# VMAC: Message Authentication Code using Universal Hashing
**<draft-krovetz-vmac-00.txt>**

By submitting this Internet-Draft, each author represents that any applicable patent or other IPR claims of which he or she is aware have been or will be disclosed, and any of which he or she becomes aware will be disclosed, in accordance with Section 6 of BCP 79.

Status of this Memo

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/1id-abstracts.html

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

Abstract

This specification describes how to generate an authentication tag using the VMAC message authentication algorithm. VMAC is designed to have exceptional performance in software on 64-bit CPU architectures while performing well on 32-bit architectures. Measured speeds are as low as one-half CPU cycle per byte on the 64-bit Intel Core 2 architecture, and under five cycles per byte on recent 32-bit PowerPC and Intel processors.

To generate the authentication tag on a given message, a "universal" hash function is applied to the message and key to produce a short, fixed-length hash value, and this hash value is then xor'ed with a key-derived pseudorandom pad. VMAC tags can be either 64 or 128 bits in length and have proven forgery probabilities on the order of 1/2^59 and 1/2^94, respectively.

## Table of Contents

## 1 Introduction

VMAC is a message authentication code (MAC) algorithm designed for high performance. It is backed by a rigorous formal analysis, and there are no intellectual property claims made by any of the authors to any ideas used in its design.

VMAC is a MAC in the style of Wegman and Carter [4, 6]. A fast "universal" hash function is used to hash an input message M into a short string. This short string is then masked by xor'ing with a pseudorandom pad, resulting in the VMAC tag. Security depends on the sender and receiver sharing a randomly-chosen secret hash function and pseudorandom pad. This is achieved by using keyed hash function H and pseudorandom function F. A tag is generated by performing the computation

```
  Tag = H_K1(M) xor F_K2(Nonce)
```

where K1 and K2 are secret random keys shared by sender and receiver, and Nonce is a value that changes with each generated tag. The receiver needs to know which nonce was used by the sender, so some method of synchronizing nonces needs to be used. This can be done by explicitly sending the nonce along with the message and tag, or agreeing upon the use of some other non-repeating value such as a sequence number. The nonce need not be kept secret, but care needs to be taken to ensure that, over the lifetime of a VMAC key, a different nonce is used with each message.

VMAC uses a function, called VHASH (also specified in this document), as the keyed hash function H and uses a pseudorandom function F whose default implementation uses the AES algorithm. VMAC is designed to produce 64- or 128-bit tags, depending on the desired security level. The theory of Wegman-Carter MACs and the analysis of VMAC show that if one "instantiates" VMAC with truly random keys and pads then the probability that an attacker (even a computationally unbounded one) produces a correct tag for messages of its choosing upto j bits in length is less than 1/2^59 or 1/2^117 when the tags output by VMAC are of length 64 or 128 bits, respectively (here the symbol ^ represents exponentiation). When an attacker makes N forgery attempts the probability of getting one or more tags right increases linearly to about N/2^59 or Nj/2^117. In a real implementation of VMAC, using AES to produce keys and pads, the forgery probabilities listed above increase by a small amount related to the security of AES. As long as AES is secure this small additive term is insignificant for any practical attack. See Section 6.2 for more details. Analysis relevant to VMAC security is in [5].

VMAC performs best in environments where 64-bit quantities are

efficiently multiplied into 128-bit results. In producing 64-bit tags on an Intel Core 2 CPU, VMAC processes messages at a rate of about one-half CPU cycle per byte on messages of two kilobytes. On a 32-bit Intel Core CPU, which does not support 64-bit multiplication well, VMAC achieves a rate of under five cycles per byte. On shorter messages VMAC still performs well: about two cycles per byte on 64 byte messages on the Core 2. Tags of 128 bits require slightly less than twice the computation as 64-bit tags.

Optimized source code, performance data and papers concerning VMAC can be found at http://www.fastcrypto.org/vmac.

## 2 Notation and basic operations

The specification of VMAC involves the manipulation of strings and numbers. String variables are denoted with an initial upper-case letter, whereas numeric variables are denoted in all lower case. The algorithms of VMAC are denoted in all upper-case letters. Simple functions, like those for string-length and string-xor, are written in all lower case.

Whenever a variable is followed by an underscore ("_"), the underscore is intended to denote a subscript, with the subscripted expression evaluated to resolve the meaning of the variable. For example, if i=2, then M_{2 * i} refers to the variable M_4.

### 2.1 Operations on strings

Messages to be hashed are viewed as strings of bits. The following notation is used to manipulate these strings.

```
bitlength(S): The length of string S in bits.

zeros(n):     The string made of n zero-bits.

S xor T:      The string which is the bitwise exclusive-or of S and
              T.  Strings S and T always have the same length.

S[i]:         The i-th bit of the string S (indices begin at 1).

S[i...j]:     The substring of S consisting of bits i through j.

S || T:       The string S concatenated with string T.

zeropad(S,n): The string S, padded with zero-bits to the nearest
              multiple of n bits in length.  If S is empty or
```

already a multiple of n in length, nothing is appended. Formally, zeropad(S,n) = S || T, where T is the shortest string of zero-bits so that bitlength(S || T) is a multiple of n.

## 2.2 Operations on integers

Standard notation is used for most mathematical operations, such as "*" for multiplication, "+" for addition and "mod" for modular reduction. Some less standard notations are defined here.

a^i: The integer a raised to the i-th power.

ceil(x): The smallest integer not less than x.

prime(n): The largest prime number less than 2^n.

The prime numbers used in VMAC are:

| n | prime(n) [Decimal] | prime(n) [Hexadecimal] |
|---|---|---|
| 61 | 2^61 - 1 | 0x1FFFFFFF FFFFFFFF |
| 127 | 2^127 - 1 | 0x7FFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF |

## 2.3 String-Integer conversion operations

Conversion between strings and integers is done using the following functions. Each function treats initial bits as more significant than later ones.

str2uint(S): The non-negative integer whose binary representation is the string S. More formally, if S is t bits long then str2uint(S) = 2^{t-1} * S[1] + 2^{t-2} * S[2] + ... + 2^{1} * S[t-1] + S[t].

uint2str(n,i): The i-bit string S so that str2uint(S) = n.

## 2.4 Mathematical operations on strings

One of the primary operations in VMAC is addition and multiplication of strings. The operations "+_64", "+_128" and "*_128" are defined

"S +_64 T" as uint2str(str2uint(S) + str2uint(T) mod 2^64, 64),

"S +_128 T" as uint2str(str2uint(S) + str2uint(T) mod 2^128, 128),
"S *_128 T" as uint2str(str2uint(S) * str2uint(T) mod 2^128, 128).

These operations correspond well with addition and multiplication operations which are performed efficiently by modern computers.

### 2.5 ENDIAN-SWAP: Adjusting endian orientation

Message data is read little-endian to speed tag generation on little-endian computers.

#### 2.5.1 ENDIAN-SWAP Algorithm

Input:
  S, string with bitlength divisible by 64.
Output:
  T, string S with each 64-bit substring endian-reversed.

Compute T using the following algorithm.

```
//
// Partition S into 64-bit substrings
//
n = bitlength(S) / 64
Let S_1, S_2, ..., S_n be strings of length 64 bits
   so that S_1 || S_2 || ... || S_n = S.

//
// Endian-reverse each, and build-up T
//
T = <empty string>
for i = 1 to n do
  Let W_1, W_2, ..., W_8  be strings of length 8 bits
     so that W_1 || W_2 || ... || W_8 = S_i
  SReversed_i = W_4 || W_3 || ... || W_1
  T = T || SReversed_i
end for

Return T
```

## 3 Key and pad derivation functions

Pseudorandom bits are needed internally by VHASH and at the time of tag generation.  The functions listed in this section use a block cipher to generate these bits.

### 3.1 Block cipher choice

VMAC uses the services of a block cipher. The selection of a block cipher defines the following constants and functions.

BLOCKLEN — The length, in bits, of the plaintext block on which the block cipher operates.

KEYLEN — The block cipher's key length, in bits.

ENCIPHER(K,P) — The application of the block cipher on P (a string of BLOCKLEN bits) using key K (a string of KEYLEN bits).

As an example, if AES is used with 192-bit keys, then BLOCKLEN would equal 128 (because AES employs 128-bit blocks), KEYLEN would equal 192, and ENCIPHER would refer to the AES function for 192-bit AES keys.

Unless specified otherwise, AES with 128-bit keys shall be assumed to be the chosen block cipher for VMAC. Only if explicitly specified otherwise, and agreed by communicating parties, shall some other block cipher be used. In any case, BLOCKLEN must be at least 128. AES is defined in another document [1].

### 3.2 KDF: Key-derivation function

The key-derivation function generates pseudorandom bits used to key the hash functions.

#### 3.2.1 KDF Algorithm

```
Input:
  K, string with bitlength KEYLEN.
  index, an integer in the range 0...255.
  numbits, a non-negative integer.
Output:
  Y, string with bitlength numbits.
```

Compute Y using the following algorithm.

```
//
// Calculate number of block cipher iterations
//
n = ceil(numbits / BLOCKLEN)
Y = <empty string>
```

```
     //
     // Build Y using block cipher in a counter mode
     //
     for i = 0 to (n-1) do
       T = uint2str(index, 8) || uint2str(i, BLOCKLEN-8)
       Y = Y || ENCIPHER(K, T)
     end for
     Y = Y[1...numbits]

     Return Y
```

### 3.3 PDF: Pad-derivation function

This function takes a key and a nonce and returns a pseudorandom pad for use in tag generation. A pad of length 64 or 128 bits can be generated. Notice that when the block-cipher block-length is twice as long as the pad, nonces that differ only in their last bit are derived from the same block cipher encryption. This allows caching and sharing a single block cipher invocation for sequential nonces.

#### 3.3.1 PDF Algorithm

```
   Input:
     K, string with bitlength KEYLEN.
     Nonce, string of length in the range 1...BLOCKLEN bits.
     taglen, the integer 64 or 128.
   Output:
     Y, string of length taglen bits.

   Compute Y using the following algorithm.

      //
      // Extract and zero low bits of Nonce if needed.
      // If BLOCKLEN/taglen < 2, this step does nothing but set index=0
      //
      Let i be the greatest integer for which BLOCKLEN/taglen <= 2^i
      index = str2uint(Nonce) mod 2^i
      Nonce = Nonce xor uint2str(index, bitlength(Nonce))

      //
      // Make Nonce BLOCKLEN bits by appending zeros if needed
      //
      Nonce = Nonce || zeros(BLOCKLEN - bitlength(Nonce))

      //
      // Generate subkey, encipher and extract indexed substring
```

```
//
T = ENCIPHER(K, Nonce)
Y = T[index * taglen + 1 ... index * taglen + taglen ]

Return Y
```

## 4 VMAC tag generation

Tag generation for VMAC proceeds by using VHASH (defined in the next section) to hash the message, applying the PDF to the nonce and computing the xor of the resulting strings. The first bit of the nonce must be zero to ensure the KDF and PDF functions do not pass the same values to the block cipher. The length of the pad and hash can be either 64 or 128 bits.

### 4.1 VMAC Algorithm

```
Input:
  K, string of length KEYLEN bits.
  M, string of any length.
  Nonce, string of length 1 to BLOCKLEN bits  // first bit MUST be 0.
  taglen, the integer 64 or 128.
Output:
  Tag, string of length taglen bits.
```

Compute Tag using the following algorithm.

```
HashedMessage = VHASH(K, M, taglen)
Pad           = PDF(K, Nonce, taglen)
if taglen = 64 then
  Tag         = Pad +_64  HashedMessage
else
  Tag         = Pad +_128 HashedMessage
end if

Return Tag
```

### 4.2 VMAC-64 and VMAC-128

The preceding VMAC definition has a parameter "taglen" which specifies the length of tag generated by the algorithm. The following aliases define names that make tag length explicit in the name.

```
VMAC-64(K, M, Nonce) = VMAC(K, M, Nonce, 64)
```

```
VMAC-128(K, M, Nonce) = VMAC(K, M, Nonce, 128)
```

## 5 VHASH: Universal hash function

VHASH is a keyed hash function, which takes as input a string, and produces an 64- or 128-bit output. VHASH does its work in two or three stages, or layers, depending on whether an 64- or 128-bit output is requested. A message is first hashed by L1-HASH, its output is then hashed by L2-HASH, whose output is then hashed by L3-HASH if taglen is eight.

Please note that VHASH has certain combinatoric properties making it suitable for Wegman-Carter message authentication. VHASH is not a cryptographic hash function and is not a suitable general replacement for functions like SHA-1.

VHASH is presented here in a top-down manner. First VHASH is described, then each of its component hashes are presented.

### 5.1 VHASH Algorithm

```
Input:
  K, string of length KEYLEN bits.
  M, string of any length.
  taglen, the integer 64 or 128.
Output:
  Y, string of length taglen bits.
```

Compute Y using the following algorithm.

```
  A = L1-HASH(K, M, taglen)
  B = L2-HASH(K, A, taglen)
  if taglen = 64 then
     Y = L3-HASH(K, B)
  else
     Y = B
  end if

  Return Y
```

### 5.2 L1-HASH: First-layer hash

The first-layer hash breaks the message into blocks, each of length L1KEYLEN (defined as 128 bytes), and hashes each with a function called NH. Concatenating the results forms a string which is shorter

than the original.  One could customize VHASH by changing L1KEYLEN to any multiple of 128, achieving different performance characteristics, but the resulting algorithm would not be interoperable with the standard algorithm defined in this document.

### 5.2.1  L1-HASH Algorithm

```
Input:
  K, string of length KEYLEN bits.
  M, string of any length.
  taglen, the integer 64 or 128.
Output:
  Y, string of length (2 * taglen * ceil(bitlength(M)/L1KEYLEN))
     bits.

Compute Y using the following algorithm.

  //
  // Set subkey for L1-HASH
  //
  L1KEYLEN = 1024
  T = KDF(K, 128, L1KEYLEN+128)
  K_1 = T[1 ... L1KEYLEN]
  K_2 = T[129 ... L1KEYLEN + 128]   // Only used when taglen = 128

  //
  // Partition M into L1KEYLEN-bit sgements (last one may be shorter)
  //
  t = max(ceil(bitlength(M) / L1KEYLEN), 1)
  Let M_1, M_2, ..., M_t be strings so that M = M_1 || M_2 || ... ||
     M_t, and bitlength(M_i) = L1KEYLEN for all 0 < i < t.

  //
  // For each segment, except the last: endian-adjust, NH hash,
  // and use the results to build output Y.
  //
  Y = <empty string>
  for i = 1 to t-1 do
     ENDIAN-SWAP(M_i)
     Y = Y || NH(K_1, M_i)
     if taglen = 128 then
        Y = Y || NH(K_2, M_i)  // Hash twice for 128-bit outputs
     end if
  end for

  //
  // For the last block: pad to 128-bit multiple, endian-adjust,
```

```
    // NH hash and add bit-length.  Concatenate the result to Y.
    //
    Len = uint2str(bitlength(M_t), 64) || zeros(64)
    M_t = zeropad(M_t, 128)
    ENDIAN-SWAP(M_t)
    Y = Y || (NH(K_1, M_t) +_128 Len)
    if taglen = 128 then
       Y = Y || NH(K_2, M_t)
    end if

    Return Y
```

### 5.2.2 NH Algorithm

Because this routine is applied directly to every bit of input data, an optimized implementation of it yields great benefit.

```
  Input:
    K, string with length a multiple of 128 bits.
    M, string with length a multiple of 128 bits, but no longer than K.
  Output:
    Y, string of length 128 bits.

  Compute Y using the following algorithm.

    //
    // Partition M and K into 64-bit substrings
    //
    t = bitlength(M) / 64
    Let M_1, M_2, ..., M_t be 64-bit strings
      so that M = M_1 || M_2 || ... || M_t.
    Let K_1, K_2, ..., K_t be 64-bit strings
      so that K_1 || K_2 || ... || K_t  is a prefix of K.

    //
    // Perform NH hash on each.
    //
    Y = zeros(128)
    i = 1
    while (i < t) do
      Y = Y +_128 ((M_i +_64 K_i) *_128 (M_{i+1} +_64 K_{i+1}))
      i = i + 2
    end while
    Y = zeros(2) || Y[3...128]  // Zero first two bits

    Return Y
```

### 5.3 L2-HASH: Second-layer hash

The second-layer rehashes the L1-HASH output using a polynomial hash.

#### 5.3.1 L2-HASH Algorithm

```
Input:
  K, string of length KEYLEN bits.
  M, string with length a multiple of 128 bits.
Output:
  Y, string of length 128 bits.

Compute y using the following algorithm.

  //
  //  Create subkey
  //
  T = KDF(K, 192, 128)
  k = str2uint(zeros(2) || T[ 3...32] || zeros(2) || T[35... 64] ||
               zeros(2) || T[67...96] || zeros(2) || T[99...128])

  //
  // Partition M into 128-bit substrings
  //
  n = bitlength(M) / 128
  Let M_1, M_2, ..., M_n be strings of length 128 bits
    so that M = M_1 || M_2 || ... || M_n

  //
  // Polynomial hash M
  //
  y = 1
  for i = 1 to n do
    m_i = str2uint(M_i)
    y = (k * y + m_i) mod prime(127)
  end for
  y = (k * y) mod prime(127)
  Y = uint2str(y, 128)

  Return Y
```

### 5.4 L3-HASH: Third-layer hash

The output from L2-HASH is 128 bits long. This final hash function hashes the 128-bit string to a fixed length of 64 bits. Note that the "do" loop during subkey generation has less than 1/2^58

probability of requiring more than one iteration.

### 5.4.1 L3-HASH Algorithm

```
Input:
  K, string of length KEYLEN bits.
  M, string of length 128 bits.
Output:
  Y, string of length 64 bits.

Compute Y using the following algorithm.

  i = 0
  repeat
    T    = KDF(K, 224+i, 128)
    k_1  = str2uint(T[ 4... 64])
    k_2  = str2uint(T[68...128])
    i    = i + 1
  until (k_1 < prime(61)) and (k_2 < prime(61))

  m_1  = str2uint(M[ 5... 64])
  m_2  = str2uint(M[69...128])
  y = (m_1 * k_1 + m_2 * k_2) mod prime(61)

  Y = uint2str(y, 64)

  Return Y
```

## 6 Security considerations

Here we describe some security considerations important for the proper understanding and use of VMAC.

### 6.1 Resistance to cryptanalysis

The strength of VMAC depends on the strength of its underlying cryptographic functions: the key-derivation function (KDF) and the pad-derivation function (PDF). In this specification both operations are implemented using a block cipher, by default the Advanced Encryption Standard (AES). However, the core of the VMAC design, the VHASH function, does not depend on cryptographic assumptions: its strength is specified by a purely mathematical property stated in terms of collision probability, and this property is proven unconditionally [5]. This means the strength of VHASH is guaranteed regardless of advances in cryptanalysis and that an adversarial

attack on VMAC that forges with probability significantly exceeding the established collision probability of VHASH will give rise to an attack of comparable complexity which breaks the block cipher, in the sense of distinguishing the block cipher from a family of random permutations. This design approach essentially obviates the need for cryptanalysis on VMAC: cryptanalytic efforts might as well focus on the block cipher.

### 6.2 Tag lengths and forging probability

A MAC algorithm is used to authenticate messages between two parties that share a secret MAC key K. An authentication tag is computed for a message using K and, in some MAC algorithms such as VMAC, a nonce. Messages transmitted between parties are accompanied by their tag and, possibly, nonce. Breaking the MAC means that the attacker is able to generate, on its own, with no knowledge of the key K, a new message M (ie, one not previously transmitted between the legitimate parties) and to compute on M a correct authentication tag under the key K. This is called a forgery. Note that if the authentication tag is specified to be of length t then the attacker can trivially break the MAC with probability 1/2^t. For this the attacker can just generate any message of its choice and try a random tag; obviously, the tag is correct with probability 1/2^t. By repeated guesses the attacker can increase linearly its probability of success.

In the case of VMAC-64, for example, the above guessing-attack strategy is close to optimal. An adversary can correctly guess a 64-bit VMAC tag with probability 1/2^64 by simply guessing a random value. The theory of Wegman-Carter MACs and results of [5] show that no attack strategy can produce a correct tag with probability better than about 1/2^59 if VMAC were to use a random function in its work rather than AES. Another result shows that so long as AES is secure as a pseudorandom permutation, it can be used instead of a random function without significantly increasing the 1/2^59 forging probability, assuming that no more than 2^64 messages are authenticated with the same key [2]. Similarly for VMAC-128, the per-message forgery probability, when using a random function rather than AES to instantiate VMAC and limiting messages to j bits, is no more than j/2^117.

AES has undergone extensive study and is assumed to be very secure as a pseudorandom permutation. If we assume that no attacker with feasible computational power can distinguish randomly keyed AES from a randomly chosen permutation with probability delta (more precisely, delta is a function of the computational resources of the attacker and of its ability to sample the function), then we obtain that no such attacker can forge j-bit messages in VMAC with probability

greater than about 1/2^59 or j/2^117, plus delta. Over N forgery attempts, forgery occurs with probability no more than N/^59 or N/2^117, plus delta. The value delta could possibly be greater than 1/2^59 or 1/2^88, in which case the probability of VMAC forging is dominated by a term representing the security of AES.

With VMAC, off-line computation aimed at exceeding the forging probability is hopeless as long as the underlying cipher is not broken. An attacker attempting to forge VMAC tags will need to interact with the entity that verifies message tags and try a large number of forgeries before one is likely to succeed. The system architecture will determine the extent to which this is possible. In a well-architected system there should not be any high-bandwidth capability for presenting forged MACs and determining if they are valid. In particular, the number of authentication failures at the verifying party should be limited. If a large number of such attempts are detected the session key in use should be dropped and the event reported.

Let us reemphasize: a forging probability of 1/2^59 does not mean that there is an attack that runs in 2^59 time; to the contrary, as long as the block cipher in use is not broken there is no such attack for VMAC. Instead, a 1/2^59 forging probability means that if an attacker could have N forgery attempts, then the attacker would have no more than N/2^59 probability of getting one or more of them right.

It should be pointed out that once an attempted forgery is successful, it is possible, in principle, that subsequent messages under this key may be more easily forged. This is important to understand in gauging the severity of a successful forgery, even though no such attack on VMAC is known to date. Due to the short-lived nature of most authentication sessions, 64-bit tags seem appropriate for many security architectures and commercial applications. If, however, one wants a more conservative option, at a cost of about double the computation, VMAC's 128-bit tags may be more appropriate.

## 6.3 Nonce considerations

VMAC requires a nonce with length upto BLOCKLEN bits. (For technical reasons, the first bit of every nonce must be zero.) All nonces in an authentication session must be equal in length. For secure operation, no nonce value should be repeated within the life of a single VMAC session-key. There is no guarantee of message authenticity when a nonce is repeated, and so messages accompanied by a repeated nonce should be considered not authenticated.

To authenticate messages over a duplex channel (where two parties send messages to each other), a different key could be used for each direction. If the same key is used in both directions, then it is crucial that all nonces be distinct. For example, one party can use even nonces while the other party uses odd ones. The receiving party must verify that the sender is using a nonce of the correct form.

This specification does not indicate how nonce values are created, updated, or communicated between the entity producing a tag and the entity verifying a tag. The following are possibilities:

1. The nonce is a 64-bit unsigned number, Counter, which is initialized to zero, which is incremented by one following the generation of each authentication tag, and which is always communicated along with the message and the authentication tag. An error occurs at the sender if there is an attempt to authenticate more than 2^63 messages within a session.

2. The nonce is a BLOCKLEN-bit unsigned number, Counter, which is initialized to zero and which is incremented by one following the generation of each authentication tag. The Counter is not explicitly communicated between the sender and receiver. Instead, the two are assumed to communicate over a reliable transport, and each maintains its own counter so as to keep track of what the current nonce value is.

3. The nonce is a BLOCKLEN-bit random value with first bit zero, communicated along with the messgae and tag. Because repetitions in a random n-bit value are expected at around 2^(n/2) trials, the number of messages to be communicated in a session using n-bit random nonces should not be allowed to approach 2^(n/2).

We emphasize that the value of the nonce need not be kept secret. When VMAC is used within a higher-level protocol there may already be a field, such as a sequence number, which can be co-opted so as to specify the nonce needed by VMAC.

### 6.4 Replay attacks

A replay attack entails the attacker repeating a message, nonce, and authentication tag. In many applications, replay attacks may be quite damaging and must be prevented. In VMAC, this would normally be done at the receiver by having the receiver check that no nonce value is used twice. On a reliable connection, when the nonce is a counter, this is trivial. On an unreliable connection, when the nonce is a counter, one would normally cache some window of recent nonces. Out-of-order message delivery in excess of what the window

allows will result in rejecting otherwise valid authentication tags. We emphasize that it is up to the receiver to determine when a given (message, nonce, tag) triple will be deemed authentic. Certainly the tag should be valid for the message and nonce, as determined by VMAC, but the message may still be deemed inauthentic because the nonce is detected to be a replay.

## 7 IANA Considerations

This document has no actions for IANA.

## Appendix - Test vectors

Following are some sample VMAC outputs over a collection of input values, using AES with 128-bit keys. Let key K and nonce N be defined by the following ASCII strings.

```
K  = "abcdefghijklmnop"              // A 128-bit VMAC key
N  = "bcdefghi"                      // A 64-bit nonce
```

The tags generated by VMAC using key K and nonce N are:

| Message | 64-bit Tag | 128-bit Tag |
|---|---|---|
| <empty> | 4EDE4AE94EDD87E1 | E87569084EFF3E1CCA1500C5A6A89CE6 |
| 'abc' * 1 | 4157A6D46E3EC1A1 | E5B10669E5B61668A11E3351CC1A7211 |
| 'abc' * 16 | 4D3C8A9C2A09E2DE | 12A64330F81D8B6407CE90667303FEE2 |
| 'abc' * 100 | 4FD5EC2FCFE31FBE | 10A63F27D4B292723739B4BB6F17A4C1 |
| 'abc' * 10^6 | 4E13F57841D33D58 | 22C65CC2CFE9BED72E485CA6EB8A48BE |

The first column lists a small sample of messages which are strings of repeated ASCII 'abc' strings. The remaining columns give in hexadecimal the tags generated when VMAC is called with the corresponding message, nonce N and key K.

## References

### Normative References

[1] FIPS-197, "Advanced Encryption Standard (AES)", National Institute of Standards and Technology, 2001.

### Informative References

[2] D. Bernstein, "Stronger security bounds for permutations",

unpublished manuscript, 2005. This work refines "Stronger security bounds for Wegman-Carter-Shoup authenticators", Advances in Cryptology - EUROCRYPT 2005, LNCS vol. 3494, pp. 164-180, Springer-Verlag, 2005.

[3] J. Black, S. Halevi, A. Hevia, H. Krawczyk, T. Krovetz, and P. Rogaway, "UMAC: Message authentication code using universal hashing", RFC 4418, IETF, 2006.

[4] L. Carter and M. Wegman, "Universal classes of hash functions", Journal of Computer and System Sciences, 18 (1979), pp. 143-154.

[5] T. Krovetz, "Message auhentication on 64-bit architectures", Selected Areas in Cryptography - SAC 2006, Springer-Verlag, 2006.

[6] M. Wegman and L. Carter, "New hash functions and their use in authentication and set equality", Journal of Computer and System Sciences, 22 (1981), pp. 265-279.

## Author contact information

### Author's Address

Ted Krovetz
Department of Computer Science
California State University
Sacramento CA 95819
USA

EMail: tdk@acm.org

## Full Copyright Statement

Copyright (C) The Internet Society (2006).

This document is subject to the rights, licenses and restrictions contained in BCP 78, and except as set forth therein, the authors retain all their rights.

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE

INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

## Intellectual Property

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the ISOC's procedures with respect to rights in ISOC Documents can be found in BCP 78 and BCP 79.

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at http://www.ietf.org/ipr.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.

## Acknowledgments

This document borrows much text from RFC 4418 [3]. That document describes another message authentication scheme, UMAC, and was co-written by John Black, Shai Halevi, Alejandro Hevia, Hugo Krawczyk, Ted Krovetz and Phillip Rogaway. Funding for the RFC Editor function is currently provided by the Internet Society.