Network Working Group
Internet-Draft

M. Eisler
Network Appliance, Inc.
N. Williams
Sun Microsystems, Inc.
July 2004

The Channel Conjunction Mechanism (CCM) for GSS
draft-ietf-nfsv4-ccm-03

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 3
2. Introduction 3
3. Overview 3
3.1. Example Application of CCM 4
3.2. A Suite of CCM Mechanisms 4
3.3. QOPs 5
4. Token Formats 5
4.1. Mechanism Object Identifier 5
4.2. Tokens for the CCM-BIND mechanisms 6
4.2.1. Context Establishment Tokens for CCM-BIND Mechanisms . . . 6
4.2.1.1. Initial Context Token for CCM-BIND 6
4.2.1.2. Subsequent Context Tokens for CCM-BIND 6
4.2.2. Post Context Establishment Token Formats 8
4.2.2.1. MIC Token for CCM-BIND 9
4.2.2.2. Wrap Token for CCM-BIND 9
4.2.3. Other Tokens for CCM-BIND 9
4.3. Tokens for CCM-MIC 9
4.3.1. Context Establishment Tokens for CCM-MIC 9
4.3.1.1. Initial Context Token for CCM-MIC 9
4.3.1.2. Subsequent Context Tokens for CCM-MIC 11
4.3.1.2.1. Subsequent Initiator Context Token for CCM-MIC . . . 11
4.3.1.2.2. Response Token for CCM-MIC 11
4.3.2. MIC Token for CCM-MIC 13
4.3.3. Wrap Token for CCM-MIC 13
4.3.4. Context Deletion Token 13
4.3.5. Exported Context Token 13
4.3.6. Other Tokens for CCM-MIC 13
5. Implementation Issues 13
5.1. Management of ccmMicCcmBindCtxHandle 14
5.2. CCM-BIND Versus CCM-MIC 14
5.3. Initiating CCM-MIC Contexts 14
5.4. Accepting CCM-MIC Contexts 16
5.5. Non-Token Generating GSS-API Routines 16
5.6. CCM-MIC and GSS_Delete_sec_context() 16
5.7. GSS Status Codes 16
5.7.1. Status Codes for CCM-BIND 16
5.7.2. Status Codes for CCM-MIC 17
5.7.2.1. CCM-MIC: GSS_Accept_sec_context() status codes 17
5.7.2.2. CCM-MIC: GSS_Init_sec_context() status codes 18
6. Advice for NFSv4 Implementors 19
7. Security Considerations 19
8. CCM XDR Description 22
9. IANA Considerations 24
10. Acknowledgements 25
11. Normative References 25
12. Informative References 26

13. Authors' Addresses 27

14. IPR Notices 27
15. Copyright Notice 28

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. GSS can support a concept call channel bindings, where a GSS mechanism context is bound to a secure channel (see section 1.1.6 of RFC2743). As noted in RFC2743, the purpose of channel bindings is to limit the scope within which an intercepted GSS context token can be used by an attacker. Non-null channel bindings can be derived from the secure channel's encryption keys or derived from the 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 defined. CCM-NULL requires the use of null channel bindings.

Non-null channel bindings are highly dependent on the underlying channel's characteristics. For example with IPsec, the channel bindings for manual keys, IKEv1, and IKEv2 would be different from each other. Non-null channel bindings are also underspecified in the current GSS specifications. Thus this document does not define a key-based or address-based form for CCM.

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-NULL and CCM-MIC. Specifications that make normative references to CCM are free to mandate any subset of the suite CCM mechanisms.

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 in the next section) in the value to generate and verify the MICs.

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. When qop_req is 0:

* The MIC token for CCM is always a single octet, of value zero.

* The wrap output token for CCM is equal to the concatenation of the input token and a single octet (which is equal to zero).

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. This document defines one such mechanism:

{ iso(1) identified-organization(3) dod(6) internet(1)

```
security(5) mechanisms(5) ccm-family(TBD1) ccm-bind(1) ccm-
null(1) }
```

The above object identifier is not a complete mechanism OID. A complete mechanism OID would consist of the above OID as prefix, followed by a real mechanism OID, such as that of Kerberos V5 as defined in [RFC1964].

Future extensions to CCM may add non-null channel binding mechanisms under the ccm-bind(1) node in the OID space.

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

4.2.1. 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 can use one more context token exchange than the underlying real mechanism. This is so that that target can be protected from a replay attack in the event the real mechanism is does not have replay protection. See [Kasslin] for more information on the attack.

4.2.1.1. 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. 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.

4.2.1.2. 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.

The form of a SubsequentContextToken for a CCM-BIND mechanism, is always encoded in XDR [RFC1832]. There is a form for the context the target produces for the initiator, and a form for the context the initiator produces for the target. These forms are:

```
enum ccmVerifyState {
        CCM_UNVERIFIED = 0,
        CCM_VERIFY_FAILED = 1,
        CCM_VERIFIED = 2
};

struct ccmBindSubCtxTknTargToInit {
        ccmVerifyState ccmBindStatus;
        opaque ccmBindRealToken<>;
        opaque ccmBindNonce<>;
};

struct ccmBindSubCtxTknInitToTarg {
        opaque ccmBindRealToken<>;
        opaque ccmBindMic<>;
};
```

If of non-zero length, the ccmBindRealToken<> field in each of the above two data types is always that generated by the real mechanism that CCM-BIND is operating over.

The ccmBindSubCtxTknTargToInit type is the token the target returns to the initiator. The ccmBindStatus is usually set to CCM_UNVERIFIED, except as described otherwise. The ccmBindRealToken field is always equal to the output of the real mechanism's GSS_Accept_sec_context() entry point. The ccmBindNonce field will always be a zero length until the real mechanism's GSS_Accept_sec_context() routine returns GSS_S_COMPLETE. If CCM-BIND knows that the underlying mechanism has replay protection during context establishment, then it MAY set ccmBindStatus to CCM_VERIFIED, and set ccmBindNonce to a zero length value. In this case, CCM-BIND returns GSS_S_COMPLETE to the caller of its GSS_Accept_sec_context() entry point.

If CCM-BIND does not know if the underlying mechanism has replay protection, or it knows it does not have replay protection, then ccmBindStatus MUST be CCM_UNVERIFIED, and ccmBindNonce MUST be a non-zero length randomly generated string, or a pseudo-random string generated such that it is unlikely the target will generate a duplicate in the future. While the underlying real mechanism returned GSS_S_COMPLETE, CCM-BIND returns GSS_S_CONTINUE_NEEDED to the caller

of its GSS_Accept_sec_context() entry point.

When the initiator receives a ccmBindSubCtxTknTargToInit token, it will call the real mechanism's GSS_Init_sec_context() entry point to process the ccmBindRealToken<> value. If the GSS_Init_sec_context() entry point of the real mechanism returns GSS_S_CONTINUED needed, then the CCM-BIND initiator will set the ccmBindMic field to a zero length string. If GSS_S_COMPLETE was returned, then normally the ccmBindStatus will be CCM_UNVERIFIED. If so, then the expectation of the CCM-BIND initiator is that the ccmBindNonce field from the target MUST be non-zero length. The initiator will set the ccmBindMic field to the output of GSS_GetMIC() of the ccmBindNonce field, and GSS_S_CONTINUED will be returned to the caller of CCM-BIND's GSS_Init_sec_context() entry point. If GSS_GetMIC(), fails, then the error should be returned to the caller of CCM-BIND's GSS_Init_sec_context() entry point, and null output token. If the error from GSS_GetMIC() is not among the set permitted to be returned from GSS_Init_sec_context(), then the error should be mapped as follows. GSS_S_CONTEXT_EXPIRED and GSS_S_BAD_QOP should be mapped to GSS_S_FAILURE.

If the initiator finds that the value the target returned for ccmBindStatus is CCM_VERIFIED, then the CCM-BIND context is established, and GSS_S_COMPLETE is returned to the caller of CCM_BIND's GSS_Init_sec_context() entry point.

When the target receives a ccmBindSubCtxTknInitToTarg with a ccmBindMic<> value that is non-zero in length, it will call GSS_VerifyMIC(). The message value will be the ccmBindNonce value the target generated in the previous context exchange. The per_msg_token argument will be the ccmBindMic value. The context_handle argument will be that of the established context of the underlying real mechanism. If GSS_VerifyMIC() returns GSS_S_COMPLETE, then ccmBindStatus will be set to CCM_VERIFIED. Otherwise, ccmBindStatus will be set to CCM_VERIFY_FAILED. The return value of GSS_VerifyMIC() will be returned to the caller of CCM-BIND's GSS_Accept_sec_context() entry point, except for those errors are not in the set permitted by GSS_Accept_sec_context. GSS_S_UNSEQ_TOKEN and GSS_S_GAP_TOKEN should be mapped to GSS_S_OLD_TOKEN. GSS_S_CONTEXT_EXPIRED should be mapped to GSS_S_FAILURE.

When the initiator receives a token with ccmBindStatus set to CCM_VERIFIED, it marks the CCM-BIND context as established, and returns GSS_S_COMPLETE to the caller of its GSS_Init_sec_context() entry point. If ccmBindStatus was set to CCM_DENIED, it returns GSS_S_FAILURE to the caller of its GSS_Init_sec_context() entry point.

4.2.2. Post Context Establishment Token Formats

4.2.2.1. 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 one octet in length with a value of zero. 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).

4.2.2.2. 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() concatenated with a single octet with a value of zero.

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).

4.2.3. Other Tokens for CCM-BIND

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

4.3. Tokens for CCM-MIC

4.3.1. Context Establishment Tokens for CCM-MIC

4.3.1.1. 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:

```
struct ccmMicUnWrappedInitToken {
        unsigned int ctxMicIndex;
        opaque ccmMicCcmBindCtxHandle[20];
        opaque ccmMicNonce<>;
};

/*
```

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

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 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 ccmMicCcmBindCtxHandle 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-NULL context exchange over a Kerberos V5 context exchange would be:

  ```
  SHA-1( {
       SHA-1(CCM-NULL's first initiator token),
       SHA-1(CCM-NULL's first initial target token),
       SHA-1(CCM-NULL's final initiator token),
       SHA-1(CCM-NULL's final target token)
  } )
  ```

  Since the SHA-1 standard mandates a 160 bit output, (20 octets), ccmMicCcmBindCtxHandle is a fixed length, 20 octet string.

* The field ccmMicNonce is set a random or pseudo random value. It MUST have length greater than or equal to that of ccmBindNonce value the target gave the server when the CCM-BIND context was established It is provided so as to ensure more variability of the the mic that GSS will calculate when ccmMicUnWrappedInitToken is GSS_Wrap()ed into ccmBindMicWrappedInitToken.

* The field ctxMicIndex is the identifier of the CCM-MIC context relative to the CCM-BIND context (as identified by ccmMicCcmBindCtxHandle) that the initiator is assigning. The value for ctxMicIndex is selected by the initiator such that it is larger than any previous ctxMicIndex for the given ccmMicCcmBindCtxHandle. This way, the target need only keep track of the largest ctxMicIndex received. Once ctxMicIndex has reached the maximum value for an unsigned 32 bit integer, the

given ccmMicCcmBindCtxHandle can no longer be used.

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

4.3.1.2. 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.

4.3.1.2.1. Subsequent Initiator Context Token for CCM-MIC

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

4.3.1.2.2. Response Token for CCM-MIC

The CCM response token, in XDR encoding is:

```
   enum ccmMicStatus {
           CCM_OK = 0,

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

           /*
            * The GSS context corresponding to
            * ccmMicCcmBindCtxHandle has expired.
            */

           CCM_ERR_HANDLE_EXPIRED = 2,

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

           /*
            * The ctxMicIndex has already been received
            * by the target.  Or the maximum ctxMicIndex 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

    };

    /*
     * GSS errors returned by the underlying mechanism
     */
    struct ccmMicRealGssErr {
            unsigned int ccmMicGssMajor;
            unsigned int ccmMicGssMinor;
    };

    /*
     * The response context token for CCM-MIC.
     */
    union ccmMicResp switch (ccmMicStatus status) {
            case CCM_OK:
                    opaque ccmMicRespInitTkn<>;
            case CCM_ERR_TKN_UNWRAP:
            case CCM_ERR_TKN_GET_MIC:
                    ccmMicRealGssErr ccmMicGssErr;
```

```
            default:
                    void;
    };
```

If a value of the status field is CCM_OK, then the CCM-MIC context has been established on the target. The field ccmMicRespInitTkn is equal to the output of GSS_GetMIC() (qop_req is CCM_REAL_QOP (1)) on the entire and first token that came from the initiator. In other words, the first 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 ccmMicRespInitTkn 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 ccmMicGssErr.ccmMicGssMajor and ccmMicGssErr.ccmMicGssMinor are set to the major and minor GSS statuses as returned by GSS_Unwrap() or GSS_GetMIC(). The values for the ccmMicGssMajor field are as defined in [RFC2744]. The values for the ccmMicGssMinor field are both mechanism dependent and mechanism implementation dependent. They are nonetheless potentially useful as debugging aids.

4.3.2. MIC Token for CCM-MIC

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

4.3.3. Wrap Token for CCM-MIC

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

4.3.4. Context Deletion Token

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

4.3.5. Exported Context Token

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

4.3.6. Other Tokens for CCM-MIC

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

5. 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.

5.1. Management of ccmMicCcmBindCtxHandle

The ccmMicCcmBindCtxHandle 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.

5.2. 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.

5.3. Initiating CCM-MIC Contexts

The issue is how to associate an CCM-BIND established security context with a new CCM-MIC context, There are 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 ccmMicCcmBindCtxHandle 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(). 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 established 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 ccmMicResp. 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 ccmMicCcmBindCtxPtr;
} ccmMicCcmBindBootStrap;
```

   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 ccmMicCcmBindBootStrap. It would use ccmMicCcmBindCtxPtr to lookup the corresponding ccmMicCcmBindCtxHandle in the aforementioned gss_ctx_id_t to ccmMicCcmBindCtxHandle 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.

5.4. Accepting CCM-MIC Contexts

The CCM-MIC target receives an opaque ccmMicCcmBindCtxHandle 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 ccmMicRespInitTkn field of the output token. By comparing the ctxMicIndex in the initiator's token with highest value recorded by the target, the target takes care to ensure that initiator has not replayed an initial CCM-MIC context token.

5.5. 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.

5.6. 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.

5.7. GSS Status Codes

5.7.1. 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.

5.7.2. 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().

5.7.2.1. 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 ccmMicStatus 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.

5.7.2.2. 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 ccmMicStatus 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

6. Advice for NFSv4 Implementors

The NFSv4.0 specification does not mandate CCM, so client and server implementations 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.

7. 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 or connection 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. 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.

* 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.

* 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.

* If an application is using IPsec and CCM-NULL 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 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.

8. CCM XDR Description

Here is the XDR description for CCM-BIND and CCM-MIC:

```
enum ccmVerifyState {
        CCM_UNVERIFIED = 0,
        CCM_VERIFY_FAILED = 1,
        CCM_VERIFIED = 2
};

struct ccmBindSubCtxTknTargToInit {
        ccmVerifyState ccmBindStatus;
        opaque ccmBindRealToken<>;
        opaque ccmBindNonce<>;
};

struct ccmBindSubCtxTknInitToTarg {
        opaque ccmBindRealToken<>;
        opaque ccmBindMic<>;
};

enum ccmMicStatus {
        CCM_OK = 0,

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

        /*
         * The GSS context corresponding to
```

```
             * ccmMicCcmBindCtxHandle has expired.
             */
            CCM_ERR_HANDLE_EXPIRED = 2,

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

            /*
             * The ctxMicIndex has already been received
             * by the target.  Or the maximum ctxMicIndex 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

      };
```

```
/*
 * GSS errors returned by the underlying mechanism
 */
struct ccmMicRealGssErr {
        unsigned int ccmMicGssMajor;
        unsigned int ccmMicGssMinor;
};

/*
 * The response context token for CCM-MIC.
 */
union ccmMicResp switch (ccmMicStatus status) {
        case CCM_OK:
                opaque ccmMicRespInitTkn<>;
        case CCM_ERR_TKN_UNWRAP:
        case CCM_ERR_TKN_GET_MIC:
                ccmMicRealGssErr ccmMicGssErr;
        default:
                void;
};
```

9. 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 TBD5. 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 TBD5 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-null-spkm3	1.3.6.1.5.5.TBD1.1.1. 1.3.6.1.5.5.1.3	0	rpc_gss_svc_none
TBD5	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

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

11. 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.

12. 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.

[Kasslin]
Kasslin, K. "Attacks on Kerberos V in a Windows 2000 Environment", 2003.
http://www.hut.fi/~autikkan/hakkeri/docs/phase1/pdf/LATEST_final_report.pdf

13. 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

14. IPR Notices

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.

15. Copyright Notice