Internet Engineering Task Force (IETF)
Request for Comments: 6476
Category: Standards Track
ISSN: 2070-1721

P. Gutmann
University of Auckland

January 2012

Using Message Authentication Code (MAC) Encryption in the Cryptographic Message Syntax (CMS)

Abstract

This document specifies the conventions for using Message Authentication Code (MAC) encryption with the Cryptographic Message Syntax (CMS) authenticated-enveloped-data content type. This mirrors the use of a MAC combined with an encryption algorithm that's already employed in IPsec, Secure Socket Layer / Transport Layer Security (SSL/TLS) and Secure SHell (SSH), which is widely supported in existing crypto libraries and hardware and has been extensively analysed by the crypto community.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc6476.

Table of Contents

1. Introduction

This document specifies the conventions for using MAC-authenticated encryption with the Cryptographic Message Syntax (CMS) authenticated-enveloped-data content type. This mirrors the use of a MAC combined with an encryption algorithm that's already employed in IPsec, SSL/TLS and SSH, which is widely supported in existing crypto libraries and hardware and has been extensively analysed by the crypto community.

1.1. Conventions Used in This Document

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

2. Background

Integrity-protected encryption is a standard feature of session-oriented security protocols like [IPsec], [SSH], and [TLS]. Until recently, however, integrity-protected encryption wasn't available for message-based security protocols like CMS, although [OpenPGP] added a form of integrity protection by encrypting a SHA-1 hash of the message alongside the message contents to provide authenticate-and-encrypt protection. Usability studies have shown that users expect encryption to provide integrity protection [Garfinkel], creating cognitive dissonance problems when the security mechanisms don't in fact provide this assurance.

This document applies the same encrypt-and-authenticate mechanism already employed in IPsec, SSH, and SSL/TLS to CMS (technically some of these actually use authenticate-and-encrypt rather than encrypt-and-authenticate, since what's authenticated is the plaintext and not the ciphertext). This mechanism is widely supported in existing crypto libraries and hardware and has been extensively analysed by the crypto community [EncryptThenAuth].

3. CMS Encrypt-and-Authenticate Overview

Conventional CMS encryption uses a content-encryption key (CEK) to encrypt a message payload, with the CEK typically being in turn encrypted by a key-encryption key (KEK). Authenticated encryption requires two keys: one for encryption and a second one for authentication. Like other mechanisms that use authenticated encryption, this document employs a pseudorandom function (PRF) to convert a single block of keying material into the two keys required for encryption and authentication. This converts the standard CMS encryption operation:

```
KEK( CEK ) || CEK( data )
```

into:

```
KEK( master_secret ) || MAC( CEK( data ) )
```

where the MAC key MAC-K and encryption key CEK-K are derived from the master_secret via:

```
MAC-K := PRF( master_secret, "authentication" );
CEK-K := PRF( master_secret, "encryption" );
```

3.1. Rationale

There are several possible means of deriving the two keys required for the encrypt-and-authenticate process from the single key normally provided by the key exchange or key transport mechanisms. Several of these, however, have security or practical issues. For example, any mechanism that uses the single exchanged key in its entirety for encryption (using, perhaps, PRF(key) as the MAC key) can be converted back to unauthenticated data by removing the outer MAC layer and rewriting the CMS envelope back to plain EnvelopedData or EncryptedData. By applying the PRF intermediate step, any attempt at a rollback attack will result in a decryption failure.

The option chosen here -- the use of a PRF to derive the necessary sets of keying material from a master secret -- is well-established through its use in IPsec, SSH, and SSL/TLS and is widely supported in both crypto libraries and in encryption hardware.

The PRF used is Password-Based Key Derivation Function 2 (PBKDF2) because its existing use in CMS makes it the most obvious candidate for such a function. In the future, if a universal PRF -- for example, [HKDF] -- is adopted, then this can be substituted for PBKDF2 by specifying it in the prfAlgorithm field covered in Section 4.

The resulting processing operations consist of a combination of the operations used for the existing CMS content types EncryptedData and AuthenticatedData, allowing them to be implemented relatively simply using existing code.

4. CMS Encrypt-and-Authenticate

The encrypt-and-authenticate mechanism is implemented within the existing CMS RecipientInfo framework by defining a new pseudo-algorithm type, authEnc, which is used in place of a monolithic encrypt and hash algorithm. The RecipientInfo is used as a key container for the master secret used by the pseudo-algorithm from which the encryption and authentication keys for existing single-purpose encrypt-only and MAC-only algorithms are derived. Thus, instead of using the RecipientInfo to communicate (for example) an AES or HMAC-SHA1 key, it communicates a master secret from which the required AES encryption and HMAC-SHA1 authentication keys are derived.

The authEnc pseudo-algorithm comes in two forms: one conveying 128 bits of keying material and one conveying 256 bits:

```
id-smime OBJECT IDENTIFIER ::= { iso(1) member-body(2)
            us(840) rsadsi(113549) pkcs(1) pkcs9(9) 16 }

id-alg  OBJECT IDENTIFIER ::= { id-smime 3 }

id-alg-authEnc-128 OBJECT IDENTIFIER ::= { id-alg 15 }
id-alg-authEnc-256 OBJECT IDENTIFIER ::= { id-alg 16 }
```

The algorithm parameters are as follows:

```
AuthEncParams ::= SEQUENCE {
    prfAlgorithm   [0] AlgorithmIdentifier DEFAULT PBKDF2,
    encAlgorithm       AlgorithmIdentifier,
    macAlgorithm       AlgorithmIdentifier
    }
```

prfAlgorithm is the PRF algorithm used to convert the master secret into the encryption and MAC keys. The default PRF is [PBKDF2], which in turn has a default PRF algorithm of HMAC-SHA1. When this default setting is used, the PBKDF2-params 'salt' parameter is an empty string, and the 'iterationCount' parameter is one, turning the KDF into a pure PRF.

encAlgorithm is the encryption algorithm and associated parameters to be used to encrypt the content.

macAlgorithm is the MAC algorithm and associated parameters to be used to authenticate/integrity-protect the content.

When the prfAlgorithm AlgorithmIdentifier is used in conjunction with PBKDF2 to specify a PRF other than the default PBKDF2-with-HMAC-SHA1 one, the PBKDF2-params require that two additional algorithm parameters be specified. The 'salt' parameter MUST be an empty (zero-length) string, and the 'iterationCount' parameter MUST be one, since these values aren't used in the PRF process. In their encoded form as used for the PBKDF2-params, these two parameters have the value 08 00 02 01 01.

As a guideline for authors specifying the use of PRFs other than PBKDF2, any additional parameters such as salts, tags, and identification strings SHOULD be set to empty strings, and any iteration count SHOULD be set to one.

4.1. Encrypt-and-Authenticate Message Processing

The randomly generated master secret to be communicated via the RecipientInfo(s) is converted to separate encryption and authentication keys and applied to the encrypt-and-authenticate process as follows. The notation "PRF(key, salt, iterations)" is used to denote an application of the PRF to the given keying value and salt, for the given number of iterations:

1. The MAC algorithm key is derived from the master secret via:

   MAC-K ::= PRF(master_secret, "authentication", 1);

2. The encryption algorithm key is derived from the master secret via:

   Enc-K ::= PRF(master_secret, "encryption", 1);

3. The data is processed as described in [AuthEnv], and specifically since the mechanisms used are a union of EncryptedData and AuthenticatedData, as per [CMS]. The one exception to this is that the EncryptedContentInfo.ContentEncryptionAlgorithmIdentifier data is MACed before the encrypted content is MACed. The EncryptedData processing is applied to the data first, and then the AuthenticatedData processing is applied to the result, so that the nesting is as follows:

   MAC(contentEncrAlgoID || encrypt(content));

4. If authenticated attributes are present, then they are encoded as described in [AuthEnv] and MACed after the encrypted content, so that the processing is as follows:

   MAC(contentEncrAlgoID || encrypt(content) || authAttr);

4.2. Rationale

When choosing between encrypt-and-authenticate and authenticate-and-encrypt, the more secure option is encrypt-and-authenticate. There has been extensive analysis of this in the literature; the best coverage is probably [EncryptThenAuth].

The EncryptedContentInfo.ContentEncryptionAlgorithmIdentifier is the SEQUENCE containing the id-alg-authEnc-128/id-alg-authEnc-256 OBJECT IDENTIFIER and its associated AuthEncParams. This data is MACed exactly as encoded, without any attempt to re-code it into a canonical form like DER.

The EncryptedContentInfo.ContentEncryptionAlgorithmIdentifier must be protected alongside the encrypted content; otherwise, an attacker could manipulate the encrypted data indirectly by manipulating the encryption algorithm parameters, which wouldn't be detected through MACing the encrypted content alone. For example, by changing the encryption IV, it's possible to modify the results of the decryption after the encrypted data has been verified via a MAC check.

The authEnc pseudo-algorithm has two "key sizes" rather than the one-size-fits-all that the PRF impedance-matching would provide. This is done to address real-world experience in the use of AES keys, where users demanded AES-256 alongside AES-128 because of some perception that the former was "twice as good" as the latter. Providing an option for keys that go to 11 avoids potential user acceptance problems when someone notices that the authEnc pseudo-key has "only" 128 bits when they expect their AES keys to be 256 bits long.

Using a fixed-length key rather than making it a user-selectable parameter is done for the same reason as AES's quantised key lengths: there's no benefit to allowing, say, 137-bit keys over basic 128- and 256-bit lengths; it adds unnecessary complexity; if the lengths are user-defined, then there'll always be someone who wants keys that go up to 12. Providing a choice of two commonly used lengths gives users the option of choosing a "better" key size should they feel the need, while not overloading the system with unneeded flexibility.

The use of the PRF AlgorithmIdentifier presents some problems, because it's usually not specified in a manner that allows it to be easily used as a straight KDF. For example, PBKDF2 has the following parameters:

```
PBKDF2-params ::= SEQUENCE {
    salt OCTET STRING,
    iterationCount INTEGER (1..MAX),
    prf AlgorithmIdentifier {{PBKDF2-PRFs}}
                            DEFAULT algid-hmacWithSHA1
    }
```

of which only the prf AlgorithmIdentifier is used here. In order to avoid having to define new AlgorithmIdentifiers for each possible PRF, this specification sets any parameters not required for KDF functionality to no-op values. In the case of PBKDF2, this means that the salt has length zero and the iteration count is set to one, with only the prf AlgorithmIdentifier playing a part in the processing. Although it's not possible to know what form other PRFs-as-KDFs will take, a general note for their application within this specification is that any non-PRF parameters should similarly be set to no-op values.

Specifying a MAC key size gets a bit tricky; most MAC algorithms have some de facto standard key size, and for HMAC algorithms, this is usually the same as the hash output size. For example, for HMAC-MD5, it's 128 bits; for HMAC-SHA1, it's 160 bits; and for HMAC-SHA256, it's 256 bits. Other MAC algorithms also have de facto standard key sizes. For example, for AES-based MACs, it's the AES key size -- 128 bits for AES-128 and 256 bits for AES-256. This situation makes

it difficult to specify the key size in a normative fashion, since it's dependent on the algorithm type that's being used. If there is any ambiguity over which key size should be used, then it's RECOMMENDED that either the size be specified explicitly in the macAlgorithm AlgorithmIdentifier or that an RFC or similar standards document be created that makes the key sizes explicit.

As with other uses of PRFs for crypto impedance-matching in protocols, like IPsec, SSL/TLS, and SSH, the amount of input to the PRF generally doesn't match the amount of output. The general philosophical implications of this are covered in various analyses of the properties and uses of PRFs. If you're worried about this, then you can try and approximately match the authEnc "key size" to the key size of the encryption algorithm being used, although even there, a perfect match for algorithms like Blowfish (448 bits) or RC5 (832 bits) is going to be difficult.

The term "master secret" comes from its use in SSL/TLS, which uses a similar PRF-based mechanism to convert its master_secret value into encryption and MAC keys (as do SSH and IPsec). The master_secret value isn't a key in the conventional sense, but merely a secret value that's then used to derive two (or, in the cases of SSL/TLS, SSH, and IPsec, several) keys and related cryptovariables.

Apart from the extra step added to key management, all of the processing is already specified as part of the definition of the standard CMS content-types Encrypted/EnvelopedData and AuthenticatedData. This significantly simplifies both the specification and the implementation task, as no new content-processing mechanisms are introduced.

4.3. Test Vectors

The following test vectors may be used to verify an implementation of MAC-authenticated encryption. This represents a text string encrypted and authenticated using the ever-popular password "password" via CMS PasswordRecipientInfo. The encryption algorithm used for the first value is triple DES, whose short block size (compared to AES) makes it easier to corrupt arbitrary bytes for testing purposes within the self-healing Cipher Block Chaining (CBC) mode, which will result in correct decryption but a failed MAC check. The encryption algorithm used for the second value is AES.

For the triple DES-encrypted data, corrupting a byte at positions 192-208 can be used to check that payload-data corruption is detected, and corrupting a byte at positions 168-174 can be used to

check that metadata corruption is detected. The corruption in these byte ranges doesn't affect normal processing and so wouldn't normally be detected.

The test data has the following characteristics:

version is set to 0.

originatorInfo isn't needed and is omitted.

recipientInfo uses passwordRecipientInfo to allow easy testing with a fixed text string.

authEncryptedContentInfo uses the authEnc128 pseudo-algorithm with a key of 128 bits used to derive triple DES/AES and HMAC-SHA1 keys.

authAttrs aren't used and are omitted.

mac is the 20-byte HMAC-SHA1 MAC value.

unauthAttrs aren't used and are omitted.

```
  0  227: SEQUENCE {
  3   11:   OBJECT IDENTIFIER id-ct-authEnvelopedData
                              (1 2 840 113549 1 9 16 1 23)
 16  211:   [0] {
 19  208:     SEQUENCE {
 22    1:       INTEGER 0
 25   97:       SET {
 27   95:         [3] {
 29    1:           INTEGER 0
 32   27:           [0] {
 34    9:             OBJECT IDENTIFIER pkcs5PBKDF2
                                        (1 2 840 113549 1 5 12)
 45   14:             SEQUENCE {
 47    8:               OCTET STRING B7 EB 23 A7 6B D2 05 16
 57    2:               INTEGER 5000
        :               }
        :             }
 61   35:           SEQUENCE {
 63   11:             OBJECT IDENTIFIER pwriKEK
                                        (1 2 840 113549 1 9 16 3 9)
```

```
    76   20:             SEQUENCE {
    78    8:               OBJECT IDENTIFIER des-EDE3-CBC
                                     (1 2 840 113549 3 7)
    88    8:               OCTET STRING 66 91 02 45 6B 73 BB 99
           :               }
           :             }
    98   24:           OCTET STRING
           :             30 A3 7A B5 D8 F2 87 50 EC 41 04 AE 89 99 26 F0
           :             2E AE 4F E3 F3 52 2B A3
           :           }
           :         }
   124   82:       SEQUENCE {
   126    9:         OBJECT IDENTIFIER data (1 2 840 113549 1 7 1)
   137   51:         SEQUENCE {
   139   11:           OBJECT IDENTIFIER authEnc128
                                   (1 2 840 113549 1 9 16 3 15)
   152   36:           SEQUENCE {
   154   20:             SEQUENCE {
   156    8:               OBJECT IDENTIFIER des-EDE3-CBC
                                     (1 2 840 113549 3 7)
   166    8:               OCTET STRING D2 D0 81 71 4D 3D 9F 11
           :               }
   176   12:             SEQUENCE {
   178    8:               OBJECT IDENTIFIER hmacSHA (1 3 6 1 5 5 8 1 2)
   188    0:               NULL
           :               }
           :             }
           :           }
   190   16:         [0] 3A C6 06 61 41 5D 00 7D 11 35 CD 69 E1 56 CA 10
           :         }
   208   20:       OCTET STRING
           :         33 65 E8 F0 F3 07 06 86 1D A8 47 2C 6D 3A 1D 94
           :         21 40 64 7E
           :       }
           :     }
           :   }
```

```
-----BEGIN PKCS7-----
MIHjBgsqhkiG9w0BCRABF6CB0zCB0AIBADFho18CAQCgGwYJKoZIhvcNAQUMMA4E
CLfrI6dr0gUWAgITiDAjBgsqhkiG9w0BCRADCTAUBggqhkiG9w0DBwQIZpECRWtz
u5kEGDCjerXY8odQ7EEEromZJvAurk/j81IrozBSBgkqhkiG9w0BBwEwMwYLKoZI
hvcNAQkQAw8wJDAUBggqhkiG9w0DBwQI0tCBcU09nxEwDAYIKwYBBQUIAQIFAIAQ
OsYGYUFdAH0RNc1p4VbKEAQUM2Xo8PMHBoYdqEcsbTodlCFAZH4=
-----END PKCS7-----
```

```
   0  253: SEQUENCE {
   3   11:   OBJECT IDENTIFIER id-ct-authEnvelopedData
                               (1 2 840 113549 1 9 16 1 23)
  16  237:   [0] {
  19  234:     SEQUENCE {
  22    1:       INTEGER 0
  25  114:       SET {
  27  112:         [3] {
  29    1:           INTEGER 0
  32   27:           [0] {
  34    9:             OBJECT IDENTIFIER pkcs5PBKDF2
                                         (1 2 840 113549 1 5 12)
  45   14:             SEQUENCE {
  47    8:               OCTET STRING E7 B7 87 DF 82 1D 12 CC
  57    2:               INTEGER 5000
        :               }
        :             }
  61   44:           SEQUENCE {
  63   11:             OBJECT IDENTIFIER pwriKEK
                                         (1 2 840 113549 1 9 16 3 9)
  76   29:             SEQUENCE {
  78    9:               OBJECT IDENTIFIER aes128-CBC
                                           (2 16 840 1 101 3 4 1 2)
  89   16:               OCTET STRING
        :               11 D9 5C 52 0A 3A BF 22 B2 30 70 EF F4 7D 6E F6
        :               }
        :             }
 107   32:           OCTET STRING
        :             18 39 22 27 C3 C2 2C 2A A6 9F 2A B0 77 24 75 AA
        :             D8 58 9C CD BB 4C AE D3 0D C2 CB 1D 83 94 6C 37
        :           }
        :         }
 141   91:       SEQUENCE {
 143    9:         OBJECT IDENTIFIER data (1 2 840 113549 1 7 1)
 154   60:         SEQUENCE {
 156   11:           OBJECT IDENTIFIER authEnc128
                                       (1 2 840 113549 1 9 16 3 15)
 169   45:           SEQUENCE {
 171   29:             SEQUENCE {
 173    9:               OBJECT IDENTIFIER aes128-CBC
                                           (2 16 840 1 101 3 4 1 2)
 184   16:               OCTET STRING
        :               B7 25 02 76 84 3C 58 1B A5 30 E2 40 27 EE C3 06
        :               }
```

```
 202   12:             SEQUENCE {
 204    8:               OBJECT IDENTIFIER hmacSHA (1 3 6 1 5 5 8 1 2)
 214    0:               NULL
        :               }
        :             }
        :           }
 216   16:         [0] 98 36 0F 0C 79 62 36 B5 2D 2D 9E 1C 62 85 1E 10
        :         }
 234   20:       OCTET STRING
        :         88 A4 C1 B2 BA 78 1B CA F9 14 B0 E5 FC D1 8D F8
        :         02 E2 B2 9E
        :       }
        :     }
        :   }
```

```
-----BEGIN PKCS7-----
MIH9BgsqhkiG9w0BCRABF6CB7TCB6gIBADFyo3ACAQCgGwYJKoZIhvcNAQUMMA4E
COe3h9+CHRLMAgITiDAsBgsqhkiG9w0BCRADCTAdBglghkgBZQMEAQIEEBHZXFIK
Or8isjBw7/R9bvYEIBg5IifDwiwqpp8qsHckdarYWJzNu0yu0w3Cyx2DlGw3MFsG
CSqGSIb3DQEHATA8BgsqhkiG9w0BCRADDzAtMB0GCWCGSAFlAwQBAgQQtyUCdoQ8
WBulMOJAJ+7DBjAMBggrBgEFBQgBAgUAgBCYNg8MeWI2tS0tnhxihR4QBBSIpMGy
ungbyvkUsOX80Y34AuKyng==
-----END PKCS7-----
```

5. SMIMECapabilities Attribute

An S/MIME client SHOULD announce the set of cryptographic functions that it supports by using the SMIMECapabilities attribute [SMIME]. If the client wishes to indicate support for MAC-authenticated encryption, the capabilities attribute MUST contain the authEnc128 and/or authEnc256 OID specified above with algorithm parameters ABSENT. The other algorithms used in the authEnc algorithm, such as the MAC and encryption algorithm, are selected based on the presence of these algorithms in the SMIMECapabilities attribute or by mutual agreement.

6. Security Considerations

Unlike other CMS authenticated-data mechanisms, such as SignedData and AuthenticatedData, AuthEnv's primary transformation isn't authentication but encryption; so AuthEnvData may decrypt successfully (in other words, the primary data transformation present in the mechanism will succeed), but the secondary function of authentication using the MAC value that follows the encrypted data could still fail. This can lead to a situation in which an implementation might output decrypted data before it reaches and verifies the MAC value. In other words, decryption is performed inline and the result is available immediately, while the

authentication result isn't available until all of the content has been processed. If the implementation prematurely provides data to the user and later comes back to inform them that the earlier data was, in retrospect, tainted, this may cause users to act prematurely on the tainted data.

This situation could occur in a streaming implementation where data has to be made available as soon as possible (so that the initial plaintext is emitted before the final ciphertext and MAC value are read), or one where the quantity of data involved rules out buffering the recovered plaintext until the MAC value can be read and verified. In addition, an implementation that tries to be overly helpful may treat missing non-payload trailing data as non-fatal, allowing an attacker to truncate the data somewhere before the MAC value and thereby defeat the data authentication. This is complicated even further by the fact that an implementation may not be able to determine, when it encounters truncated data, whether the remainder (including the MAC value) will arrive presently (a non-failure) or whether it's been truncated by an attacker and should therefore be treated as a MAC failure. (Note that this same issue affects other types of data authentication like signed and MACed data as well, since an over-optimistic implementation may return data to the user before checking for a verification failure is possible.)

The exact solution to these issues is somewhat implementation-specific, with some suggested mitigations being as follows: implementations should buffer the entire message if possible and verify the MAC before performing any decryption. If this isn't possible due to streaming or message-size constraints, then implementations should consider breaking long messages into a sequence of smaller ones, each of which can be processed atomically as above. If even this isn't possible, then implementations should make obvious to the caller or user that an authentication failure has occurred and that the previously returned or output data shouldn't be used. Finally, any data-formatting problem, such as obviously truncated data or missing trailing data, should be treated as a MAC verification failure even if the rest of the data was processed correctly.

7. IANA Considerations

This document contains two algorithm identifiers defined by the S/MIME Working Group Registrar in an arc delegated by RSA to the S/MIME Working Group: iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) modules(0).

8. Acknowledgements

The author would like to thank Jim Schaad and the members of the S/MIME mailing list for their feedback on this document, and David Ireland for help with the test vectors.

9. References

9.1. Normative References

[AuthEnv] Housley, R., "Cryptographic Message Syntax (CMS) Authenticated-Enveloped-Data Content Type", RFC 5083, November 2007.

[CMS] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, September 2009.

[PBKDF2] Kaliski, B., "PKCS #5: Password-Based Cryptography Specification Version 2", RFC 2898, September 2000.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[SMIME] Ramsdell, B. and S. Turner, "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification", RFC 5751, January 2010.

9.2. Informative References

[EncryptThenAuth] Krawczyk, H., "The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?)", Springer-Verlag LNCS 2139, August 2001.

[Garfinkel] Garfinkel, S., "Design Principles and Patterns for Computer Systems That Are Simultaneously Secure and Usable", May 2005.

[HKDF] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, May 2010.

[IPsec] Kent, S. and K. Seo, "Security Architecture for the Internet Protocol", RFC 4301, December 2005.

[OpenPGP] Callas, J., Donnerhacke, L., Finney, H., Shaw, D., and R. Thayer, "OpenPGP Message Format", RFC 4880, November 2007.

[SSH] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Transport Layer Protocol", RFC 4253, January 2006.

[TLS] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

Author's Address

Peter Gutmann
University of Auckland
Department of Computer Science
New Zealand

EMail: pgut001@cs.auckland.ac.nz