Network Working Group
Request for Comments: 3962
Category: Standards Track

K. Raeburn
MIT

February 2005

Advanced Encryption Standard (AES) Encryption for Kerberos 5

Status of This Memo

Copyright Notice

Abstract

The United States National Institute of Standards and Technology (NIST) has chosen a new Advanced Encryption Standard (AES), which is significantly faster and (it is believed) more secure than the old Data Encryption Standard (DES) algorithm. This document is a specification for the addition of this algorithm to the Kerberos cryptosystem suite.

1. Introduction

This document defines encryption key and checksum types for Kerberos 5 using the AES algorithm recently chosen by NIST. These new types support 128-bit block encryption and key sizes of 128 or 256 bits.

Using the "simplified profile" of [KCRYPTO], we can define a pair of encryption and checksum schemes. AES is used with ciphertext stealing to avoid message expansion, and SHA-1 [SHA1] is the associated checksum function.

2. Conventions used in this Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14, RFC 2119 [KEYWORDS].

3. Protocol Key Representation

The profile in [KCRYPTO] treats keys and random octet strings as conceptually different. But since the AES key space is dense, we can use any bit string of appropriate length as a key. We use the byte representation for the key described in [AES], where the first bit of the bit string is the high bit of the first byte of the byte string (octet string) representation.

4. Key Generation from Pass Phrases or Random Data

Given the above format for keys, we can generate keys from the appropriate amounts of random data (128 or 256 bits) by simply copying the input string.

To generate an encryption key from a pass phrase and salt string, we use the PBKDF2 function from PKCS #5 v2.0 ([PKCS5]), with parameters indicated below, to generate an intermediate key (of the same length as the desired final key), which is then passed into the DK function with the 8-octet ASCII string "kerberos" as is done for des3-cbc-hmac-sha1-kd in [KCRYPTO]. (In [KCRYPTO] terms, the PBKDF2 function produces a "random octet string", hence the application of the random-to-key function even though it's effectively a simple identity operation.) The resulting key is the user's long-term key for use with the encryption algorithm in question.

```
tkey = random2key(PBKDF2(passphrase, salt, iter_count, keylength))
key = DK(tkey, "kerberos")
```

The pseudorandom function used by PBKDF2 will be a SHA-1 HMAC of the passphrase and salt, as described in Appendix B.1 to PKCS#5.

The number of iterations is specified by the string-to-key parameters supplied. The parameter string is four octets indicating an unsigned number in big-endian order. This is the number of iterations to be performed. If the value is 00 00 00 00, the number of iterations to be performed is 4,294,967,296 (2**32). (Thus the minimum expressible iteration count is 1.)

For environments where slower hardware is the norm, implementations of protocols such as Kerberos may wish to limit the number of iterations to prevent a spoofed response supplied by an attacker from consuming lots of client-side CPU time; if such a limit is implemented, it SHOULD be no less than 50,000. Even for environments with fast hardware, 4 billion iterations is likely to take a fairly long time; much larger bounds might still be enforced, and it might be wise for implementations to permit interruption of this operation by the user if the environment allows for it.

If the string-to-key parameters are not supplied, the value used is 00 00 10 00 (decimal 4,096, indicating 4,096 iterations).

Note that this is not a requirement, nor even a recommendation, for this value to be used in "optimistic preauthentication" (e.g., attempting timestamp-based preauthentication using the user's long-term key without having first communicated with the KDC) in the absence of additional information, or as a default value for sites to use for their principals' long-term keys in their Kerberos database. It is simply the interpretation of the absence of the string-to-key parameter field when the KDC has had an opportunity to provide it.

Sample test vectors are given in Appendix B.

5. Ciphertext Stealing

Cipher block chaining is used to encrypt messages, with the initial vector stored in the cipher state. Unlike previous Kerberos cryptosystems, we use ciphertext stealing to handle the possibly partial final block of the message.

Ciphertext stealing is described on pages 195-196 of [AC], and section 8 of [RC5]; it has the advantage that no message expansion is done during encryption of messages of arbitrary sizes as is typically done in CBC mode with padding. Some errata for [RC5] are listed in Appendix A and are considered part of the ciphertext stealing technique as used here.

Ciphertext stealing, as defined in [RC5], assumes that more than one block of plain text is available. If exactly one block is to be encrypted, that block is simply encrypted with AES (also known as ECB mode). Input smaller than one block is padded at the end to one block; the values of the padding bits are unspecified. (Implementations MAY use all-zero padding, but protocols MUST NOT rely on the result being deterministic. Implementations MAY use random padding, but protocols MUST NOT rely on the result not being deterministic. Note that in most cases, the Kerberos encryption profile will add a random confounder independent of this padding.)

For consistency, ciphertext stealing is always used for the last two blocks of the data to be encrypted, as in [RC5]. If the data length is a multiple of the block size, this is equivalent to plain CBC mode with the last two ciphertext blocks swapped.

A test vector is given in Appendix B.

The initial vector carried out from one encryption for use in a subsequent encryption is the next-to-last block of the encryption output; this is the encrypted form of the last plaintext block. When decrypting, the next-to-last block of the supplied ciphertext is carried forward as the next initial vector. If only one ciphertext block is available (decrypting one block, or encrypting one block or less), then that one block is carried out instead.

6. Kerberos Algorithm Profile Parameters

This is a summary of the parameters to be used with the simplified algorithm profile described in [KCRYPTO]:

protocol key format	128- or 256-bit string
string-to-key function	PBKDF2+DK with variable iteration count (see above)
default string-to-key parameters	00 00 10 00
key-generation seed length	key size
random-to-key function	identity function
hash function, H	SHA-1
HMAC output size, h	12 octets (96 bits)
message block size, m	1 octet
encryption/decryption functions, E and D	AES in CBC-CTS mode (cipher block size 16 octets), with next-to-last block (last block if only one) as CBC-style ivec

Using this profile with each key size gives us two each of encryption and checksum algorithm definitions.

7. Assigned Numbers

The following encryption type numbers are assigned:

encryption types		
type name	etype value	key size
aes128-cts-hmac-sha1-96	17	128
aes256-cts-hmac-sha1-96	18	256

The following checksum type numbers are assigned:

checksum types		
type name	sumtype value	length
hmac-sha1-96-aes128	15	96
hmac-sha1-96-aes256	16	96

These checksum types will be used with the corresponding encryption types defined above.

8. Security Considerations

This new algorithm has not been around long enough to receive the decades of intense analysis that DES has received. It is possible that some weakness exists that has not been found by the cryptographers analyzing these algorithms before and during the AES selection process.

The use of the HMAC function has drawbacks for certain pass phrase lengths. For example, a pass phrase longer than the hash function block size (64 bytes, for SHA-1) is hashed to a smaller size (20 bytes) before applying the main HMAC algorithm. However, entropy is generally sparse in pass phrases, especially in long ones, so this may not be a problem in the rare cases of users with long pass phrases.

Also, generating a 256-bit key from a pass phrase of any length may be deceptive, as the effective entropy in pass-phrase-derived key cannot be nearly that large given the properties of the string-to-key function described here.

The iteration count in PBKDF2 appears to be useful primarily as a constant multiplier for the amount of work required for an attacker using brute-force methods. Unfortunately, it also multiplies, by the same amount, the work needed by a legitimate user with a valid password. Thus the work factor imposed on an attacker (who may have many powerful workstations at his disposal) must be balanced against the work factor imposed on the legitimate user (who may have a PDA or cell phone); the available computing power on either side increases as time goes on, as well. A better way to deal with the brute-force attack is through preauthentication mechanisms that provide better protection of the user's long-term key. Use of such mechanisms is out of the scope of this document.

If a site does wish to use this means of protection against a brute-force attack, the iteration count should be chosen based on the facilities available to both attacker and legitimate user, and the amount of work the attacker should be required to perform to acquire the key or password.

As an example:

> The author's tests on a 2GHz Pentium 4 system indicated that in one second, nearly 90,000 iterations could be done, producing a 256-bit key. This was using the SHA-1 assembly implementation from OpenSSL, and a pre-release version of the PBKDF2 code for MIT's Kerberos package, on a single system. No attempt was made to do multiple hashes in parallel, so we assume an attacker doing so can probably do at least 100,000 iterations per second -- rounded up to 2**17, for ease of calculation. For simplicity, we also assume the final AES encryption step costs nothing.
>
> Paul Leach estimates [LEACH] that a password-cracking dictionary may have on the order of 2**21 entries, with capitalization, punctuation, and other variations contributing perhaps a factor of 2**11, giving a ballpark estimate of 2**32.
>
> Thus, for a known iteration count N and a known salt string, an attacker with some number of computers comparable to the author's would need roughly N*2**15 CPU seconds to convert the entire dictionary plus variations into keys.
>
> An attacker using a dozen such computers for a month would have roughly 2**25 CPU seconds available. So using 2**12 (4,096) iterations would mean an attacker with a dozen such computers dedicated to a brute-force attack against a single key (actually, any password-derived keys sharing the same salt and iteration

count) would process all the variations of the dictionary entries in four months and, on average, would likely find the user's password in two months.

Thus, if this form of attack is of concern, users should be required to change their passwords every few months, and an iteration count a few orders of magnitude higher should be chosen. Perhaps several orders of magnitude, as many users will tend to use the shorter and simpler passwords (to the extent they can, given a site's password quality checks) that the attacker would likely try first.

Since this estimate is based on currently available CPU power, the iteration counts used for this mode of defense should be increased over time, at perhaps 40%-60% each year or so.

Note that if the attacker has a large amount of storage available, intermediate results could be cached, saving a lot of work for the next attack with the same salt and a greater number of iterations than had been run at the point where the intermediate results were saved. Thus, it would be wise to generate a new random salt string when passwords are changed. The default salt string, derived from the principal name, only protects against the use of one dictionary of keys against multiple users.

If the PBKDF2 iteration count can be spoofed by an intruder on the network, and the limit on the accepted iteration count is very high, the intruder may be able to introduce a form of denial of service attack against the client by sending a very high iteration count, causing the client to spend a great deal of CPU time computing an incorrect key.

An intruder spoofing the KDC reply, providing a low iteration count and reading the client's reply from the network, may be able to reduce the work needed in the brute-force attack outlined above. Thus, implementations may seek to enforce lower bounds on the number of iterations that will be used.

Since threat models and typical end-user equipment will vary widely from site to site, allowing site-specific configuration of such bounds is recommended.

Any benefit against other attacks specific to the HMAC or SHA-1 algorithms is probably achieved with a fairly small number of iterations.

In the "optimistic preauthentication" case mentioned in section 3, the client may attempt to produce a key without first communicating with the KDC. If the client has no additional information, it can only guess as to the iteration count to be used. Even such heuristics as "iteration count X was used to acquire tickets for the same principal only N hours ago" can be wrong. Given the recommendation above for increasing the iteration counts used over time, it is impossible to recommend any specific default value for this case; allowing site-local configuration is recommended. (If the lower and upper bound checks described above are implemented, the default count for optimistic preauthentication should be between those bounds.)

Ciphertext stealing mode, as it requires no additional padding in most cases, will reveal the exact length of each message being encrypted rather than merely bounding it to a small range of possible lengths as in CBC mode. Such obfuscation should not be relied upon at higher levels in any case; if the length must be obscured from an outside observer, this should be done by intentionally varying the length of the message to be encrypted.

9. IANA Considerations

Kerberos encryption and checksum type values used in section 7 were previously reserved in [KCRYPTO] for the mechanisms defined in this document. The registries have been updated to list this document as the reference.

10. Acknowledgements

Thanks to John Brezak, Gerardo Diaz Cuellar, Ken Hornstein, Paul Leach, Marcus Watts, Larry Zhu, and others for feedback on earlier versions of this document.

A. Errata for RFC 2040 Section 8

(Copied from the RFC Editor's errata web site on July 8, 2004.)

Reported By: Bob Baldwin; baldwin@plusfive.com
Date: Fri, 26 Mar 2004 06:49:02 -0800

In Section 8, Description of RC5-CTS, of the encryption method, it says:

> 1. Exclusive-or Pn-1 with the previous ciphertext block, Cn-2, to create Xn-1.

It should say:

> 1. Exclusive-or Pn-1 with the previous ciphertext block, Cn-2, to create Xn-1. For short messages where Cn-2 does not exist, use IV.

Reported By: Bob Baldwin; baldwin@plusfive.com
Date: Mon, 22 Mar 2004 20:26:40 -0800

In Section 8, first paragraph, second sentence says:

> This mode handles any length of plaintext and produces ciphertext whose length matches the plaintext length.

In Section 8, first paragraph, second sentence should read:

> This mode handles any length of plaintext longer than one block and produces ciphertext whose length matches the plaintext length.

In Section 8, step 6 of the decryption method says:

> 6. Decrypt En to create Pn-1.

In Section 8, step 6 of the decryption method should read:

> 6. Decrypt En and exclusive-or with Cn-2 to create Pn-1. For short messages where Cn-2 does not exist, use the IV.

B. Sample Test Vectors

Sample values for the PBKDF2 HMAC-SHA1 string-to-key function are included below.

```
Iteration count = 1
Pass phrase = "password"
Salt = "ATHENA.MIT.EDUraeburn"
128-bit PBKDF2 output:
    cd ed b5 28 1b b2 f8 01 56 5a 11 22 b2 56 35 15
128-bit AES key:
    42 26 3c 6e 89 f4 fc 28 b8 df 68 ee 09 79 9f 15
256-bit PBKDF2 output:
    cd ed b5 28 1b b2 f8 01 56 5a 11 22 b2 56 35 15
    0a d1 f7 a0 4b b9 f3 a3 33 ec c0 e2 e1 f7 08 37
256-bit AES key:
    fe 69 7b 52 bc 0d 3c e1 44 32 ba 03 6a 92 e6 5b
    bb 52 28 09 90 a2 fa 27 88 39 98 d7 2a f3 01 61

Iteration count = 2
Pass phrase = "password"
Salt="ATHENA.MIT.EDUraeburn"
128-bit PBKDF2 output:
    01 db ee 7f 4a 9e 24 3e 98 8b 62 c7 3c da 93 5d
128-bit AES key:
    c6 51 bf 29 e2 30 0a c2 7f a4 69 d6 93 bd da 13
256-bit PBKDF2 output:
    01 db ee 7f 4a 9e 24 3e 98 8b 62 c7 3c da 93 5d
    a0 53 78 b9 32 44 ec 8f 48 a9 9e 61 ad 79 9d 86
256-bit AES key:
    a2 e1 6d 16 b3 60 69 c1 35 d5 e9 d2 e2 5f 89 61
    02 68 56 18 b9 59 14 b4 67 c6 76 22 22 58 24 ff

Iteration count = 1200
Pass phrase = "password"
Salt = "ATHENA.MIT.EDUraeburn"
128-bit PBKDF2 output:
    5c 08 eb 61 fd f7 1e 4e 4e c3 cf 6b a1 f5 51 2b
128-bit AES key:
    4c 01 cd 46 d6 32 d0 1e 6d be 23 0a 01 ed 64 2a
256-bit PBKDF2 output:
    5c 08 eb 61 fd f7 1e 4e 4e c3 cf 6b a1 f5 51 2b
    a7 e5 2d db c5 e5 14 2f 70 8a 31 e2 e6 2b 1e 13
256-bit AES key:
    55 a6 ac 74 0a d1 7b 48 46 94 10 51 e1 e8 b0 a7
    54 8d 93 b0 ab 30 a8 bc 3f f1 62 80 38 2b 8c 2a
```

```
Iteration count = 5
Pass phrase = "password"
Salt=0x1234567878563412
128-bit PBKDF2 output:
    d1 da a7 86 15 f2 87 e6 a1 c8 b1 20 d7 06 2a 49
128-bit AES key:
    e9 b2 3d 52 27 37 47 dd 5c 35 cb 55 be 61 9d 8e
256-bit PBKDF2 output:
    d1 da a7 86 15 f2 87 e6 a1 c8 b1 20 d7 06 2a 49
    3f 98 d2 03 e6 be 49 a6 ad f4 fa 57 4b 6e 64 ee
256-bit AES key:
    97 a4 e7 86 be 20 d8 1a 38 2d 5e bc 96 d5 90 9c
    ab cd ad c8 7c a4 8f 57 45 04 15 9f 16 c3 6e 31
(This test is based on values given in [PECMS].)

Iteration count = 1200
Pass phrase = (64 characters)
  "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
Salt="pass phrase equals block size"
128-bit PBKDF2 output:
    13 9c 30 c0 96 6b c3 2b a5 5f db f2 12 53 0a c9
128-bit AES key:
    59 d1 bb 78 9a 82 8b 1a a5 4e f9 c2 88 3f 69 ed
256-bit PBKDF2 output:
    13 9c 30 c0 96 6b c3 2b a5 5f db f2 12 53 0a c9
    c5 ec 59 f1 a4 52 f5 cc 9a d9 40 fe a0 59 8e d1
256-bit AES key:
    89 ad ee 36 08 db 8b c7 1f 1b fb fe 45 94 86 b0
    56 18 b7 0c ba e2 20 92 53 4e 56 c5 53 ba 4b 34

Iteration count = 1200
Pass phrase = (65 characters)
  "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
Salt = "pass phrase exceeds block size"
128-bit PBKDF2 output:
    9c ca d6 d4 68 77 0c d5 1b 10 e6 a6 87 21 be 61
128-bit AES key:
    cb 80 05 dc 5f 90 17 9a 7f 02 10 4c 00 18 75 1d
256-bit PBKDF2 output:
    9c ca d6 d4 68 77 0c d5 1b 10 e6 a6 87 21 be 61
    1a 8b 4d 28 26 01 db 3b 36 be 92 46 91 5e c8 2a
256-bit AES key:
    d7 8c 5c 9c b8 72 a8 c9 da d4 69 7f 0b b5 b2 d2
    14 96 c8 2b eb 2c ae da 21 12 fc ee a0 57 40 1b
```

```
Iteration count = 50
Pass phrase = g-clef (0xf09d849e)
Salt = "EXAMPLE.COMpianist"
128-bit PBKDF2 output:
    6b 9c f2 6d 45 45 5a 43 a5 b8 bb 27 6a 40 3b 39
128-bit AES key:
    f1 49 c1 f2 e1 54 a7 34 52 d4 3e 7f e6 2a 56 e5
256-bit PBKDF2 output:
    6b 9c f2 6d 45 45 5a 43 a5 b8 bb 27 6a 40 3b 39
    e7 fe 37 a0 c4 1e 02 c2 81 ff 30 69 e1 e9 4f 52
256-bit AES key:
    4b 6d 98 39 f8 44 06 df 1f 09 cc 16 6d b4 b8 3c
    57 18 48 b7 84 a3 d6 bd c3 46 58 9a 3e 39 3f 9e
```

Some test vectors for CBC with ciphertext stealing, using an initial vector of all-zero.

```
AES 128-bit key:
  0000:  63 68 69 63 6b 65 6e 20 74 65 72 69 79 61 6b 69

IV:
  0000:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Input:
  0000:  49 20 77 6f 75 6c 64 20 6c 69 6b 65 20 74 68 65
  0010:  20
Output:
  0000:  c6 35 35 68 f2 bf 8c b4 d8 a5 80 36 2d a7 ff 7f
  0010:  97
Next IV:
  0000:  c6 35 35 68 f2 bf 8c b4 d8 a5 80 36 2d a7 ff 7f

IV:
  0000:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Input:
  0000:  49 20 77 6f 75 6c 64 20 6c 69 6b 65 20 74 68 65
  0010:  20 47 65 6e 65 72 61 6c 20 47 61 75 27 73 20
Output:
  0000:  fc 00 78 3e 0e fd b2 c1 d4 45 d4 c8 ef f7 ed 22
  0010:  97 68 72 68 d6 ec cc c0 c0 7b 25 e2 5e cf e5
Next IV:
  0000:  fc 00 78 3e 0e fd b2 c1 d4 45 d4 c8 ef f7 ed 22
```

```
   IV:
     0000:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   Input:
     0000:  49 20 77 6f 75 6c 64 20 6c 69 6b 65 20 74 68 65
     0010:  20 47 65 6e 65 72 61 6c 20 47 61 75 27 73 20 43
   Output:
     0000:  39 31 25 23 a7 86 62 d5 be 7f cb cc 98 eb f5 a8
     0010:  97 68 72 68 d6 ec cc c0 c0 7b 25 e2 5e cf e5 84
   Next IV:
     0000:  39 31 25 23 a7 86 62 d5 be 7f cb cc 98 eb f5 a8

   IV:
     0000:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   Input:
     0000:  49 20 77 6f 75 6c 64 20 6c 69 6b 65 20 74 68 65
     0010:  20 47 65 6e 65 72 61 6c 20 47 61 75 27 73 20 43
     0020:  68 69 63 6b 65 6e 2c 20 70 6c 65 61 73 65 2c
   Output:
     0000:  97 68 72 68 d6 ec cc c0 c0 7b 25 e2 5e cf e5 84
     0010:  b3 ff fd 94 0c 16 a1 8c 1b 55 49 d2 f8 38 02 9e
     0020:  39 31 25 23 a7 86 62 d5 be 7f cb cc 98 eb f5
   Next IV:
     0000:  b3 ff fd 94 0c 16 a1 8c 1b 55 49 d2 f8 38 02 9e

   IV:
     0000:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   Input:
     0000:  49 20 77 6f 75 6c 64 20 6c 69 6b 65 20 74 68 65
     0010:  20 47 65 6e 65 72 61 6c 20 47 61 75 27 73 20 43
     0020:  68 69 63 6b 65 6e 2c 20 70 6c 65 61 73 65 2c 20
   Output:
     0000:  97 68 72 68 d6 ec cc c0 c0 7b 25 e2 5e cf e5 84
     0010:  9d ad 8b bb 96 c4 cd c0 3b c1 03 e1 a1 94 bb d8
     0020:  39 31 25 23 a7 86 62 d5 be 7f cb cc 98 eb f5 a8
   Next IV:
     0000:  9d ad 8b bb 96 c4 cd c0 3b c1 03 e1 a1 94 bb d8
```

```
   IV:
     0000:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   Input:
     0000:  49 20 77 6f 75 6c 64 20 6c 69 6b 65 20 74 68 65
     0010:  20 47 65 6e 65 72 61 6c 20 47 61 75 27 73 20 43
     0020:  68 69 63 6b 65 6e 2c 20 70 6c 65 61 73 65 2c 20
     0030:  61 6e 64 20 77 6f 6e 74 6f 6e 20 73 6f 75 70 2e
   Output:
     0000:  97 68 72 68 d6 ec cc c0 c0 7b 25 e2 5e cf e5 84
     0010:  39 31 25 23 a7 86 62 d5 be 7f cb cc 98 eb f5 a8
     0020:  48 07 ef e8 36 ee 89 a5 26 73 0d bc 2f 7b c8 40
     0030:  9d ad 8b bb 96 c4 cd c0 3b c1 03 e1 a1 94 bb d8
   Next IV:
     0000:  48 07 ef e8 36 ee 89 a5 26 73 0d bc 2f 7b c8 40
```

Normative References

[AC] Schneier, B., "Applied Cryptography", second edition, John Wiley and Sons, New York, 1996.

[AES] National Institute of Standards and Technology, U.S. Department of Commerce, "Advanced Encryption Standard", Federal Information Processing Standards Publication 197, Washington, DC, November 2001.

[KCRYPTO] Raeburn, K., "Encryption and Checksum Specifications for Kerberos 5", RFC 3961, February 2005.

[KEYWORDS] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[PKCS5] Kaliski, B., "PKCS #5: Password-Based Cryptography Specification Version 2.0", RFC 2898, September 2000.

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

[SHA1] National Institute of Standards and Technology, U.S. Department of Commerce, "Secure Hash Standard", Federal Information Processing Standards Publication 180-1, Washington, DC, April 1995.

Informative References

[LEACH] Leach, P., email to IETF Kerberos working group mailing list, 5 May 2003, ftp://ftp.ietf.org/ietf-mail-archive/krb-wg/2003-05.mail.

[PECMS] Gutmann, P., "Password-based Encryption for CMS", RFC 3211, December 2001.

Author's Address

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

EMail: raeburn@mit.edu

Full Copyright Statement

Intellectual Property

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.