINFO request. The response to the SECINFO request should list first the CCM-BIND mechanisms it supports, second the CCM-MIC mechanism (if supported), and finally, the conventional security flavors the server will accept for access to file object. If the client supports CCM, it will use it. Otherwise, it will have to stick with a conventional flavor.

Since the CCM-MIC OID is general, rather than a separate CCM-MIC OID for every real mechanism, the NFS server will have be careful to make sure that a CCM-MIC context is authorized access an object. For example suppose /export is exported such that SPKM-3 is the authorized underlying mechanism, and CCM-NULL + SPKM-3 and CCM-MIC are similarly authorized to access /export. Suppose CCM-NULL is created over a Kerberos V5 context, and then CCM-MIC is used to derived a context from the CCM-NULL context. If the NFS server simply records that the OID of CCM-MIC is authorized to access /export, then Kerberos V5 authenticated users will be mistakenly allowed access. Instead, the server needs to examine what context the CCM-MIC context is associated with, and check that context's OID against the authorized list of OIDs for /export.

11. Man in the Middle Attacks without CCM-KEY

In this example, NFS with/ RPCSEC_GSS will be the application, and IPsec the secure channel.

Man in the middle (MITM) avoidance means making sure that the client and server are the same at both layers, NFS and IPsec, but since the principal names at the one layer will be radically different from the names at the other, how can one be certain that there is no MITM at the IPsec layer before leaving it to IPsec to provide session protection to the NFS layer? The answer is to use channel bindings, which, conceptually, are an exchange, at the NFS/GSS layer, of signatures of the principal names or session ID/keys involved at the IPsec layer.

Consider an attacker who can cause a client's IPsec stack to establish an SA with the attacker, instead of the server intended by the NFS layer (this is accomplished by spoofing the DNS server). Suppose further that the attacker can fool the client's IPsec layer without also fooling its NFS/RPCSEC_GSS layer (for example, if Kerberos V5 is being used as the real mechanism, and avoids the use of DNS to canonicalize the server principal name -- admittedly, this

avoidance is unlikely -- a DNS spoof attack will be detected by the

NFS client, because the Kerberos Key Distribution Center (KDC) generates tickets associated with pairs of principals, not host names). Suppose that the attacker's host is in part of the site's IPsec infrastructure (perhaps the attacker broke into that host). Then the attacker might be able to act as a MITM between the client and the server who gets all the plain text and even gets to modify it, if CCM-NULL is wrapping Kerberos V5 at the RPCSEC_GSS level. Both, the client and the server would see that IPsec is in use between them, but they would each see a different ID for its IPsec peer. Channel bindings are used to prove that the client and server each see the same two peer names at the lower (in this case, IPsec) layer, and therefore with CCM-KEY there is no MITM.

DNSSEC would of course defeat the attack, but DNSSEC was not, at the time this document was written, in widespread use.

12. Security Considerations

There are many considerations for the use CCM, since it is reducing security at one protocol layer in trade for equivalent security at another layer. In this discussion, we will assume that cryptography is being used in the application and lower protocol layers.

* CCM should not be used whenever the combined key strength/algorithm strength of the lower protocol layer securing the connection is weaker than what the underlying GSS context can provide.

* CCM should not be used if the lower level protocol does not offer comparable or superior security services to that the application would achieve with GSS. For example, if the lower level protocol offers integrity, but the application wants privacy, then CCM is inappropriate.

* The use of CCM contexts over secured connections can be characterized nearly secure instead of as secure as using the underlying GSS context for protecting each application message procedure call. The reason is that applications can multiplex the traffic of multiple principals over a single connection and so the ciphertext in the traffic is encrypted with multiple session keys. Whereas, a secure connection method such as IPsec is protected with per host session keys. Therefore, an attacker has more cipher text per session key to perform cryptanalysis via connections protected with IPsec, versus connections protected with GSS.

* Related to the previous bullet, the management of private keys for a secure channel is often outside the control of the user of

CCM. If the secure channel's private keys are compromised, then

all users of the secure channel are compromised.

* CCM contexts created during one session or transport connection SHOULD not be used for subsequent sessions or transport connections. In other words, full initiator to target authentication SHOULD occur each time a session or transport connection is established. Otherwise, there is nothing preventing an attacker from using a CCM context from one authenticated session or connection to trivially establish another, unauthenticated session or connection. For efficiency, a CCM-BIND context from a previous session MAY be used to establish a CCM-MIC context.

  If the application protocol using CCM has no concept of a session and does not use a connection oriented transport, then there is no sequence of state transitions that tie the CCM context creation steps with the subsequent message traffic of the application protocol. Thus it can be hard to assert that the subsequent message traffic is truly originated by the CCM initiator's principal. For this reason, CCM SHOULD NOT be used with applications that do not have sessions or do not use connection oriented transports.

* The underlying secure channel SHOULD be end to end, from initiator to the target. It is permissible for the user to configure the underlying secure channel to not be end to end, but this should only be done if user has confidence in the intermediate end points. For example, suppose the application is being used behind a firewall that performs network address translation. It is possible to have an IPsec secure channel from the initiator to the firewall, and a second secure channel from the firewall to the target, but not from the initiator to the target. So, if the firewall is compromised by an attacker in the middle, the use of CCM to avoid per message authentication is useless. Furthermore, without channel bindings mandated by CCM-KEY, it is not possible for the initiator and target to enforce end to end channel security. Of course, if the initiator's node created a IP-layer tunnel between it and the target, end to end channel security would be achieved, but without the use of CCM-KEY, the initiator and target applications would have no way of knowing that.

* It has been stated that it is not uncommon to find IPsec deployments where multiple nodes share common private keys [Black]. The use of CCM is discouraged in such environments, since the compromise of one node compromises all the other nodes sharing the same private key.

* Applications using CCM MUST ensure that the binding between the

CCM context and the secure channel is legitimate for each message that references the CCM context. In other words, the referenced CCM context in a message MUST be established in the same secure channel as the message. The use of CCM-KEY enforces this binding.

* When the same secure channel is multiplexing traffic for multiple users, the initiator has to ensure the CCM context is only accessible to the initiator principal that has established it in the first place. One possible way to ensure that is by placing CCM contexts in the privileged address space offering only controlled indexed access.

* CCM does not unnecessarily inflate the scope of the trust domain, as does for example AUTH_SYS [RFC1831] over IPSec. By requiring the authentication in the CCM context initialization (using a previously established context), the trust domain does not extend to the client.

* Both the traditional mechanisms and CCM rely on the security of the client to protect locally logged on users. Compromise of the client impacts all users on the same client. CCM does not make the problem worse.

* The CCM context MUST be established over the same secure channel that the subsequent message traffic will be using. This way, the binding between the initial authentication and the subsequent traffic is ensured. Again, the use of CCM-KEY is one way to assert this binding.

* The section entitled "CCM-KEY and Anonymous IPsec", suggests a method for simulating anonymous IPsec via self-signed certificates. If one is careless, this is will neuter all IPsec authentication, a real problem for those applications not using CCM-KEY. The use of the self-signed certificates in IPsec should be restricted by port in the IPsec Security Policy Database (SPD) only to those application using CCM-KEY. Note however, that port selector support is OPTIONAL in IPsec.

* If an application is using IPsec and is not using CCM-KEY, then then the site where the application is deployed should configure the IPsec SPD to carefully limit the ports and nodes that are allowed create security associations to application targets.

* CCM-KEY's IPsec bindings use public SA information, and CCM-ADDR's bindings are simply public network addresses. If the secure channel is IPsec, and non-anonymous certificates are used with IKE, then a MITM cannot spoof the target's and initiator's

IP addresses, because the attacker will presumably be unable to

spoof the Certificate Authority that signed the certificates. Thus, when IPsec is used as the secure channel, and non-anonymous certificates are used with IKE, CCM-ADDR is as secure as CCM-KEY.

* CCM contexts should not be used forever without re-authenticating periodically via the underlying mechanism. One rational approach is for the CCM context to persist no longer than the underlying mechanism context. Implementing this via the GSS-API is simple. Applications can periodically invoke gss_context_time() to find out how long the context will be valid. Moreover, CCM can enforce this by invoking gss_context_time() and the system time of day API to get an expiration date when the CCM mechanism is established. Each subsequent call can check the time of day against the expiration, and if expired, return GSS_S_CONTEXT_EXPIRED.

13. IANA Considerations

XXX Note 1 to IANA: The CCM-BIND mechanism OID prefixes and the CCM-MIC mechanism OID must be assigned and registered by IANA. Please look for TBD1 in this document and notify the RFC Editor what value you have assigned.

XXX Note 1 to RFC Editor: When IANA has made the OID assignments, please do the following:

* Delete the "XXX Note 1 to RFC Editor: ..." paragraph.

* Replace occurrences of TBD1 with the value assigned by IANA.

* Replace the "XXX Note 1 to IANA: ..." paragraph with:
  OIDs for the CCM-BIND mechanism prefix, and for the CCM-MIC mechanism have been assigned by, and registered with IANA, with this document as the reference.

XXX Note 2 to IANA: Please assign RPC flavor numbers for values currently place held in this document as TBD2 through TBD10. Also please establish the registry that RFC2623 mandates.

XXX Note 2 to RFC Editor: When IANA has made the RPC flavor number assignments, please do the following:

* Delete the "XXX Note 2 to RFC Editor: ..." paragraph.

* Replace occurrences of TBD2 through and including TBD10 withe flavor number assignments from IANA.

Section 6, "IANA Considerations" of [RFC2623] established a registry for mapping GSS mechanism OIDs to RPC pseudo flavor numbers. This registry was augmented in the NFSv4 specification [RFC3530] with several more entries. This document adds the following entries to the registry:

```
 1 == number of pseudo flavor
 2 == name of pseudo flavor
 3 == mechanism's OID
 4 == quality of protection
 5 == RPCSEC_GSS service
```

1	2	3	4	5
TBD2	ccm-mic	1.3.6.1.5.5.TBD1.2	0	rpc_gss_svc_none
TBD3	ccm-null-krb5	1.3.6.1.5.5.TBD1.1.1. 1.2.840.113554.1.2.2	0	rpc_gss_svc_none
TBD4	ccm-addr-krb5	1.3.6.1.5.5.TBD1.1.2. 1.2.840.113554.1.2.2	0	rpc_gss_svc_none
TBD5	ccm-key-krb5	1.3.6.1.5.5.TBD1.1.3. 1.2.840.113554.1.2.2	0	rpc_gss_svc_none
TBD6	ccm-null-spkm3	1.3.6.1.5.5.TBD1.1.1. 1.3.6.1.5.5.1.3	0	rpc_gss_svc_none
TBD6	ccm-addr-spkm3	1.3.6.1.5.5.TBD1.1.2. 1.3.6.1.5.5.1.3	0	rpc_gss_svc_none
TBD7	ccm-key-spkm3	1.3.6.1.5.5.TBD1.1.3. 1.3.6.1.5.5.1.3	0	rpc_gss_svc_none
TBD8	ccm-null-lipkey	1.3.6.1.5.5.TBD1.1.1. 1.3.6.1.5.5.1.3	0	rpc_gss_svc_none
TBD9	ccm-addr-lipkey	1.3.6.1.5.5.TBD1.1.2. 1.3.6.1.5.5.1.3	0	rpc_gss_svc_none
TBD10	ccm-addr-lipkey	1.3.6.1.5.5.TBD1.1.3. 1.3.6.1.5.5.1.3	0	rpc_gss_svc_none

14. Acknowledgements

Dave Noveck, for the observation that NFS version 4 servers could downgrade from integrity service to plain authentication service if IPsec was enabled. David Black, Peng Dai, Sam Hartman, and Julian

Satran, for their critical comments. Much of the text for the "Security Considerations" section comes directly from David and Peng.

15. Normative References

[RFC1832]
R. Srinivasan, RFC1832, "XDR: External Data Representation Standard", August, 1995.

[RFC2025]
C. Adams, RFC2025: "The Simple Public-Key GSS-API Mechanism (SPKM)," October 1996, Status: Standards Track.

[RFC2119]
S. Bradner, RFC2119, "Key words for use in RFCs to Indicate Requirement Levels," March 1997.

[RFC2401]
S. Kent, R. Atkinson, RFC2401, "Security Architecture for the Internet Protocol ", November, 1998.

[RFC2409]
D. Harkins and D. Carrel, RFC2119: "The Internet Key Exchange (IKE)," November 1998.

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

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

[RFC2847]
M. Eisler, RFC2847: "LIPKEY - A Low Infrastructure Public Key Mechanism Using SPKM," June 2000, Status: Standards Track.

[FIPS]U.S. Department of Commerce / National Institute of Standards and Technology, FIPS PUB 180-1, "Secure Hash Standard", May 11, 1993.

[IKEv2]
C. Kaufman, draft-ietf-ipsec-ikev2-07.txt: "Internet Key Exchange (IKEv2) Protocol," A work in progress, April 2003.

XXX - Note 3 to RFC Editor: In the event this work in progress is not approved for publication when the CCM document is, then the sections of the CCM document that refer to IKEv2 in a

normative manner are to be removed for submission as a separate document.

[SSHv2]
T. Ylonen et. al., draft-ietf-secsh-transport-15.txt: "SSH Transport Layer Protocol," A work in progress, September 2002.

XXX - Note 4 to RFC Editor: In the event this work in progress is not approved for publication when the CCM document is, then the sections of the CCM document that refer to SSHv2 in a normative manner are to be removed for submission as a separate document.

16. Informative References

[RFC1831]
R. Srinivasan, RFC1831, "RPC: Remote Procedure Call Protocol Specification Version 2", August, 1995.

[RFC1964]
J. Linn, RFC1964, "The Kerberos Version 5 GSS-API Mechanism", June 1996.

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

[RFC2623]
M. Eisler, RFC2623, "NFS Version 2 and Version 3 Security Issues and the NFS Protocol's Use of RPCSEC_GSS and Kerberos V5", June 1999.

[RFC3530]
S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, D. Noveck, RFC3530, "Network File System (NFS) version 4 Protocol", April 2003.

[Black]
D. Black, EMail message on the NFSv4 working group alias, February 28, 2003.

[DAFS]
Mark Wittle (Editor), "DAFS Direct Access File System Protocol, Version: 1.00", September 1, 2001.

17. Authors' Addresses

Mike Eisler

5765 Chase Point Circle
Colorado Springs, CO 80919
USA

Phone: 719-599-9026
EMail: mike@eisler.com

Nicolas Williams
Sun Microsystems, Inc.
5300 Riata Trace CT
Austin, TX 78727
USA

EMail: nicolas.williams@sun.com

18. IPR Notices

The IETF takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on the IETF's procedures with respect to rights in standards-track and standards-related documentation can be found in BCP-11. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementors or users of this specification can be obtained from the IETF Secretariat.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this standard. Please address the information to the IETF Executive Director.

19. Copyright Notice