                  Encryption and Checksum Specifications
                            for Kerberos 5

Status of This Memo

   This document specifies an Internet standards track protocol for the
   Internet community, and requests discussion and suggestions for
   improvements. Please refer to the current edition of the "Internet
   Official Protocol Standards" (STD 1) for the standardization state
   and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Abstract

   This document describes a framework for defining encryption and
   checksum mechanisms for use with the Kerberos protocol, defining an
   abstraction layer between the Kerberos protocol and related
   protocols, and the actual mechanisms themselves. The document also
   defines several mechanisms. Some are taken from RFC 1510, modified
   in form to fit this new framework and occasionally modified in
   content when the old specification was incorrect. New mechanisms are
   presented here as well. This document does NOT indicate which
   mechanisms may be considered "required to implement".

Table of Contents

1. Introduction

   The Kerberos protocols [Kerb] are designed to encrypt messages of
   arbitrary sizes, using block encryption ciphers or, less commonly,
   stream encryption ciphers. Encryption is used to prove the
   identities of the network entities participating in message
   exchanges. However, nothing in the Kerberos protocol requires that
   any specific encryption algorithm be used, as long as the algorithm
   includes certain operations.

   The following sections specify the encryption and checksum mechanisms
   currently defined for Kerberos, as well as a framework for defining
   future mechanisms. The encoding, chaining, padding, and other
   requirements for each are described. Appendix A gives test vectors
   for several functions.

2. Concepts

   Both encryption and checksum mechanisms are profiled in later
   sections. Each profile specifies a collection of operations and
   attributes that must be defined for a mechanism. A Kerberos
   encryption or checksum mechanism specification is not complete if it
   does not define all of these operations and attributes.

   An encryption mechanism must provide for confidentiality and
   integrity of the original plaintext. (Incorporating a checksum may
   permit integrity checking, if the encryption mode does not provide an
   integrity check itself.) It must also provide non-malleability

[Bellare98] [Dolev91]. Use of a random confounder prepended to the
plaintext is recommended. It should not be possible to determine if
two ciphertexts correspond to the same plaintext without the key.

A checksum mechanism [1] must provide proof of the integrity of the
associated message and must preserve the confidentiality of the
message in case it is not sent in the clear. Finding two plaintexts
with the same checksum should be infeasible. It is NOT required that
an eavesdropper be unable to determine whether two checksums are for
the same message, as the messages themselves would presumably be
visible to any such eavesdropper.

Due to advances in cryptography, some cryptographers consider using
the same key for multiple purposes unwise. Since keys are used in
performing a number of different functions in Kerberos, it is
desirable to use different keys for each of these purposes, even
though we start with a single long-term or session key.

We do this by enumerating the different uses of keys within Kerberos
and by making the "usage number" an input to the encryption or
checksum mechanisms; such enumeration is outside the scope of this
document. Later sections define simplified profile templates for
encryption and checksum mechanisms that use a key derivation function
applied to a CBC mode (or similar) cipher and a checksum or hash
algorithm.

We distinguish the "base key" specified by other documents from the
"specific key" for a specific encryption or checksum operation. It
is expected but not required that the specific key be one or more
separate keys derived from the original protocol key and the key
usage number. The specific key should not be explicitly referenced
outside of this document. The typical language used in other
documents should be something like, "encrypt this octet string using
this key and this usage number"; generation of the specific key and
cipher state (described in the next section) are implicit. The
creation of a new cipher-state object, or the re-use of one from a
previous encryption operation, may also be explicit.

New protocols defined in terms of the Kerberos encryption and
checksum types should use their own key usage values. Key usages are
unsigned 32-bit integers; zero is not permitted.

All data is assumed to be in the form of strings of octets or eight-
bit bytes. Environments with other byte sizes will have to emulate
this behavior in order to get correct results.

Each algorithm is assigned an encryption type (or "etype") or
checksum type number, for algorithm identification within the
Kerberos protocol. The full list of current type number assignments
is given in section 8.

3.   Encryption Algorithm Profile

An encryption mechanism profile must define the following attributes
and operations. The operations must be defined as functions in the
mathematical sense. No additional or implicit inputs (such as
Kerberos principal names or message sequence numbers) are permitted.

protocol key format
   This describes which octet string values represent valid keys.
   For encryption mechanisms that don't have perfectly dense key
   spaces, this will describe the representation used for encoding
   keys. It need not describe invalid specific values; all key
   generation routines should avoid such values.

specific key structure
   This is not a protocol format at all, but a description of the
   keying material derived from the chosen key and used to encrypt or
   decrypt data or compute or verify a checksum. It may, for
   example, be a single key, a set of keys, or a combination of the
   original key with additional data. The authors recommend using
   one or more keys derived from the original key via one-way key
   derivation functions.

required checksum mechanism
   This indicates a checksum mechanism that must be available when
   this encryption mechanism is used. Since Kerberos has no built in
   mechanism for negotiating checksum mechanisms, once an encryption
   mechanism is decided, the corresponding checksum mechanism can be
   used.

key-generation seed length, K
   This is the length of the random bitstring needed to generate a
   key with the encryption scheme's random-to-key function (described
   below). This must be a fixed value so that various techniques for
   producing a random bitstring of a given length may be used with
   key generation functions.

key generation functions
   Keys must be generated in a number of cases, from different types
   of inputs. All function specifications must indicate how to
   generate keys in the proper wire format and must avoid generating
   keys that significantly compromise the confidentiality of
   encrypted data, if the cryptosystem has such. Entropy from each

source should be preserved as much as possible. Many of the
inputs, although unknown, may be at least partly predictable
(e.g., a password string is likely to be entirely in the ASCII
subset and of fairly short length in many environments; a semi-
random string may include time stamps). The benefit of such
predictability to an attacker must be minimized.

string-to-key (UTF-8 string, UTF-8 string, opaque)->(protocol-key)
This function generates a key from two UTF-8 strings and an opaque
octet string. One of the strings is usually the principal's pass
phrase, but generally it is merely a secret string. The other
string is a "salt" string intended to produce different keys from
the same password for different users or realms. Although the
strings provided will use UTF-8 encoding, no specific version of
Unicode should be assumed; all valid UTF-8 strings should be
allowed. Strings provided in other encodings MUST first be
converted to UTF-8 before applying this function.

The third argument, the octet string, may be used to pass
mechanism-specific parameters into this function. Since doing so
implies knowledge of the specific encryption system, generating
non-default parameter values should be an uncommon operation, and
normal Kerberos applications should be able to treat this
parameter block as an opaque object supplied by the Key
Distribution Center or defaulted to some mechanism-specific
constant value.

The string-to-key function should be a one-way function so that
compromising a user's key in one realm does not compromise it in
another, even if the same password (but a different salt) is used.

random-to-key (bitstring[K])->(protocol-key)
This function generates a key from a random bitstring of a
specific size. All the bits of the input string are assumed to be
equally random, even though the entropy present in the random
source may be limited.

key-derivation (protocol-key, integer)->(specific-key)
In this function, the integer input is the key usage value, as
described above. An attacker is assumed to know the usage values.
The specific-key output value was described in section 2.

string-to-key parameter format
This describes the format of the block of data that can be passed
to the string-to-key function above to configure additional
parameters for that function. Along with the mechanism of
encoding parameter values, bounds on the allowed parameters should
also be described to avoid allowing a spoofed KDC to compromise

the user's password. If practical it may be desirable to
construct the encoding so that values unacceptably weakening the
resulting key cannot be encoded.

Local security policy might permit tighter bounds to avoid excess
resource consumption. If so, the specification should recommended
defaults for these bounds. The description should also outline
possible weaknesses if bounds checks or other validations are not
applied to a parameter string received from the network.

As mentioned above, this should be considered opaque to most
normal applications.

default string-to-key parameters (octet string)
    This default value for the "params" argument to the string-to-key
    function should be used when the application protocol (Kerberos or
    other) does not explicitly set the parameter value. As indicated
    above, in most cases this parameter block should be treated as an
    opaque object.

cipher state
    This describes any information that can be carried over from one
    encryption or decryption operation to the next, for use with a
    given specific key. For example, a block cipher used in CBC mode
    may put an initial vector of one block in the cipher state. Other
    encryption modes may track nonces or other data.

    This state must be non-empty and must influence encryption so that
    messages are decrypted in the same order they were a encrypted, if
    the cipher state is carried over from one encryption to the next.
    Distinguishing out-of-order or missing messages from corrupted
    messages is not required. If desired, this can be done at a
    higher level by including sequence numbers and not "chaining" the
    cipher state between encryption operations.

    The cipher state may not be reused in multiple encryption or
    decryption operations. These operations all generate a new cipher
    state that may be used for following operations using the same key
    and operation.

    The contents of the cipher state must be treated as opaque outside
    of encryption system specifications.

initial cipher state (specific-key, direction)->(state)
    This describes the generation of the initial value for the cipher
    state if it is not being carried over from a previous encryption
    or decryption operation.

This describes any initial state setup needed before encrypting
arbitrary amounts of data with a given specific key. The specific
key and the direction of operations to be performed (encrypt
versus decrypt) must be the only input needed for this
initialization.

This state should be treated as opaque in any uses outside of an
encryption algorithm definition.

IMPLEMENTATION NOTE: [Kerb1510] was vague on whether and to what
degree an application protocol could exercise control over the
initial vector used in DES CBC operations. Some existing
implementations permit setting the initial vector. This framework
does not provide for application control of the cipher state
(beyond "initialize" and "carry over from previous encryption"),
as the form and content of the initial cipher state can vary
between encryption systems and may not always be a single block of
random data.

New Kerberos application protocols should not assume control over
the initial vector, or that one even exists. However, a general-
purpose implementation may wish to provide the capability, in case
applications explicitly setting it are encountered.

encrypt (specific-key, state, octet string)->(state, octet string)
   This function takes the specific key, cipher state, and a non-
   empty plaintext string as input and generates ciphertext and a new
   cipher state as outputs. If the basic encryption algorithm itself
   does not provide for integrity protection (e.g., DES in CBC mode),
   then some form of verifiable MAC or checksum must be included.
   Some random factor such as a confounder should be included so that
   an observer cannot know if two messages contain the same
   plaintext, even if the cipher state and specific keys are the
   same. The exact length of the plaintext need not be encoded, but
   if it is not and if padding is required, the padding must be added
   at the end of the string so that the decrypted version may be
   parsed from the beginning.

   The specification of the encryption function must indicate not
   only the precise contents of the output octet string, but also the
   output cipher state. The application protocol may carry the
   output cipher state forward from one encryption with a given
   specific key to another; the effect of this "chaining" must be
   defined [2].

   Assuming that values for the specific key and cipher state are
   correctly-produced, no input octet string may result in an error
   indication.

   decrypt (specific-key, state, octet string)->(state, octet string)
      This function takes the specific key, cipher state, and ciphertext
      as inputs and verifies the integrity of the supplied ciphertext.
      If the ciphertext's integrity is intact, this function produces
      the plaintext and a new cipher state as outputs; otherwise, an
      error indication must be returned, and the data discarded.

      The result of the decryption may be longer than the original
      plaintext, as, for example, when the encryption mode adds padding
      to reach a multiple of a block size. If this is the case, any
      extra octets must come after the decoded plaintext. An
      application protocol that needs to know the exact length of the
      message must encode a length or recognizable "end of message"
      marker within the plaintext [3].

      As with the encryption function, a correct specification for this
      function must indicate not only the contents of the output octet
      string, but also the resulting cipher state.

   pseudo-random (protocol-key, octet-string)->(octet-string)
      This pseudo-random function should generate an octet string of
      some size that is independent of the octet string input. The PRF
      output string should be suitable for use in key generation, even
      if the octet string input is public. It should not reveal the
      input key, even if the output is made public.

   These operations and attributes are all that is required to support
   Kerberos and various proposed preauthentication schemes.

   For convenience of certain application protocols that may wish to use
   the encryption profile, we add the constraint that, for any given
   plaintext input size, a message size must exist between that given
   size and that size plus 65,535 such that the length of the decrypted
   version of the ciphertext will never have extra octets at the end.

   Expressed mathematically, for every message length L1, there exists a
   message size L2 such that

      L2 >= L1
      L2 < L1 + 65,536
      for every message M with |M| = L2, decrypt(encrypt(M)) = M

   A document defining a new encryption type should also describe known
   weaknesses or attacks, so that its security may be fairly assessed,
   and should include test vectors or other validation procedures for
   the operations defined. Specific references to information that is
   readily available elsewhere are sufficient.

4. Checksum Algorithm Profile

   A checksum mechanism profile must define the following attributes and
   operations:

   associated encryption algorithm(s)
      This indicates the types of encryption keys this checksum
      mechanism can be used with.

      A keyed checksum mechanism may have more than one associated
      encryption algorithm if they share the same wire-key format,
      string-to-key function, default string-to-key-parameters, and key
      derivation function. (This combination means that, for example, a
      checksum type, key usage value, and password are adequate to get
      the specific key used to compute a checksum.)

      An unkeyed checksum mechanism can be used with any encryption
      type, as the key is ignored, but its use must be limited to cases
      where the checksum itself is protected, to avoid trivial attacks.

   get_mic function
      This function generates a MIC token for a given specific key (see
      section 3) and message (represented as an octet string) that may
      be used to verify the integrity of the associated message. This
      function is not required to return the same deterministic result
      for each use; it need only generate a token that the verify_mic
      routine can check.

      The output of this function will also dictate the size of the
      checksum. It must be no larger than 65,535 octets.

   verify_mic function
      Given a specific key, message, and MIC token, this function
      ascertains whether the message integrity has been compromised.
      For a deterministic get_mic routine, the corresponding verify_mic
      may simply generate another checksum and compare the two.

   The get_mic and verify_mic operations must allow inputs of arbitrary
   length; if any padding is needed, the padding scheme must be
   specified as part of these functions.

   These operations and attributes are all that should be required to
   support Kerberos and various proposed preauthentication schemes.

   As with encryption mechanism definition documents, documents defining
   new checksum mechanisms should indicate validation processes and
   known weaknesses.

5. Simplified Profile for CBC Ciphers with Key Derivation

   The profile outlined in sections 3 and 4 describes a large number of
   operations that must be defined for encryption and checksum
   algorithms to be used with Kerberos. Here we describe a simpler
   profile that can generate both encryption and checksum mechanism
   definitions, filling in uses of key derivation in appropriate places,
   providing integrity protection, and defining multiple operations for
   the cryptosystem profile based on a smaller set of operations. Not
   all of the existing cryptosystems for Kerberos fit into this
   simplified profile, but we recommend that future cryptosystems use it
   or something based on it [4].

   Not all the operations in the complete profiles are defined through
   this mechanism; several must still be defined for each new algorithm
   pair.

5.1. A Key Derivation Function

   Rather than define some scheme by which a "protocol key" is composed
   of a large number of encryption keys, we use keys derived from a base
   key to perform cryptographic operations. The base key must be used
   only for generating the derived keys, and this derivation must be
   non-invertible and entropy preserving. Given these restrictions,
   compromise of one derived key does not compromise others. Attack of
   the base key is limited, as it is only used for derivation and is not
   exposed to any user data.

   To generate a derived key from a base key, we generate a pseudorandom
   octet string by using an algorithm DR, described below, and generate
   a key from that octet string by using a function dependent on the
   encryption algorithm. The input length needed for that function,
   which is also dependent on the encryption algorithm, dictates the
   length of the string to be generated by the DR algorithm (the value
   "k" below). These procedures are based on the key derivation in
   [Blumenthal96].

       Derived Key = DK(Base Key, Well-Known Constant)

       DK(Key, Constant) = random-to-key(DR(Key, Constant))

       DR(Key, Constant) = k-truncate(E(Key, Constant,
                                         initial-cipher-state))

   Here DR is the random-octet generation function described below, and
   DK is the key-derivation function produced from it. In this
   construction, E(Key, Plaintext, CipherState) is a cipher, Constant is
   a well-known constant determined by the specific usage of this

function, and k-truncate truncates its argument by taking the first k
bits. Here, k is the key generation seed length needed for the
encryption system.

The output of the DR function is a string of bits; the actual key is
produced by applying the cryptosystem's random-to-key operation on
this bitstring.

If the Constant is smaller than the cipher block size of E, then it
must be expanded with n-fold() so it can be encrypted. If the output
of E is shorter than k bits, it is fed back into the encryption as
many times as necessary. The construct is as follows (where |
indicates concatenation):

    K1 = E(Key, n-fold(Constant), initial-cipher-state)
    K2 = E(Key, K1, initial-cipher-state)
    K3 = E(Key, K2, initial-cipher-state)
    K4 = ...

    DR(Key, Constant) = k-truncate(K1 | K2 | K3 | K4 ...)

n-fold is an algorithm that takes m input bits and "stretches" them
to form n output bits with equal contribution from each input bit to
the output, as described in [Blumenthal96]:

    We first define a primitive called n-folding, which takes a
    variable-length input block and produces a fixed-length output
    sequence. The intent is to give each input bit approximately
    equal weight in determining the value of each output bit. Note
    that whenever we need to treat a string of octets as a number, the
    assumed representation is Big-Endian -- Most Significant Byte
    first.

    To n-fold a number X, replicate the input value to a length that
    is the least common multiple of n and the length of X. Before
    each repetition, the input is rotated to the right by 13 bit
    positions. The successive n-bit chunks are added together using
    1's-complement addition (that is, with end-around carry) to yield
    a n-bit result....

Test vectors for n-fold are supplied in appendix A [5].

In this section, n-fold is always used to produce c bits of output,
where c is the cipher block size of E.

The size of the Constant must not be larger than c, because reducing
the length of the Constant by n-folding can cause collisions.

If the size of the Constant is smaller than c, then the Constant must
be n-folded to length c. This string is used as input to E. If the
block size of E is less than the random-to-key input size, then the
output from E is taken as input to a second invocation of E. This
process is repeated until the number of bits accumulated is greater
than or equal to the random-to-key input size. When enough bits have
been computed, the first k are taken as the random data used to
create the key with the algorithm-dependent random-to-key function.

As the derived key is the result of one or more encryptions in the
base key, deriving the base key from the derived key is equivalent to
determining the key from a very small number of plaintext/ciphertext
pairs. Thus, this construction is as strong as the cryptosystem
itself.

5.2. Simplified Profile Parameters

These are the operations and attributes that must be defined:

protocol key format
string-to-key function
default string-to-key parameters
key-generation seed length, k
random-to-key function
     As above for the normal encryption mechanism profile.

unkeyed hash algorithm, H
     This should be a collision-resistant hash algorithm with fixed-
     size output, suitable for use in an HMAC [HMAC]. It must support
     inputs of arbitrary length. Its output must be at least the
     message block size (below).

HMAC output size, h
     This indicates the size of the leading substring output by the
     HMAC function that should be used in transmitted messages. It
     should be at least half the output size of the hash function H,
     and at least 80 bits; it need not match the output size.

message block size, m
     This is the size of the smallest units the cipher can handle in
     the mode in which it is being used. Messages will be padded to a
     multiple of this size. If a block cipher is used in a mode that

can handle messages that are not multiples of the cipher block
size, such as CBC mode with cipher text stealing (CTS, see [RC5]),
this value would be one octet. For traditional CBC mode with
padding, it would be the underlying cipher's block size.

This value must be a multiple of eight bits (one octet).

encryption/decryption functions, E and D
   These are basic encryption and decryption functions for messages
   of sizes that are multiples of the message block size. No
   integrity checking or confounder should be included here. For
   inputs these functions take the IV or similar data, a protocol-
   format key, and an octet string, returning a new IV and octet
   string.

   The encryption function is not required to use CBC mode but is
   assumed to be using something with similar properties. In
   particular, prepending a cipher block-size confounder to the
   plaintext should alter the entire ciphertext (comparable to
   choosing and including a random initial vector for CBC mode).

   The result of encrypting one cipher block (of size c, above) must
   be deterministic for the random octet generation function DR in
   the previous section to work. For best security, it should also
   be no larger than c.

cipher block size, c
   This is the block size of the block cipher underlying the
   encryption and decryption functions indicated above, used for key
   derivation and for the size of the message confounder and initial
   vector. (If a block cipher is not in use, some comparable
   parameter should be determined.) It must be at least 5 octets.

   This is not actually an independent parameter; rather, it is a
   property of the functions E and D. It is listed here to clarify
   the distinction between it and the message block size, m.

Although there are still a number of properties to specify, they are
fewer and simpler than in the full profile.

5.3. Cryptosystem Profile Based on Simplified Profile

The above key derivation function is used to produce three
intermediate keys. One is used for computing checksums of
unencrypted data. The other two are used for encrypting and
checksumming plaintext to be sent encrypted.

The ciphertext output is the concatenation of the output of the basic
encryption function E and a (possibly truncated) HMAC using the
specified hash function H, both applied to the plaintext with a
random confounder prefix and sufficient padding to bring it to a
multiple of the message block size. When the HMAC is computed, the
key is used in the protocol key form.

Decryption is performed by removing the (partial) HMAC, decrypting
the remainder, and verifying the HMAC. The cipher state is an
initial vector, initialized to zero.

The substring notation "[1..h]" in the following table should be read
as using 1-based indexing; leading substrings are used.

                    Cryptosystem from Simplified Profile
----
protocol key format   As given.

specific key structure   Three protocol-format keys: { Kc, Ke, Ki }.

key-generation seed   As given.
length

required checksum   As defined below in section 5.4.
mechanism

cipher state   Initial vector (usually of length c)

initial cipher state   All bits zero

encryption function   conf = Random string of length c
                           pad = Shortest string to bring confounder
                                   and plaintext to a length that's a
                                   multiple of m.
                           (C1, newIV) = E(Ke, conf | plaintext | pad,
                                             oldstate.ivec)
                           H1 = HMAC(Ki, conf | plaintext | pad)
                           ciphertext = C1 | H1[1..h]
                           newstate.ivec = newIV

decryption function   (C1,H1) = ciphertext
                           (P1, newIV) = D(Ke, C1, oldstate.ivec)
                           if (H1 != HMAC(Ki, P1)[1..h])
                              report error
                           newstate.ivec = newIV

default string-to-key   As given.
params

pseudo-random function   tmp1 = H(octet-string)
                           tmp2 = truncate tmp1 to multiple of m
                           PRF = E(DK(protocol-key, prfconstant),
                                   tmp2, initial-cipher-state)

   The "prfconstant" used in the PRF operation is the three-octet string
   "prf".

                    Cryptosystem from Simplified Profile
----
key generation functions:

string-to-key function   As given.

random-to-key function   As given.

key-derivation function   The "well-known constant" used for the DK
                            function is the key usage number, expressed as
                            four octets in big-endian order, followed by
                            one octet indicated below.

                            Kc = DK(base-key, usage | 0x99);
                            Ke = DK(base-key, usage | 0xAA);
                            Ki = DK(base-key, usage | 0x55);

5.4. Checksum Profiles Based on Simplified Profile

   When an encryption system is defined with the simplified profile
   given in section 5.2, a checksum algorithm may be defined for it as
   follows:

              Checksum Mechanism from Simplified Profile
              ----
            associated cryptosystem   As defined above.

            get_mic   HMAC(Kc, message)[1..h]

            verify_mic   get_mic and compare

   The HMAC function and key Kc are as described in section 5.3.

6. Profiles for Kerberos Encryption and Checksum Algorithms

   These profiles describe the encryption and checksum systems defined
   for Kerberos. The astute reader will notice that some of them do not
   fulfill all the requirements outlined in previous sections. These
   systems are defined for backward compatibility; newer implementations
   should (whenever possible) attempt to utilize encryption systems that
   satisfy all the profile requirements.

   The full list of current encryption and checksum type number
   assignments, including values currently reserved but not defined in
   this document, is given in section 8.

6.1. Unkeyed Checksums

   These checksum types use no encryption keys and thus can be used in
   combination with any encryption type, but they may only be used with
   caution, in limited circumstances where the lack of a key does not
   provide a window for an attack, preferably as part of an encrypted
   message [6]. Keyed checksum algorithms are recommended.

6.1.1. The RSA MD5 Checksum

   The RSA-MD5 checksum calculates a checksum by using the RSA MD5
   algorithm [MD5-92]. The algorithm takes as input an input message of
   arbitrary length and produces as output a 128-bit (sixteen octet)
   checksum.

                              rsa-md5
               ----
               associated cryptosystem   any

               get_mic   rsa-md5(msg)

               verify_mic   get_mic and compare

   The rsa-md5 checksum algorithm is assigned a checksum type number of
   seven (7).

6.1.2. The RSA MD4 Checksum

   The RSA-MD4 checksum calculates a checksum using the RSA MD4
   algorithm [MD4-92]. The algorithm takes as input an input message of
   arbitrary length and produces as output a 128-bit (sixteen octet)
   checksum.

                              rsa-md4
               ----
               associated cryptosystem   any

               get_mic   md4(msg)

               verify_mic   get_mic and compare

   The rsa-md4 checksum algorithm is assigned a checksum type number of
   two (2).

6.1.3. CRC-32 Checksum

   This CRC-32 checksum calculates a checksum based on a cyclic
   redundancy check as described in ISO 3309 [CRC] but modified as
   described below. The resulting checksum is four (4) octets in
   length. The CRC-32 is neither keyed nor collision-proof; thus, the
   use of this checksum is not recommended. An attacker using a
   probabilistic chosen-plaintext attack as described in [SG92] might be
   able to generate an alternative message that satisfies the checksum.

   The CRC-32 checksum used in the des-cbc-crc encryption mode is
   identical to the 32-bit FCS described in ISO 3309 with two
   exceptions: The sum with the all-ones polynomial times x**k is
   omitted, and the final remainder is not ones-complemented. ISO 3309
   describes the FCS in terms of bits, whereas this document describes
   the Kerberos protocol in terms of octets. To clarify the ISO 3309
   definition for the purpose of computing the CRC-32 in the des-cbc-crc
   encryption mode, the ordering of bits in each octet shall be assumed
   to be LSB first. Given this assumed ordering of bits within an
   octet, the mapping of bits to polynomial coefficients shall be
   identical to that specified in ISO 3309.

   Test values for this modified CRC function are included in appendix
   A.5.

```
                                crc32
            ---------------------------------------------
            associated cryptosystem   any

            get_mic                   crc32(msg)

            verify_mic                get_mic and compare
```

   The crc32 checksum algorithm is assigned a checksum type number of
   one (1).

6.2. DES-Based Encryption and Checksum Types

   These encryption systems encrypt information under the Data
   Encryption Standard [DES77] by using the cipher block chaining mode
   [DESM80]. A checksum is computed as described below and placed in
   the cksum field. DES blocks are eight bytes. As a result, the data
   to be encrypted (the concatenation of confounder, checksum, and
   message) must be padded to an eight byte boundary before encryption.
   The values of the padding bytes are unspecified.

Plaintext and DES ciphertext are encoded as blocks of eight octets,
which are concatenated to make the 64-bit inputs for the DES
algorithms. The first octet supplies the eight most significant bits
(with the octet's MSB used as the DES input block's MSB, etc.), the
second octet the next eight bits, and so on. The eighth octet
supplies the 8 least significant bits.

Encryption under DES using cipher block chaining requires an
additional input in the form of an initialization vector; this vector
is specified below for each encryption system.

The DES specifications [DESI81] identify four 'weak' and twelve
'semi-weak' keys; these keys SHALL NOT be used for encrypting
messages for use in Kerberos. The "variant keys" generated for the
RSA-MD5-DES, RSA-MD4-DES, and DES-MAC checksum types by an
eXclusive-OR of a DES key with a constant are not checked for this
property.

A DES key is eight octets of data. This consists of 56 bits of
actual key data, and eight parity bits, one per octet. The key is
encoded as a series of eight octets written in MSB-first order. The
bits within the key are also encoded in MSB order. For example, if
the encryption key is
$(B1,B2,...,B7,P1,B8,...,B14,P2,B15,...,B49,P7,B50,...,B56,P8)$, where
$B1,B2,...,B56$ are the key bits in MSB order, and $P1,P2,...,P8$ are the
parity bits, the first octet of the key would be $B1,B2,...,B7,P1$
(with B1 as the most significant bit). See the [DESM80] introduction
for reference.

Encryption Data Format

The format for the data to be encrypted includes a one-block
confounder, a checksum, the encoded plaintext, and any necessary
padding, as described in the following diagram. The msg-seq field
contains the part of the protocol message to be encrypted.

```
        +-----------+----------+---------+-----+
        |confounder | checksum | msg-seq | pad |
        +-----------+----------+---------+-----+
```

One generates a random confounder of one block, placing it in
'confounder'; zeros out the 'checksum' field (of length appropriate
to exactly hold the checksum to be computed); adds the necessary
padding; calculates the appropriate checksum over the whole sequence,
placing the result in 'checksum'; and then encrypts using the
specified encryption type and the appropriate key.

String or Random-Data to Key Transformation

To generate a DES key from two UTF-8 text strings (password and
salt), the two strings are concatenated, password first, and the
result is then padded with zero-valued octets to a multiple of eight
octets.

The top bit of each octet (always zero if the password is plain
ASCII, as was assumed when the original specification was written) is
discarded, and the remaining seven bits of each octet form a
bitstring. This is then fan-folded and eXclusive-ORed with itself to
produce a 56-bit string. An eight-octet key is formed from this
string, each octet using seven bits from the bitstring, leaving the
least significant bit unassigned. The key is then "corrected" by
correcting the parity on the key, and if the key matches a 'weak' or
'semi-weak' key as described in the DES specification, it is
eXclusive-ORed with the constant 0x00000000000000F0. This key is
then used to generate a DES CBC checksum on the initial string with
the salt appended. The result of the CBC checksum is then
"corrected" as described above to form the result, which is returned
as the key.

For purposes of the string-to-key function, the DES CBC checksum is
calculated by CBC encrypting a string using the key as IV and the
final eight byte block as the checksum.

Pseudocode follows:

```
    removeMSBits(8byteblock) {
      /* Treats a 64 bit block as 8 octets and removes the MSB in
         each octet (in big endian mode) and concatenates the
         result.  E.g., the input octet string:
             01110000 01100001 11110011  01110011 11110111 01101111
             11110010 01100100
         results in the output bitstring:
             1110000 1100001 1110011  1110011 1110111 1101111
             1110010 1100100  */
    }

    reverse(56bitblock) {
      /* Treats a 56-bit block as a binary string and reverses it.
         E.g., the input string:
             1000001 1010100 1001000  1000101 1001110 1000001
             0101110 1001101
         results in the output string:
             1011001 0111010 1000001  0111001 1010001 0001001
             0010101 1000001  */
    }
```

```
add_parity_bits(56bitblock) {
  /* Copies a 56-bit block into a 64-bit block, left shifts
     content in each octet, and add DES parity bit.
     E.g., the input string:
        1100000 0001111 0011100  0110100 1000101 1100100
        0110110 0010111
     results in the output string:
        11000001 00011111 00111000  01101000 10001010 11001000
        01101101 00101111  */
}

key_correction(key) {
     fixparity(key);
     if (is_weak_key(key))
          key = key XOR 0xF0;
     return(key);
}

mit_des_string_to_key(string,salt) {
     odd = 1;
     s = string | salt;
     tempstring = 0; /* 56-bit string */
     pad(s); /* with nulls to 8 byte boundary */
     for (8byteblock in s) {
          56bitstring = removeMSBits(8byteblock);
          if (odd == 0) reverse(56bitstring);
          odd = ! odd;
          tempstring = tempstring XOR 56bitstring;
     }
     tempkey = key_correction(add_parity_bits(tempstring));
     key = key_correction(DES-CBC-check(s,tempkey));
     return(key);
}

des_string_to_key(string,salt,params) {
     if (length(params) == 0)
          type = 0;
     else if (length(params) == 1)
          type = params[0];
     else
          error("invalid params");
     if (type == 0)
          mit_des_string_to_key(string,salt);
     else
          error("invalid params");
}
```

One common extension is to support the "AFS string-to-key" algorithm, which is not defined here, if the type value above is one (1).

For generation of a key from a random bitstring, we start with a 56-bit string and, as with the string-to-key operation above, insert parity bits. If the result is a weak or semi-weak key, we modify it by eXclusive-OR with the constant 0x00000000000000F0:

```
    des_random_to_key(bitstring) {
         return key_correction(add_parity_bits(bitstring));
    }
```

6.2.1. DES with MD5

The des-cbc-md5 encryption mode encrypts information under DES in CBC mode with an all-zero initial vector and with an MD5 checksum (described in [MD5-92]) computed and placed in the checksum field.

The encryption system parameters for des-cbc-md5 are as follows:

```
                            des-cbc-md5
   -------------------------------------------------------------------
   protocol key format       8 bytes, parity in low bit of each

   specific key structure    copy of original key

   required checksum         rsa-md5-des
   mechanism

   key-generation seed       8 bytes
   length

   cipher state              8 bytes (CBC initial vector)

   initial cipher state      all-zero

   encryption function       des-cbc(confounder | checksum | msg | pad,
                                     ivec=oldstate)
                             where
                             checksum = md5(confounder | 0000...
                                             | msg | pad)

                             newstate = last block of des-cbc output

   decryption function       decrypt encrypted text and verify checksum

                             newstate = last block of ciphertext
```

```
                            des-cbc-md5
   ----------------------------------------------------------------
   default string-to-key     empty string
   params

   pseudo-random function    des-cbc(md5(input-string), ivec=0)

   key generation functions:

   string-to-key             des_string_to_key

   random-to-key             des_random_to_key

   key-derivation            identity
```

   The des-cbc-md5 encryption type is assigned the etype value three
   (3).

6.2.2. DES with MD4

   The des-cbc-md4 encryption mode also encrypts information under DES
   in CBC mode, with an all-zero initial vector. An MD4 checksum
   (described in [MD4-92]) is computed and placed in the checksum field.

```
                            des-cbc-md4
   ----------------------------------------------------------------
   protocol key format       8 bytes, parity in low bit of each

   specific key structure    copy of original key

   required checksum         rsa-md4-des
   mechanism

   key-generation seed       8 bytes
   length

   cipher state              8 bytes (CBC initial vector)

   initial cipher state      all-zero

   encryption function       des-cbc(confounder | checksum | msg | pad,
                                     ivec=oldstate)
                             where
                             checksum = md4(confounder | 0000...
                                            | msg | pad)

                             newstate = last block of des-cbc output
```

                          des-cbc-md4
     ----

     decryption function   decrypt encrypted text and verify checksum

                                newstate = last block of ciphertext

     default string-to-key   empty string
     params

     pseudo-random function   des-cbc(md5(input-string), ivec=0)

     key generation functions:

     string-to-key   des_string_to_key

     random-to-key   copy input, then fix parity bits

     key-derivation   identity

     Note that des-cbc-md4 uses md5, not md4, in the PRF definition.

     The des-cbc-md4 encryption algorithm is assigned the etype value two
     (2).

6.2.3. DES with CRC

     The des-cbc-crc encryption type uses DES in CBC mode with the key
     used as the initialization vector, with a four-octet CRC-based
     checksum computed as described in section 6.1.3. Note that this is
     not a standard CRC-32 checksum, but a slightly modified one.

                          des-cbc-crc
     ----
     protocol key format   8 bytes, parity in low bit of each

     specific key structure   copy of original key

     required checksum   rsa-md5-des
     mechanism

     key-generation seed   8 bytes
     length

     cipher state   8 bytes (CBC initial vector)

```
                              des-cbc-crc
    ----------------------------------------------------------------
    initial cipher state   copy of original key

    encryption function    des-cbc(confounder | checksum | msg | pad,
                                   ivec=oldstate)
                           where
                           checksum = crc(confounder | 00000000
                                               | msg | pad)

                           newstate = last block of des-cbc output

    decryption function    decrypt encrypted text and verify checksum

                           newstate = last block of ciphertext

    default string-to-key  empty string
    params

    pseudo-random function des-cbc(md5(input-string), ivec=0)

    key generation functions:

    string-to-key          des_string_to_key

    random-to-key          copy input, then fix parity bits

    key-derivation         identity
```

The des-cbc-crc encryption algorithm is assigned the etype value one
(1).

6.2.4. RSA MD5 Cryptographic Checksum Using DES

The RSA-MD5-DES checksum calculates a keyed collision-proof checksum
by prepending an eight octet confounder before the text, applying the
RSA MD5 checksum algorithm, and encrypting the confounder and the
checksum by using DES in cipher-block-chaining (CBC) mode with a
variant of the key, where the variant is computed by eXclusive-ORing
the key with the hexadecimal constant 0xF0F0F0F0F0F0F0F0. The
initialization vector should be zero. The resulting checksum is 24
octets long.

```
                                  rsa-md5-des
          ----------------------------------------------------------------
          associated cryptosystem  des-cbc-md5, des-cbc-md4, des-cbc-crc

          get_mic                  des-cbc(key XOR 0xF0F0F0F0F0F0F0F0,
                                           conf | rsa-md5(conf | msg))

          verify_mic               decrypt and verify rsa-md5 checksum
```

   The rsa-md5-des checksum algorithm is assigned a checksum type number
   of eight (8).

6.2.5. RSA MD4 Cryptographic Checksum Using DES

   The RSA-MD4-DES checksum calculates a keyed collision-proof checksum
   by prepending an eight octet confounder before the text, applying the
   RSA MD4 checksum algorithm [MD4-92], and encrypting the confounder
   and the checksum using DES in cipher-block-chaining (CBC) mode with a
   variant of the key, where the variant is computed by eXclusive-ORing
   the key with the constant 0xF0F0F0F0F0F0F0F0 [7]. The initialization
   vector should be zero. The resulting checksum is 24 octets long.

```
                                  rsa-md4-des
          ----------------------------------------------------------------
          associated cryptosystem  des-cbc-md5, des-cbc-md4, des-cbc-crc

          get_mic                  des-cbc(key XOR 0xF0F0F0F0F0F0F0F0,
                                           conf | rsa-md4(conf | msg),
                                           ivec=0)

          verify_mic               decrypt and verify rsa-md4 checksum
```

   The rsa-md4-des checksum algorithm is assigned a checksum type number
   of three (3).

6.2.6. RSA MD4 Cryptographic Checksum Using DES Alternative

   The RSA-MD4-DES-K checksum calculates a keyed collision-proof
   checksum by applying the RSA MD4 checksum algorithm and encrypting
   the results by using DES in cipher block chaining (CBC) mode with a
   DES key as both key and initialization vector. The resulting
   checksum is 16 octets long. This checksum is tamper-proof and
   believed to be collision-proof. Note that this checksum type is the
   old method for encoding the RSA-MD4-DES checksum; it is no longer
   recommended.

                              rsa-md4-des-k
      ----
      associated cryptosystem   des-cbc-md5, des-cbc-md4, des-cbc-crc

      get_mic   des-cbc(key, md4(msg), ivec=key)

      verify_mic   decrypt, compute checksum and compare

   The rsa-md4-des-k checksum algorithm is assigned a checksum type
   number of six (6).

6.2.7. DES CBC Checksum

   The DES-MAC checksum is computed by prepending an eight octet
   confounder to the plaintext, padding with zero-valued octets if
   necessary to bring the length to a multiple of eight octets,
   performing a DES CBC-mode encryption on the result by using the key
   and an initialization vector of zero, taking the last block of the
   ciphertext, prepending the same confounder, and encrypting the pair
   by using DES in cipher-block-chaining (CBC) mode with a variant of
   the key, where the variant is computed by eXclusive-ORing the key
   with the constant 0xF0F0F0F0F0F0F0F0. The initialization vector
   should be zero. The resulting checksum is 128 bits (sixteen octets)
   long, 64 bits of which are redundant. This checksum is tamper-proof
   and collision-proof.

                              des-mac
   ----
   associated   des-cbc-md5, des-cbc-md4, des-cbc-crc
   cryptosystem

   get_mic   des-cbc(key XOR 0xF0F0F0F0F0F0F0F0,
                           conf | des-mac(key, conf | msg | pad, ivec=0),
                           ivec=0)

   verify_mic   decrypt, compute DES MAC using confounder, compare

   The des-mac checksum algorithm is assigned a checksum type number of
   four (4).

6.2.8. DES CBC Checksum Alternative

   The DES-MAC-K checksum is computed by performing a DES CBC-mode
   encryption of the plaintext, with zero-valued padding bytes if
   necessary to bring the length to a multiple of eight octets, and by
   using the last block of the ciphertext as the checksum value. It is
   keyed with an encryption key that is also used as the initialization
   vector. The resulting checksum is 64 bits (eight octets) long. This

checksum is tamper-proof and collision-proof. Note that this
checksum type is the old method for encoding the DESMAC checksum; it
is no longer recommended.

```
                              des-mac-k
      ----------------------------------------------------------------
      associated cryptosystem   des-cbc-md5, des-cbc-md4, des-cbc-crc

      get_mic                   des-mac(key, msg | pad, ivec=key)

      verify_mic                compute MAC and compare
```

The des-mac-k checksum algorithm is assigned a checksum type number
of five (5).

6.3. Triple-DES Based Encryption and Checksum Types

This encryption and checksum type pair is based on the Triple DES
cryptosystem in Outer-CBC mode and on the HMAC-SHA1 message
authentication algorithm.

A Triple DES key is the concatenation of three DES keys as described
above for des-cbc-md5. A Triple DES key is generated from random
data by creating three DES keys from separate sequences of random
data.

Encrypted data using this type must be generated as described in
section 5.3. If the length of the input data is not a multiple of
the block size, zero-valued octets must be used to pad the plaintext
to the next eight-octet boundary. The confounder must be eight
random octets (one block).

The simplified profile for Triple DES, with key derivation as defined
in section 5, is as follows:

```
            des3-cbc-hmac-sha1-kd, hmac-sha1-des3-kd
            ------------------------------------------------
            protocol key format    24 bytes, parity in low
                                   bit of each

            key-generation seed    21 bytes
            length
```

                        des3-cbc-hmac-sha1-kd, hmac-sha1-des3-kd
            ----
            hash function   SHA-1

            HMAC output size   160 bits

            message block size   8 bytes

            default string-to-key   empty string
            params

            encryption and   triple-DES encrypt and
            decryption functions   decrypt, in outer-CBC
                                    mode (cipher block size
                                    8 octets)

            key generation functions:

            random-to-key   DES3random-to-key (see
                                    below)

            string-to-key   DES3string-to-key (see
                                    below)

   The des3-cbc-hmac-sha1-kd encryption type is assigned the value
   sixteen (16). The hmac-sha1-des3-kd checksum algorithm is assigned a
   checksum type number of twelve (12).

6.3.1. Triple DES Key Production (random-to-key, string-to-key)

   The 168 bits of random key data are converted to a protocol key value
   as follows. First, the 168 bits are divided into three groups of 56
   bits, which are expanded individually into 64 bits as follows:

   DES3random-to-key:
         1 2 3 4 5 6 7 p
         9 10 11 12 13 14 15 p
        17 18 19 20 21 22 23 p
        25 26 27 28 29 30 31 p
        33 34 35 36 37 38 39 p
        41 42 43 44 45 46 47 p
        49 50 51 52 53 54 55 p
        56 48 40 32 24 16 8 p

   The "p" bits are parity bits computed over the data bits. The output
   of the three expansions, each corrected to avoid "weak" and "semi-
   weak" keys as in section 6.2, are concatenated to form the protocol
   key value.

The string-to-key function is used to transform UTF-8 passwords into
DES3 keys. The DES3 string-to-key function relies on the "N-fold"
algorithm and DK function, described in section 5.

The n-fold algorithm is applied to the password string concatenated
with a salt value. For 3-key triple DES, the operation will involve
a 168-fold of the input password string, to generate an intermediate
key, from which the user's long-term key will be derived with the DK
function. The DES3 string-to-key function is shown here in
pseudocode:

```
    DES3string-to-key(passwordString, salt, params)
        if (params != emptyString)
         error("invalid params");
        s = passwordString + salt
        tmpKey = random-to-key(168-fold(s))
        key = DK (tmpKey, KerberosConstant)
```

Weak key checking is performed in the random-to-key and DK
operations. The KerberosConstant value is the byte string {0x6b 0x65
0x72 0x62 0x65 0x72 0x6f 0x73}. These values correspond to the ASCII
encoding for the string "kerberos".

7. Use of Kerberos Encryption Outside This Specification

Several Kerberos-based application protocols and preauthentication
systems have been designed and deployed that perform encryption and
message integrity checks in various ways. Although in some cases
there may be good reason for specifying these protocols in terms of
specific encryption or checksum algorithms, we anticipate that in
many cases this will not be true, and more generic approaches
independent of particular algorithms will be desirable. Rather than
have each protocol designer reinvent schemes for protecting data,
using multiple keys, etc., we have attempted to present in this
section a general framework that should be sufficient not only for
the Kerberos protocol itself but also for many preauthentication
systems and application protocols, while trying to avoid some of the
assumptions that can work their way into such protocol designs.

Some problematic assumptions we've seen (and sometimes made) include
the following: a random bitstring is always valid as a key (not true
for DES keys with parity); the basic block encryption chaining mode
provides no integrity checking, or can easily be separated from such
checking (not true for many modes in development that do both
simultaneously); a checksum for a message always results in the same
value (not true if a confounder is incorporated); an initial vector
is used (may not be true if a block cipher in CBC mode is not in
use).

Although such assumptions the may hold for any given set of
encryption and checksum algorithms, they may not be true of the next
algorithms to be defined, leaving the application protocol unable to
make use of those algorithms without updates to its specification.

The Kerberos protocol uses only the attributes and operations
described in sections 3 and 4. Preauthentication systems and
application protocols making use of Kerberos are encouraged to use
them as well. The specific key and string-to-key parameters should
generally be treated as opaque. Although the string-to-key
parameters are manipulated as an octet string, the representation for
the specific key structure is implementation defined; it may not even
be a single object.

We don't recommend doing so, but some application protocols will
undoubtedly continue to use the key data directly, even if only in
some of the currently existing protocol specifications. An
implementation intended to support general Kerberos applications may
therefore need to make the key data available, as well as the
attributes and operations described in sections 3 and 4 [8].

8. Assigned Numbers

The following encryption-type numbers are already assigned or
reserved for use in Kerberos and related protocols.

encryption type	etype	section or comment
des-cbc-crc	1	6.2.3
des-cbc-md4	2	6.2.2
des-cbc-md5	3	6.2.1
[reserved]	4	
des3-cbc-md5	5	
[reserved]	6	
des3-cbc-sha1	7	
dsaWithSHA1-CmsOID	9	(pkinit)
md5WithRSAEncryption-CmsOID	10	(pkinit)
sha1WithRSAEncryption-CmsOID	11	(pkinit)
rc2CBC-EnvOID	12	(pkinit)
rsaEncryption-EnvOID	13	(pkinit from PKCS#1 v1.5)
rsaES-OAEP-ENV-OID	14	(pkinit from PKCS#1 v2.0)
des-ede3-cbc-Env-OID	15	(pkinit)
des3-cbc-sha1-kd	16	6.3
aes128-cts-hmac-sha1-96	17	[KRB5-AES]
aes256-cts-hmac-sha1-96	18	[KRB5-AES]
rc4-hmac	23	(Microsoft)
rc4-hmac-exp	24	(Microsoft)
subkey-keymaterial	65	(opaque; PacketCable)

(The "des3-cbc-sha1" assignment is a deprecated version using no key derivation. It should not be confused with des3-cbc-sha1-kd.)

Several numbers have been reserved for use in encryption systems not defined here. Encryption-type numbers have unfortunately been overloaded on occasion in Kerberos-related protocols, so some of the reserved numbers do not and will not correspond to encryption systems fitting the profile presented here.

The following checksum-type numbers are assigned or reserved. As with encryption-type numbers, some overloading of checksum numbers has occurred.

Checksum type	sumtype value	checksum size	section or reference
CRC32	1	4	6.1.3
rsa-md4	2	16	6.1.2
rsa-md4-des	3	24	6.2.5
des-mac	4	16	6.2.7
des-mac-k	5	8	6.2.8
rsa-md4-des-k	6	16	6.2.6
rsa-md5	7	16	6.1.1
rsa-md5-des	8	24	6.2.4
rsa-md5-des3	9	24	??
sha1 (unkeyed)	10	20	??
hmac-sha1-des3-kd	12	20	6.3
hmac-sha1-des3	13	20	??
sha1 (unkeyed)	14	20	??
hmac-sha1-96-aes128	15	20	[KRB5-AES]
hmac-sha1-96-aes256	16	20	[KRB5-AES]
[reserved]	0x8003	?	[GSS-KRB5]

Encryption and checksum-type numbers are signed 32-bit values. Zero is invalid, and negative numbers are reserved for local use. All standardized values must be positive.

9. Implementation Notes

The "interface" described here is the minimal information that must be defined to make a cryptosystem useful within Kerberos in an interoperable fashion. The use of functional notation used in some places is not an attempt to define an API for cryptographic functionality within Kerberos. Actual implementations providing clean APIs will probably make additional information available, that could be derived from a specification written to the framework given here. For example, an application designer may wish to determine the largest number of bytes that can be encrypted without overflowing a

certain size output buffer or conversely, the maximum number of bytes
that might be obtained by decrypting a ciphertext message of a given
size. (In fact, an implementation of the GSS-API Kerberos mechanism
[GSS-KRB5] will require some of these.)

The presence of a mechanism in this document should not be taken to
indicate that it must be implemented for compliance with any
specification; required mechanisms will be specified elsewhere.
Indeed, some of the mechanisms described here for backward
compatibility are now considered rather weak for protecting critical
data.

10. Security Considerations

Recent years have brought so many advancements in large-scale attacks
capability against DES that it is no longer considered a strong
encryption mechanism. Triple-DES is generally preferred in its
place, despite its poorer performance. See [ESP-DES] for a summary
of some of the potential attacks and [EFF-DES] for a detailed
discussion of the implementation of particular attacks. However,
most Kerberos implementations still have DES as their primary
interoperable encryption type.

DES has four 'weak' keys and twelve 'semi-weak' keys, and the use of
single-DES here avoids them. However, DES also has 48 'possibly-
weak' keys [Schneier96] (note that the tables in many editions of the
reference contains errors) that are not avoided.

DES weak keys have the property that E1(E1(P)) = P (where E1 denotes
encryption of a single block with key 1). DES semi-weak keys, or
"dual" keys, are pairs of keys with the property that E1(P) = D2(P),
and thus E2(E1(P)) = P. Because of the use of CBC mode and the
leading random confounder, however, these properties are unlikely to
present a security problem.

Many of the choices concerning when to perform weak-key corrections
relate more to compatibility with existing implementations than to
any risk analysis.

Although checks are also done for the component DES keys in a
triple-DES key, the nature of the weak keys make it extremely
unlikely that they will weaken the triple-DES encryption. It is only
slightly more likely than having the middle of the three sub-keys
match one of the other two, which effectively converts the encryption
to single-DES - a case we make no effort to avoid.

The true CRC-32 checksum is not collision-proof; an attacker could
use a probabilistic chosen-plaintext attack to generate a valid
message even if a confounder is used [SG92]. The use of collision-
proof checksums is of course recommended for environments where such
attacks represent a significant threat. The "simplifications" (read:
bugs) introduced when CRC-32 was implemented for Kerberos cause
leading zeros effectively to be ignored, so messages differing only
in leading zero bits will have the same checksum.

[HMAC] and [IPSEC-HMAC] discuss weaknesses of the HMAC algorithm.
Unlike [IPSEC-HMAC], the triple-DES specification here does not use
the suggested truncation of the HMAC output. As pointed out in
[IPSEC-HMAC], SHA-1 was not developed for use as a keyed hash
function, which is a criterion of HMAC. [HMAC-TEST] contains test
vectors for HMAC-SHA-1.

The mit_des_string_to_key function was originally constructed with
the assumption that all input would be ASCII; it ignores the top bit
of each input byte. Folding with XOR is also not an especially good
mixing mechanism for preserving randomness.

The n-fold function used in the string-to-key operation for des3-
cbc-hmac-sha1-kd was designed to cause each bit of input to
contribute equally to the output. It was not designed to maximize or
equally distribute randomness in the input, and conceivably
randomness may be lost in cases of partially structured input. This
should only be an issue for highly structured passwords, however.

[RFC1851] discusses the relative strength of triple-DES encryption.
The relatively slow speed of triple-DES encryption may also be an
issue for some applications.

[Bellovin91] suggests that analyses of encryption schemes include a
model of an attacker capable of submitting known plaintexts to be
encrypted with an unknown key, as well as be able to perform many
types of operations on known protocol messages. Recent experiences
with the chosen-plaintext attacks on Kerberos version 4 bear out the
value of this suggestion.

The use of unkeyed encrypted checksums, such as those used in the
single-DES cryptosystems specified in [Kerb1510], allows for cut-
and-paste attacks, especially if a confounder is not used. In
addition, unkeyed encrypted checksums are vulnerable to chosen-
plaintext attacks: An attacker with access to an encryption oracle
can easily encrypt the required unkeyed checksum along with the

chosen plaintext. [Bellovin99] These weaknesses, combined with a
common implementation design choice described below, allow for a
cross-protocol attack from version 4 to version 5.

The use of a random confounder is an important means to prevent an
attacker from making effective use of protocol exchanges as an
encryption oracle. In Kerberos version 4, the encryption of constant
plaintext to constant ciphertext makes an effective encryption oracle
for an attacker. The use of random confounders in [Kerb1510]
frustrates this sort of chosen-plaintext attack.

Using the same key for multiple purposes can enable or increase the
scope of chosen-plaintext attacks. Some software that implements
both versions 4 and 5 of the Kerberos protocol uses the same keys for
both versions. This enables the encryption oracle of version 4 to be
used to attack version 5. Vulnerabilities to attacks such as this
cross-protocol attack make it unwise to use a key for multiple
purposes.

This document, like the Kerberos protocol, does not address limiting
the amount of data a key may be used with to a quantity based on the
robustness of the algorithm or size of the key. It is assumed that
any defined algorithms and key sizes will be strong enough to support
very large amounts of data, or they will be deprecated once
significant attacks are known.

This document also places no bounds on the amount of data that can be
handled in various operations. To avoid denial of service attacks,
implementations will probably seek to restrict message sizes at some
higher level.

11. IANA Considerations

Two registries for numeric values have been created: Kerberos
Encryption Type Numbers and Kerberos Checksum Type Numbers. These
are signed values ranging from -2147483648 to 2147483647. Positive
values should be assigned only for algorithms specified in accordance
with this specification for use with Kerberos or related protocols.
Negative values are for private use; local and experimental
algorithms should use these values. Zero is reserved and may not be
assigned.

Positive encryption- and checksum-type numbers may be assigned
following either of two policies described in [BCP26].

Standards-track specifications may be assigned values under the
Standards Action policy.

Specifications in non-standards track RFCs may be assigned values
after Expert Review. A non-IETF specification may be assigned values
by publishing an Informational or standards-track RFC referencing the
external specification; that specification must be public and
published in some permanent record, much like the IETF RFCs. It is
highly desirable, though not required, that the full specification be
published as an IETF RFC.

Smaller encryption type values should be used for IETF standards-
track mechanisms, and much higher values (16777216 and above) for
other mechanisms. (Rationale: In the Kerberos ASN.1 encoding,
smaller numbers encode to smaller octet sequences, so this favors
standards-track mechanisms with slightly smaller messages.) Aside
from that guideline, IANA may choose numbers as it sees fit.

Internet-Draft specifications should not include values for
encryption- and checksum-type numbers. Instead, they should indicate
that values would be assigned by IANA when the document is approved
as an RFC. For development and interoperability testing, values in
the private-use range (negative values) may be used but should not be
included in the draft specification.

Each registered value should have an associated unique reference
name. The lists given in section 8 were used to create the initial
registry; they include reservations for specifications in progress in
parallel with this document, and certain other values believed to
already be in use.

12. Acknowledgements

This document is an extension of the encryption specification
included in [Kerb1510] by B. Clifford Neuman and John Kohl, and much
of the text of the background, concepts, and DES specifications is
drawn directly from that document.

The abstract framework presented in this document was put together by
Jeff Altman, Sam Hartman, Jeff Hutzelman, Cliff Neuman, Ken Raeburn,
and Tom Yu, and the details were refined several times based on
comments from John Brezak and others.

Marc Horowitz wrote the original specification of triple-DES and key
derivation in a pair of Internet-Drafts (under the names draft-
horowitz-key-derivation and draft-horowitz-kerb-key-derivation) that
were later folded into a draft revision of [Kerb1510], from which
this document was later split off.

Tom Yu provided the text describing the modifications to the standard CRC algorithm as Kerberos implementations actually use it, and some of the text in the Security Considerations section.

Miroslav Jurisic provided information for one of the UTF-8 test cases for the string-to-key functions.

Marcus Watts noticed some errors in earlier versions and pointed out that the simplified profile could easily be modified to support cipher text stealing modes.

Simon Josefsson contributed some clarifications to the DES "CBC checksum" and string-to-key and weak key descriptions, and some test vectors.

Simon Josefsson, Louis LeVay, and others also caught some errors in earlier versions of this document.

A. Test Vectors

   This section provides test vectors for various functions defined or
   described in this document. For convenience, most inputs are ASCII
   strings, though some UTF-8 samples are provided for string-to-key
   functions. Keys and other binary data are specified as hexadecimal
   strings.

A.1. n-fold

   The n-fold function is defined in section 5.1. As noted there, the
   sample vector in the original paper defining the algorithm appears to
   be incorrect. Here are some test cases provided by Marc Horowitz and
   Simon Josefsson:

       64-fold("012345") =
       64-fold(303132333435) = be072631276b1955

       56-fold("password") =
       56-fold(70617373776f7264) = 78a07b6caf85fa

       64-fold("Rough Consensus, and Running Code") =
       64-fold(526f75676820436f6e73656e7375732c20616e642052756e
               6e696e6720436f6465) = bb6ed30870b7f0e0

       168-fold("password") =
       168-fold(70617373776f7264) =
               59e4a8ca7c0385c3c37b3f6d2000247cb6e6bd5b3e

       192-fold("MASSACHVSETTS INSTITVTE OF TECHNOLOGY")
       192-fold(4d41535341434856534554545320494e5354495456544520
               4f4620544543484e4f4c4f4759) =
               db3b0d8f0b061e603282b308a50841229ad798fab9540c1b

       168-fold("Q") =
       168-fold(51) =
               518a54a2 15a8452a 518a54a2 15a8452a
               518a54a2 15

       168-fold("ba") =
       168-fold(6261) =
               fb25d531 ae897449 9f52fd92 ea9857c4
               ba24cf29 7e

   Here are some additional values corresponding to folded values of the
   string "kerberos"; the 64-bit form is used in the des3 string-to-key
   (section 6.3.1).

```
    64-fold("kerberos") =
            6b657262 65726f73
    128-fold("kerberos") =
            6b657262 65726f73 7b9b5b2b 93132b93
    168-fold("kerberos") =
            8372c236 344e5f15 50cd0747 e15d62ca
            7a5a3bce a4
    256-fold("kerberos") =
            6b657262 65726f73 7b9b5b2b 93132b93
            5c9bdcda d95c9899 c4cae4de e6d6cae4
```

   Note that the initial octets exactly match the input string when the
   output length is a multiple of the input length.

A.2. mit_des_string_to_key

   The function mit_des_string_to_key is defined in section 6.2. We
   present here several test values, with some of the intermediate
   results. The fourth test demonstrates the use of UTF-8 with three
   characters. The last two tests are specifically constructed so as to
   trigger the weak-key fixups for the intermediate key produced by
   fan-folding; we have no test cases that cause such fixups for the
   final key.

```
UTF-8 encodings used in test vector:
eszett    U+00DF    C3 9F    s-caron    U+0161    C5 A1
c-acute   U+0107    C4 87    g-clef     U+1011E   F0 9D 84 9E
```

Test vector:

```
salt:         "ATHENA.MIT.EDUraeburn"
                          415448454e412e4d49542e4544557261656275726e
password:     "password"  70617373776f7264
fan-fold result:         c01e38688ac86c2e
intermediate key:        c11f38688ac86d2f
DES key:                 cbc22fae235298e3

salt:         "WHITEHOUSE.GOVdanny"
                          5748495445484f5553452e474f5664616e6e79
password:     "potatoe"   706f7461746f65
fan-fold result:         a028944ee63c0416
intermediate key:        a129944fe63d0416
DES key:                 df3d32a74fd92a01

salt:       "EXAMPLE.COMpianist"  4558414D504C452E434F4D7069616E697374
password:  g-clef (U+1011E)       f09d849e
fan-fold result:                 3c4a262c18fab090
intermediate key:                3d4a262c19fbb091
```

```
DES key:                             4ffb26bab0cd9413


salt: "ATHENA.MIT.EDUJuri" + s-caron(U+0161) + "i" + c-acute(U+0107)
                           415448454e412e4d49542e4544554a757269c5a169c487
password:       eszett(U+00DF)
                                     c39f
fan-fold result:b8f6c40e305afc9e
intermediate key:                    b9f7c40e315bfd9e
DES key:                             62c81a5232b5e69d


salt:         "AAAAAAAA"   4141414141414141
password:     "11119999"   3131313139393939
fan-fold result:          e0e0e0e0f0f0f0f0
intermediate key:         e0e0e0e0f1f1f101
DES key:                  984054d0f1a73e31


salt:         "FFFFAAAA"   4646464641414141
password:     "NNNN6666"   4e4e4e4e36363636
fan-fold result:          1e1e1e1e0e0e0e0e
intermediate key:         1f1f1f1f0e0e0efe
DES key:                  c4bf6b25adf7a4f8
```

   This trace provided by Simon Josefsson shows the intermediate
   processing stages of one of the test inputs:

```
    string_to_key (des-cbc-md5, string, salt)
          ;; string:
          ;; 'password' (length 8 bytes)
          ;; 70 61 73 73 77 6f 72 64
          ;; salt:
          ;; 'ATHENA.MIT.EDUraeburn' (length 21 bytes)
          ;; 41 54 48 45 4e 41 2e 4d  49 54 2e 45 44 55 72 61
          ;; 65 62 75 72 6e
    des_string_to_key (string, salt)
          ;; String:
          ;; 'password' (length 8 bytes)
          ;; 70 61 73 73 77 6f 72 64
          ;; Salt:
          ;; 'ATHENA.MIT.EDUraeburn' (length 21 bytes)
          ;; 41 54 48 45 4e 41 2e 4d  49 54 2e 45 44 55 72 61
          ;; 65 62 75 72 6e
    odd = 1;
    s = string | salt;
    tempstring = 0; /* 56-bit string */
    pad(s); /* with nulls to 8 byte boundary */
          ;; s = pad(string|salt):
          ;; 'passwordATHENA.MIT.EDUraeburn\x00\x00\x00'
          ;; (length 32 bytes)
```

```
            ;; 70 61 73 73 77 6f 72 64  41 54 48 45 4e 41 2e 4d
            ;; 49 54 2e 45 44 55 72 61  65 62 75 72 6e 00 00 00
    for (8byteblock in s) {
            ;; loop iteration 0
            ;; 8byteblock:
            ;; 'password' (length 8 bytes)
            ;; 70 61 73 73 77 6f 72 64
            ;; 01110000 01100001 01110011  01110011 01110111 01101111
            ;; 01110010 01100100
    56bitstring = removeMSBits(8byteblock);
            ;; 56bitstring:
            ;; 1110000 1100001 1110011  1110011 1110111 1101111
            ;; 1110010 1100100
    if (odd == 0) reverse(56bitstring);     ;; odd=1
    odd = ! odd
    tempstring = tempstring XOR 56bitstring;
            ;; tempstring
            ;; 1110000 1100001 1110011  1110011 1110111 1101111
            ;; 1110010 1100100

    for (8byteblock in s) {
            ;; loop iteration 1
            ;; 8byteblock:
            ;; 'ATHENA.M' (length 8 bytes)
            ;; 41 54 48 45 4e 41 2e 4d
            ;; 01000001 01010100 01001000  01000101 01001110 01000001
            ;; 00101110 01001101
    56bitstring = removeMSBits(8byteblock);
            ;; 56bitstring:
            ;; 1000001 1010100 1001000  1000101 1001110 1000001
            ;; 0101110 1001101
    if (odd == 0) reverse(56bitstring);     ;; odd=0
    reverse(56bitstring)
            ;; 56bitstring after reverse
            ;; 1011001 0111010 1000001  0111001 1010001 0001001
            ;; 0010101 1000001
    odd = ! odd
    tempstring = tempstring XOR 56bitstring;
            ;; tempstring
            ;; 0101001 1011011 0110010  1001010 0100110 1100110
            ;; 1100111 0100101

    for (8byteblock in s) {
            ;; loop iteration 2
            ;; 8byteblock:
            ;; 'IT.EDUra' (length 8 bytes)
            ;; 49 54 2e 45 44 55 72 61
            ;; 01001001 01010100 00101110  01000101 01000100 01010101
```

```
                    ;; 01110010 01100001
        56bitstring = removeMSBits(8byteblock);
                    ;; 56bitstring:
                    ;; 1001001 1010100 0101110  1000101 1000100 1010101
                    ;; 1110010 1100001
        if (odd == 0) reverse(56bitstring);     ;; odd=1
        odd = ! odd
        tempstring = tempstring XOR 56bitstring;
                    ;; tempstring
                    ;; 1100000 0001111 0011100  0001111 1100010 0110011
                    ;; 0010101 1000100

        for (8byteblock in s) {
                    ;; loop iteration 3
                    ;; 8byteblock:
                    ;; 'eburn\x00\x00\x00' (length 8 bytes)
                    ;; 65 62 75 72 6e 00 00 00
                    ;; 01100101 01100010 01110101  01110010 01101110 00000000
                    ;; 00000000 00000000
        56bitstring = removeMSBits(8byteblock);
                    ;; 56bitstring:
                    ;; 1100101 1100010 1110101  1110010 1101110 0000000
                    ;; 0000000 0000000
        if (odd == 0) reverse(56bitstring);     ;; odd=0
        reverse(56bitstring)
                    ;; 56bitstring after reverse
                    ;; 0000000 0000000 0000000  0111011 0100111 1010111
                    ;; 0100011 1010011
        odd = ! odd
        tempstring = tempstring XOR 56bitstring;
                    ;; tempstring
                    ;; 1100000 0001111 0011100  0110100 1000101 1100100
                    ;; 0110110 0010111

        for (8byteblock in s) {
        }
                    ;; for loop terminated

        tempkey = key_correction(add_parity_bits(tempstring));
                    ;; tempkey
                    ;; '\xc1\x1f8h\x8a\xc8m\x2f' (length 8 bytes)
                    ;; c1 1f 38 68 8a c8 6d 2f
                    ;; 11000001 00011111 00111000  01101000 10001010 11001000
                    ;; 01101101 00101111

        key = key_correction(DES-CBC-check(s,tempkey));
                    ;; key
                    ;; '\xcb\xc2\x2f\xae\x23R\x98\xe3' (length 8 bytes)
```

```
                    ;; cb c2 2f ae 23 52 98 e3
                    ;; 11001011 11000010 00101111  10101110 00100011 01010010
                    ;; 10011000 11100011

                    ;; string_to_key key:
                    ;; '\xcb\xc2\x2f\xae\x23R\x98\xe3' (length 8 bytes)
                    ;; cb c2 2f ae 23 52 98 e3
```

A.3. DES3 DR and DK

   These tests show the derived-random and derived-key values for the
   des3-hmac-sha1-kd encryption scheme, using the DR and DK functions
   defined in section 6.3.1. The input keys were randomly generated;
   the usage values are from this specification.

```
   key:                dce06b1f64c857a11c3db57c51899b2cc1791008ce973b92
   usage:              0000000155
   DR:                 935079d14490a75c3093c4a6e8c3b049c71e6ee705
   DK:                 925179d04591a79b5d3192c4a7e9c289b049c71f6ee604cd

   key:                5e13d31c70ef765746578531cb51c15bf11ca82c97cee9f2
   usage:              00000001aa
   DR:                 9f58e5a047d894101c469845d67ae3c5249ed812f2
   DK:                 9e58e5a146d9942a101c469845d67a20e3c4259ed913f207

   key:                98e6fd8a04a4b6859b75a176540b9752bad3ecd610a252bc
   usage:              0000000155
   DR:                 12fff90c773f956d13fc2ca0d0840349dbd39908eb
   DK:                 13fef80d763e94ec6d13fd2ca1d085070249dad39808eabf

   key:                622aec25a2fe2cad7094680b7c64940280084c1a7cec92b5
   usage:              00000001aa
   DR:                 f8debf05b097e7dc0603686aca35d91fd9a5516a70
   DK:                 f8dfbf04b097e6d9dc0702686bcb3489d91fd9a4516b703e

   key:                d3f8298ccb166438dcb9b93ee5a7629286a491f838f802fb
   usage:              6b65726265726f73 ("kerberos")
   DR:                 2270db565d2a3d64cfbfdc5305d4f778a6de42d9da
   DK:                 2370da575d2a3da864cebfdc5204d56df779a7df43d9da43

   key:                c1081649ada74362e6a1459d01dfd30d67c2234c940704da
   usage:              0000000155
   DR:                 348056ec98fcc517171d2b4d7a9493af482d999175
   DK:                 348057ec98fdc48016161c2a4c7a943e92ae492c989175f7

   key:                5d154af238f46713155719d55e2f1f790dd661f279a7917c
   usage:              00000001aa
   DR:                 a8818bc367dadacbe9a6c84627fb60c294b01215e5
```

```
DK:                  a8808ac267dada3dcbe9a7c84626fbc761c294b01315e5c1

key:                 798562e049852f57dc8c343ba17f2ca1d97394efc8adc443
usage:               0000000155
DR:                  c813f88b3be2b2f75424ce9175fbc8483b88c8713a
DK:                  c813f88a3be3b334f75425ce9175fbe3c8493b89c8703b49

key:                 26dce334b545292f2feab9a8701a89a4b99eb9942cecd016
usage:               00000001aa
DR:                  f58efc6f83f93e55e695fd252cf8fe59f7d5ba37ec
DK:                  f48ffd6e83f83e7354e694fd252cf83bfe58f7d5ba37ec5d
```

A.4. DES3string_to_key

   These are the keys generated for some of the above input strings for
   triple-DES with key derivation as defined in section 6.3.1.

   salt: "ATHENA.MIT.EDUraeburn"
   passwd: "password"
   key: 850bb51358548cd05e86768c313e3bfef7511937dcf72c3e

   salt: "WHITEHOUSE.GOVdanny"
   passwd: "potatoe"
   key: dfcd233dd0a43204ea6dc437fb15e061b02979c1f74f377a

   salt: "EXAMPLE.COMbuckaroo"
   passwd: "penny"
   key: 6d2fcdf2d6fbbc3ddcadb5da5710a23489b0d3b69d5d9d4a

   salt: "ATHENA.MIT.EDUJuri" + s-caron(U+0161) + "i"
             + c-acute(U+0107)
   passwd: eszett(U+00DF)
   key: 16d5a40e1ce3bacb61b9dce00470324c831973a7b952feb0

   salt: "EXAMPLE.COMpianist"
   passwd: g-clef(U+1011E)
   key: 85763726585dbc1cce6ec43e1f751f07f1c4cbb098f40b19

A.5. Modified CRC-32

   Below are modified-CRC32 values for various ASCII and octet strings.
   Only the printable ASCII characters are checksummed, without a C-
   style trailing zero-valued octet. The 32-bit modified CRC and the
   sequence of output bytes as used in Kerberos are shown. (The octet
   values are separated here to emphasize that they are octet values and
   not 32-bit numbers, which will be the most convenient form for
   manipulation in some implementations. The bit and byte order used

internally for such a number is irrelevant; the octet sequence
generated is what is important.)

```
mod-crc-32("foo") =                                   33 bc 32 73
mod-crc-32("test0123456789") =                        d6 88 3e b8
mod-crc-32("MASSACHVSETTS INSTITVTE OF TECHNOLOGY") =  f7 80 41 e3
mod-crc-32(8000) =                                    4b 98 83 3b
mod-crc-32(0008) =                                    32 88 db 0e
mod-crc-32(0080) =                                    20 83 b8 ed
mod-crc-32(80) =                                      20 83 b8 ed
mod-crc-32(80000000) =                                3b b6 59 ed
mod-crc-32(00000001) =                                96 30 07 77
```

B. Significant Changes from RFC 1510

   The encryption and checksum mechanism profiles are new. The old
   specification defined a few operations for various mechanisms but
   didn't outline what abstract properties should be required of new
   mechanisms, or how to ensure that a mechanism specification is
   complete enough for interoperability between implementations. The
   new profiles differ from the old specification in a few ways:

      Some message definitions in [Kerb1510] could be read as permitting
      the initial vector to be specified by the application; the text
      was too vague. It is explicitly not permitted in this
      specification. Some encryption algorithms may not use
      initialization vectors, so relying on chosen, secret
      initialization vectors for security is unwise. Also, the
      prepended confounder in the existing algorithms is roughly
      equivalent to a per-message initialization vector that is revealed
      in encrypted form. However, carrying state across from one
      encryption to another is explicitly permitted through the opaque
      "cipher state" object.

      The use of key derivation is new.

      Several new methods are introduced, including generation of a key
      in wire-protocol format from random input data.

      The means for influencing the string-to-key algorithm are laid out
      more clearly.

   Triple-DES support is new.

   The pseudo-random function is new.

   The des-cbc-crc, DES string-to-key and CRC descriptions have been
   updated to align them with existing implementations.

[Kerb1510] did not indicate what character set or encoding might be
used for pass phrases and salts.

In [Kerb1510], key types, encryption algorithms, and checksum
algorithms were only loosely associated, and the association was not
well described. In this specification, key types and encryption
algorithms have a one-to-one correspondence, and associations between
encryption and checksum algorithms are described so that checksums
can be computed given negotiated keys, without requiring further
negotiation for checksum types.

Notes

   [1] Although Message Authentication Code (MAC) or Message Integrity
       Check (MIC) would be more appropriate terms for many of the uses
       in this document, we continue to use the term checksum for
       historical reasons.

   [2] Extending CBC mode across messages would be one obvious example
       of this chaining. Another might be the use of counter mode, with
       a counter randomly initialized and attached to the ciphertext; a
       second message could continue incrementing the counter when
       chaining the cipher state, thus avoiding having to transmit
       another counter value. However, this chaining is only useful for
       uninterrupted, ordered sequences of messages.

   [3] In the case of Kerberos, the encrypted objects will generally be
       ASN.1 DER encodings, which contain indications of their length in
       the first few octets.

   [4] As of the time of this writing, new modes of operation have been
       proposed, some of which may permit encryption and integrity
       protection simultaneously. After some of these proposals have
       been subjected to adequate analysis, we may wish to formulate a
       new simplified profile based on one of them.

   [5] It should be noted that the sample vector in appendix B.2 of the
       original paper appears to be incorrect. Two independent
       implementations from the specification (one in C by Marc
       Horowitz, and another in Scheme by Bill Sommerfeld) agree on a
       value different from that in [Blumenthal96].

   [6] For example, in MIT's implementation of [Kerb1510], the rsa-md5
       unkeyed checksum of application data may be included in an
       authenticator encrypted in a service's key.

   [7] Using a variant of the key limits the use of a key to a
       particular function, separating the functions of generating a

checksum from other encryption performed using the session key.
The constant 0xF0F0F0F0F0F0F0F0 was chosen because it maintains
key parity. The properties of DES precluded the use of the
complement. The same constant is used for similar purpose in the
Message Integrity Check in the Privacy Enhanced Mail standard.

[8] Perhaps one of the more common reasons for directly performing
encryption is direct control over the negotiation and to select a
"sufficiently strong" encryption algorithm (whatever that means
in the context of a given application). Although Kerberos
directly provides no direct facility for negotiating encryption
types between the application client and server, there are other
means to accomplish similar goals (for example, requesting only
"strong" session key types from the KDC, and assuming that the
type actually returned by the KDC will be understood and
supported by the application server).

Normative References

[BCP26]   Narten, T. and H. Alvestrand, "Guidelines for Writing
                 an IANA Considerations Section in RFCs", BCP 26, RFC
                 2434, October 1998.

[Bellare98]   Bellare, M., Desai, A., Pointcheval, D., and P.
                 Rogaway, "Relations Among Notions of Security for
                 Public-Key Encryption Schemes". Extended abstract
                 published in Advances in Cryptology-Crypto 98
                 Proceedings, Lecture Notes in Computer Science Vol.
                 1462, H. Krawcyzk ed., Springer-Verlag, 1998.

[Blumenthal96]   Blumenthal, U. and S. Bellovin, "A Better Key Schedule
                 for DES-Like Ciphers", Proceedings of PRAGOCRYPT '96,
                 1996.

[CRC]   International Organization for Standardization, "ISO
                 Information Processing Systems - Data Communication -
                 High-Level Data Link Control Procedure - Frame
                 Structure," IS 3309, 3rd Edition, October 1984.

[DES77]   National Bureau of Standards, U.S. Department of
                 Commerce, "Data Encryption Standard," Federal
                 Information Processing Standards Publication 46,
                 Washington, DC, 1977.

    [DESI81]   National Bureau of Standards, U.S. Department of
                    Commerce, "Guidelines for implementing and using NBS
                    Data Encryption Standard," Federal Information
                    Processing Standards Publication 74, Washington, DC,
                    1981.

    [DESM80]   National Bureau of Standards, U.S. Department of
                    Commerce, "DES Modes of Operation," Federal
                    Information Processing Standards Publication 81,
                    Springfield, VA, December 1980.

    [Dolev91]   Dolev, D., Dwork, C., and M. Naor, "Non-malleable
                    cryptography", Proceedings of the 23rd Annual
                    Symposium on Theory of Computing, ACM, 1991.

    [HMAC]   Krawczyk, H., Bellare, M., and R. Canetti, "HMAC:
                    Keyed-Hashing for Message Authentication", RFC 2104,
                    February 1997.

    [KRB5-AES]   Raeburn, K., "Advanced Encryption Standard (AES)
                    Encryption for Kerberos 5", RFC 3962, February 2005.

    [MD4-92]   Rivest, R., "The MD4 Message-Digest Algorithm", RFC
                    1320, April 1992.

    [MD5-92]   Rivest, R., "The MD5 Message-Digest Algorithm ", RFC
                    1321, April 1992.

    [SG92]   Stubblebine, S. and V. D. Gligor, "On Message
                    Integrity in Cryptographic Protocols," in Proceedings
                    of the IEEE Symposium on Research in Security and
                    Privacy, Oakland, California, May 1992.

Informative References

    [Bellovin91]   Bellovin, S. M. and M. Merrit, "Limitations of the
                    Kerberos Authentication System", in Proceedings of the
                    Winter 1991 Usenix Security Conference, January, 1991.

    [Bellovin99]   Bellovin, S. M. and D. Atkins, private communications,
                    1999.

    [EFF-DES]   Electronic Frontier Foundation, "Cracking DES: Secrets
                    of Encryption Research, Wiretap Politics, and Chip
                    Design", O'Reilly & Associates, Inc., May 1998.

    [ESP-DES]   Madson, C. and N. Doraswamy, "The ESP DES-CBC Cipher
                    Algorithm With Explicit IV", RFC 2405, November 1998.

   [GSS-KRB5]   Linn, J., "The Kerberos Version 5 GSS-API Mechanism",
                    RFC 1964, June 1996.

   [HMAC-TEST]   Cheng, P. and R. Glenn, "Test Cases for HMAC-MD5 and
                    HMAC-SHA-1", RFC 2202, September 1997.

   [IPSEC-HMAC]   Madson, C. and R. Glenn, "The Use of HMAC-SHA-1-96
                    within ESP and AH", RFC 2404, November 1998.

   [Kerb]   Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The
                    Kerberos Network Authentication Service (V5)", Work in
                    Progress, September 2004.

   [Kerb1510]   Kohl, J. and C. Neuman, "The Kerberos Network
                    Authentication Service (V5)", RFC 1510, September
                    1993.

   [RC5]   Baldwin, R. and R. Rivest, "The RC5, RC5-CBC, RC5-
                    CBC-Pad, and RC5-CTS Algorithms", RFC 2040, October
                    1996.

   [RFC1851]   Karn, P., Metzger, P., and W. Simpson, "The ESP Triple
                    DES Transform", RFC 1851, September 1995.

   [Schneier96]   Schneier, B., "Applied Cryptography Second Edition",
                    John Wiley & Sons, New York, NY, 1996. ISBN 0-471-
                    12845-7.

Editor's Address

   Kenneth Raeburn
   Massachusetts Institute of Technology
   77 Massachusetts Avenue
   Cambridge, MA 02139

   EMail: raeburn@mit.edu