Network Working Group
Request for Comments: 2104
Category: Informational

H. Krawczyk
IBM
M. Bellare
UCSD
R. Canetti
IBM

February 1997

HMAC: Keyed-Hashing for Message Authentication

Status of This Memo

This memo provides information for the Internet community. This memo does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

Abstract

This document describes HMAC, a mechanism for message authentication using cryptographic hash functions. HMAC can be used with any iterative cryptographic hash function, e.g., MD5, SHA-1, in combination with a secret shared key. The cryptographic strength of HMAC depends on the properties of the underlying hash function.

1. Introduction

Providing a way to check the integrity of information transmitted over or stored in an unreliable medium is a prime necessity in the world of open computing and communications. Mechanisms that provide such integrity check based on a secret key are usually called "message authentication codes" (MAC). Typically, message authentication codes are used between two parties that share a secret key in order to validate information transmitted between these parties. In this document we present such a MAC mechanism based on cryptographic hash functions. This mechanism, called HMAC, is based on work by the authors [BCK1] where the construction is presented and cryptographically analyzed. We refer to that work for the details on the rationale and security analysis of HMAC, and its comparison to other keyed-hash methods.

HMAC can be used in combination with any iterated cryptographic hash function. MD5 and SHA-1 are examples of such hash functions. HMAC also uses a secret key for calculation and verification of the message authentication values. The main goals behind this construction are

* To use, without modifications, available hash functions. In particular, hash functions that perform well in software, and for which code is freely and widely available.

* To preserve the original performance of the hash function without incurring a significant degradation.

* To use and handle keys in a simple way.

* To have a well understood cryptographic analysis of the strength of the authentication mechanism based on reasonable assumptions on the underlying hash function.

* To allow for easy replaceability of the underlying hash function in case that faster or more secure hash functions are found or required.

This document specifies HMAC using a generic cryptographic hash function (denoted by H). Specific instantiations of HMAC need to define a particular hash function. Current candidates for such hash functions include SHA-1 [SHA], MD5 [MD5], RIPEMD-128/160 [RIPEMD]. These different realizations of HMAC will be denoted by HMAC-SHA1, HMAC-MD5, HMAC-RIPEMD, etc.

Note: To the date of writing of this document MD5 and SHA-1 are the most widely used cryptographic hash functions. MD5 has been recently shown to be vulnerable to collision search attacks [Dobb]. This attack and other currently known weaknesses of MD5 do not compromise the use of MD5 within HMAC as specified in this document (see [Dobb]); however, SHA-1 appears to be a cryptographically stronger function. To this date, MD5 can be considered for use in HMAC for applications where the superior performance of MD5 is critical. In any case, implementers and users need to be aware of possible cryptanalytic developments regarding any of these cryptographic hash functions, and the eventual need to replace the underlying hash function. (See section 6 for more information on the security of HMAC.)

2. Definition of HMAC

The definition of HMAC requires a cryptographic hash function, which we denote by H, and a secret key K. We assume H to be a cryptographic hash function where data is hashed by iterating a basic compression function on blocks of data. We denote by B the byte-length of such blocks (B=64 for all the above mentioned examples of hash functions), and by L the byte-length of hash outputs (L=16 for MD5, L=20 for SHA-1). The authentication key K can be of any length up to B, the block length of the hash function. Applications that use keys longer than B bytes will first hash the key using H and then use the resultant L byte string as the actual key to HMAC. In any case the minimal recommended length for K is L bytes (as the hash output length). See section 3 for more information on keys.

We define two fixed and different strings ipad and opad as follows (the 'i' and 'o' are mnemonics for inner and outer):

```
                ipad = the byte 0x36 repeated B times
               opad = the byte 0x5C repeated B times.
```

To compute HMAC over the data `text' we perform

```
                 H(K XOR opad, H(K XOR ipad, text))
```

Namely,

(1) append zeros to the end of K to create a B byte string
    (e.g., if K is of length 20 bytes and B=64, then K will be
     appended with 44 zero bytes 0x00)
(2) XOR (bitwise exclusive-OR) the B byte string computed in step
    (1) with ipad
(3) append the stream of data 'text' to the B byte string resulting
    from step (2)
(4) apply H to the stream generated in step (3)
(5) XOR (bitwise exclusive-OR) the B byte string computed in
    step (1) with opad
(6) append the H result from step (4) to the B byte string
    resulting from step (5)
(7) apply H to the stream generated in step (6) and output
    the result

For illustration purposes, sample code based on MD5 is provided as an appendix.

3. Keys

The key for HMAC can be of any length (keys longer than B bytes are first hashed using H). However, less than L bytes is strongly discouraged as it would decrease the security strength of the function. Keys longer than L bytes are acceptable but the extra length would not significantly increase the function strength. (A longer key may be advisable if the randomness of the key is considered weak.)

Keys need to be chosen at random (or using a cryptographically strong pseudo-random generator seeded with a random seed), and periodically refreshed. (Current attacks do not indicate a specific recommended frequency for key changes as these attacks are practically infeasible. However, periodic key refreshment is a fundamental security practice that helps against potential weaknesses of the function and keys, and limits the damage of an exposed key.)

4. Implementation Note

HMAC is defined in such a way that the underlying hash function H can be used with no modification to its code. In particular, it uses the function H with the pre-defined initial value IV (a fixed value specified by each iterative hash function to initialize its compression function). However, if desired, a performance improvement can be achieved at the cost of (possibly) modifying the code of H to support variable IVs.

The idea is that the intermediate results of the compression function on the B-byte blocks (K XOR ipad) and (K XOR opad) can be precomputed only once at the time of generation of the key K, or before its first use. These intermediate results are stored and then used to initialize the IV of H each time that a message needs to be authenticated. This method saves, for each authenticated message, the application of the compression function of H on two B-byte blocks (i.e., on (K XOR ipad) and (K XOR opad)). Such a savings may be significant when authenticating short streams of data. We stress that the stored intermediate values need to be treated and protected the same as secret keys.

Choosing to implement HMAC in the above way is a decision of the local implementation and has no effect on inter-operability.

5. Truncated output

A well-known practice with message authentication codes is to truncate the output of the MAC and output only part of the bits (e.g., [MM, ANSI]). Preneel and van Oorschot [PV] show some analytical advantages of truncating the output of hash-based MAC functions. The results in this area are not absolute as for the overall security advantages of truncation. It has advantages (less information on the hash result available to an attacker) and disadvantages (less bits to predict for the attacker). Applications of HMAC can choose to truncate the output of HMAC by outputting the t leftmost bits of the HMAC computation for some parameter t (namely, the computation is carried in the normal way as defined in section 2 above but the end result is truncated to t bits). We recommend that the output length t be not less than half the length of the hash output (to match the birthday attack bound) and not less than 80 bits (a suitable lower bound on the number of bits that need to be predicted by an attacker). We propose denoting a realization of HMAC that uses a hash function H with t bits of output as HMAC-H-t. For example, HMAC-SHA1-80 denotes HMAC computed using the SHA-1 function and with the output truncated to 80 bits. (If the parameter t is not specified, e.g. HMAC-MD5, then it is assumed that all the bits of the hash are output.)

6. Security

The security of the message authentication mechanism presented here depends on cryptographic properties of the hash function H: the resistance to collision finding (limited to the case where the initial value is secret and random, and where the output of the function is not explicitly available to the attacker), and the message authentication property of the compression function of H when applied to single blocks (in HMAC these blocks are partially unknown to an attacker as they contain the result of the inner H computation and, in particular, cannot be fully chosen by the attacker).

These properties, and actually stronger ones, are commonly assumed for hash functions of the kind used with HMAC. In particular, a hash function for which the above properties do not hold would become unsuitable for most (probably, all) cryptographic applications, including alternative message authentication schemes based on such functions. (For a complete analysis and rationale of the HMAC function the reader is referred to [BCK1].)

Given the limited confidence gained so far as for the cryptographic strength of candidate hash functions, it is important to observe the following two properties of the HMAC construction and its secure use for message authentication:

1. The construction is independent of the details of the particular hash function H in use and then the latter can be replaced by any other secure (iterative) cryptographic hash function.

2. Message authentication, as opposed to encryption, has a "transient" effect. A published breaking of a message authentication scheme would lead to the replacement of that scheme, but would have no adversarial effect on information authenticated in the past. This is in sharp contrast with encryption, where information encrypted today may suffer from exposure in the future if, and when, the encryption algorithm is broken.

The strongest attack known against HMAC is based on the frequency of collisions for the hash function H ("birthday attack") [PV,BCK2], and is totally impractical for minimally reasonable hash functions.

As an example, if we consider a hash function like MD5 where the output length equals L=16 bytes (128 bits) the attacker needs to acquire the correct message authentication tags computed (with the _same_ secret key K!) on about 2**64 known plaintexts. This would require the processing of at least 2**64 blocks under H, an impossible task in any realistic scenario (for a block length of 64 bytes this would take 250,000 years in a continuous 1Gbps link, and without changing the secret key K during all this time). This attack could become realistic only if serious flaws in the collision behavior of the function H are discovered (e.g. collisions found after 2**30 messages). Such a discovery would determine the immediate replacement of the function H (the effects of such failure would be far more severe for the traditional uses of H in the context of digital signatures, public key certificates, etc.).

Note: this attack needs to be strongly contrasted with regular collision attacks on cryptographic hash functions where no secret key is involved and where 2**64 off-line parallelizable (!) operations suffice to find collisions. The latter attack is approaching feasibility [VW] while the birthday attack on HMAC is totally impractical. (In the above examples, if one uses a hash function with, say, 160 bit of output then 2**64 should be replaced by 2**80.)

A correct implementation of the above construction, the choice of random (or cryptographically pseudorandom) keys, a secure key exchange mechanism, frequent key refreshments, and good secrecy protection of keys are all essential ingredients for the security of the integrity verification mechanism provided by HMAC.

Appendix -- Sample Code

For the sake of illustration we provide the following sample code for the implementation of HMAC-MD5 as well as some corresponding test vectors (the code is based on MD5 code as described in [MD5]).

```
/*
** Function: hmac_md5
*/

void
hmac_md5(text, text_len, key, key_len, digest)
unsigned char*  text;                /* pointer to data stream */
int             text_len;            /* length of data stream */
unsigned char*  key;                 /* pointer to authentication key */
int             key_len;             /* length of authentication key */
caddr_t         digest;              /* caller digest to be filled in */

{
        MD5_CTX context;
        unsigned char k_ipad[65];    /* inner padding -
                                      * key XORd with ipad
                                      */
        unsigned char k_opad[65];    /* outer padding -
                                      * key XORd with opad
                                      */
        unsigned char tk[16];
        int i;
        /* if key is longer than 64 bytes reset it to key=MD5(key) */
        if (key_len > 64) {

                MD5_CTX      tctx;

                MD5Init(&tctx);
                MD5Update(&tctx, key, key_len);
                MD5Final(tk, &tctx);

                key = tk;
                key_len = 16;
        }

        /*
         * the HMAC_MD5 transform looks like:
         *
         * MD5(K XOR opad, MD5(K XOR ipad, text))
         *
         * where K is an n byte key
         * ipad is the byte 0x36 repeated 64 times
```

```
         * opad is the byte 0x5c repeated 64 times
         * and text is the data being protected
         */

        /* start out by storing key in pads */
        bzero( k_ipad, sizeof k_ipad);
        bzero( k_opad, sizeof k_opad);
        bcopy( key, k_ipad, key_len);
        bcopy( key, k_opad, key_len);

        /* XOR key with ipad and opad values */
        for (i=0; i<64; i++) {
                k_ipad[i] ^= 0x36;
                k_opad[i] ^= 0x5c;
        }
        /*
         * perform inner MD5
         */
        MD5Init(&context);                   /* init context for 1st
                                              * pass */
        MD5Update(&context, k_ipad, 64)      /* start with inner pad */
        MD5Update(&context, text, text_len); /* then text of datagram */
        MD5Final(digest, &context);          /* finish up 1st pass */
        /*
         * perform outer MD5
         */
        MD5Init(&context);                   /* init context for 2nd
                                              * pass */
        MD5Update(&context, k_opad, 64);     /* start with outer pad */
        MD5Update(&context, digest, 16);     /* then results of 1st
                                              * hash */
        MD5Final(digest, &context);          /* finish up 2nd pass */
}
```

Test Vectors (Trailing '\0' of a character string not included in test):

```
  key =         0x0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b
  key_len =     16 bytes
  data =        "Hi There"
  data_len =    8  bytes
  digest =      0x9294727a3638bb1c13f48ef8158bfc9d

  key =         "Jefe"
  data =        "what do ya want for nothing?"
  data_len =    28 bytes
  digest =      0x750c783e6ab0b503eaa86e310a5db738

  key =         0xAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
key_len       16 bytes
data =        0xDDDDDDDDDDDDDDDDDDDD...
              ..DDDDDDDDDDDDDDDDDDDD...
              ..DDDDDDDDDDDDDDDDDDDD...
              ..DDDDDDDDDDDDDDDDDDDD...
              ..DDDDDDDDDDDDDDDDDDDD
data_len =    50 bytes
digest =      0x56be34521d144c88dbb8c733f0e8b3f6
```

Acknowledgments

Pau-Chen Cheng, Jeff Kraemer, and Michael Oehler, have provided useful comments on early drafts, and ran the first interoperability tests of this specification. Jeff and Pau-Chen kindly provided the sample code and test vectors that appear in the appendix. Burt Kaliski, Bart Preneel, Matt Robshaw, Adi Shamir, and Paul van Oorschot have provided useful comments and suggestions during the investigation of the HMAC construction.

References

[ANSI] ANSI X9.9, "American National Standard for Financial Institution Message Authentication (Wholesale)," American Bankers Association, 1981. Revised 1986.

[Atk] Atkinson, R., "IP Authentication Header", RFC 1826, August 1995.

[BCK1] M. Bellare, R. Canetti, and H. Krawczyk, "Keyed Hash Functions and Message Authentication", Proceedings of Crypto'96, LNCS 1109, pp. 1-15. (http://www.research.ibm.com/security/keyed-md5.html)

[BCK2] M. Bellare, R. Canetti, and H. Krawczyk, "Pseudorandom Functions Revisited: The Cascade Construction", Proceedings of FOCS'96.

[Dobb] H. Dobbertin, "The Status of MD5 After a Recent Attack", RSA Labs' CryptoBytes, Vol. 2 No. 2, Summer 1996. http://www.rsa.com/rsalabs/pubs/cryptobytes.html

[PV] B. Preneel and P. van Oorschot, "Building fast MACs from hash functions", Advances in Cryptology -- CRYPTO'95 Proceedings, Lecture Notes in Computer Science, Springer-Verlag Vol.963, 1995, pp. 1-14.

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

[MM] Meyer, S. and Matyas, S.M., Cryptography, New York Wiley, 1982.

[RIPEMD] H. Dobbertin, A. Bosselaers, and B. Preneel, "RIPEMD-160: A strengthened version of RIPEMD", Fast Software Encryption, LNCS Vol 1039, pp. 71-82. ftp://ftp.esat.kuleuven.ac.be/pub/COSIC/bosselae/ripemd/.

[SHA] NIST, FIPS PUB 180-1: Secure Hash Standard, April 1995.

[Tsu] G. Tsudik, "Message authentication with one-way hash functions", In Proceedings of Infocom'92, May 1992. (Also in "Access Control and Policy Enforcement in Internetworks", Ph.D. Dissertation, Computer Science Department, University of Southern California, April 1991.)

[VW] P. van Oorschot and M. Wiener, "Parallel Collision Search with Applications to Hash Functions and Discrete Logarithms", Proceedings of the 2nd ACM Conf. Computer and Communications Security, Fairfax, VA, November 1994.

Authors' Addresses

Hugo Krawczyk
IBM T.J. Watson Research Center
P.O.Box 704
Yorktown Heights, NY 10598

EMail: hugo@watson.ibm.com

Mihir Bellare
Dept of Computer Science and Engineering
Mail Code 0114
University of California at San Diego
9500 Gilman Drive
La Jolla, CA 92093

EMail: mihir@cs.ucsd.edu

Ran Canetti
IBM T.J. Watson Research Center
P.O.Box 704
Yorktown Heights, NY 10598

EMail: canetti@watson.ibm.com