Network Working Group K. Burgin
Internet Draft National Security Agency
Intended Status: Informational M. Peck
Expires: December 30, 2013 The MITRE Corporation
June 28, 2013

AES Encryption with HMAC-SHA2 for Kerberos 5
draft-ietf-kitten-aes-cts-hmac-sha2-01

Abstract

This document specifies two encryption types and two corresponding checksum types for Kerberos 5. The new types use AES in CTS mode (CBC mode with ciphertext stealing) for confidentiality and HMAC with a SHA-2 hash for integrity.

Status of this Memo

Copyright and License Notice

Table of Contents

1. Introduction

This document defines two encryption types and two corresponding checksum types for Kerberos 5 using AES with 128-bit or 256-bit keys. To avoid ciphertext expansion, we use the CBC-CS3 variant to CBC mode defined in [SP800-38A+] (this mode is also referred to as CTS). The new types conform to the framework specified in [RFC3961], but do not use the simplified profile.

Note that [SP800-38A+] requires the plaintext length to be greater than the block size, so the encryption types have two cases.

The encryption and checksum types defined in this document are intended to support NSA's Suite B Profile for Kerberos [suiteb-kerberos] which requires the use of SHA-256 or SHA-384 as the hash algorithm. Differences between the encryption and checksum types defined in this document and existing Kerberos encryption and checksum types are:

* The pseudorandom function used by PBKDF2 is HMAC-SHA-256 or HMAC-SHA-384.

* A key derivation function from [SP800-108] which uses the SHA-256 or SHA-384 hash algorithm is used to produce keys for encryption, integrity protection, and checksum operations.

* The IV used during content encryption is sent as part of the ciphertext, instead of using a confounder. This saves one encryption and decryption operation per message.

* The HMAC is calculated over the AES output, instead of being calculated over the plaintext. This allows the message receiver to verify the integrity of the message before decrypting the message.

* The HMAC algorithm uses the SHA-256 or SHA-384 hash algorithm for integrity protection and checksum operations.

2. Protocol Key Representation

The AES key space is dense, so we can use random or pseudorandom octet strings directly as keys. The byte representation for the key is described in [FIPS197], where the first bit of the bit string is the high bit of the first byte of the byte string (octet string).

3. Key Generation from Pass Phrases

The pseudorandom function used by PBKDF2 will be the SHA-256 or SHA-

384 HMAC of the passphrase and salt. If the enctype is "aes128-cts-hmac-sha256-128", then HMAC-SHA-256 is used as the PRF. If the enctype is "aes256-cts-hmac-sha384-192", then HMAC-SHA-384 is used as the PRF.

The final key derivation step uses the algorithm KDF-HMAC-SHA2 defined below in Section 4.

If no string-to-key parameters are specified, the default number of iterations is raised to 32,768.

To ensure that different long-term keys are used with different enctypes, we prepend the enctype name to the salt string, separated by a null byte. The enctype name is "aes128-cts-hmac-sha256-128" or "aes256-cts-hmac-sha384-192" (without the quotes). The user's long-term key is derived as follows

```
  saltp = enctype-name | 0x00 | salt
  tkey = random-to-key(PBKDF2(passphrase, saltp,
                         iter_count, keylength))
  key = KDF-HMAC-SHA2(tkey, "kerberos") where "kerberos" is the
        byte string {0x6b65726265726f73}.
```

where the pseudorandom function used by PBKDF2 is HMAC-SHA-256 when the enctype is "aes128-cts-hmac-sha256-128" and HMAC-SHA-384 when the enctype is "aes256-cts-hmac-sha384-192", the value for keylength is the AES key length, and the algorithm KDF-HMAC-SHA2 is defined in Section 4.

4. Key Derivation Function

We use a key derivation function from Section 5.1 of [SP800-108] which uses the HMAC algorithm as the PRF. The counter i is expressed as four octets in big-endian order. The length of the output key in bits (denoted as k) is also represented as four octets in big-endian order. The "Label" input to the KDF is the usage constant supplied to the key derivation function, and the "Context" input is null. Each application of the KDF only requires a single iteration of the PRF, so n = 1 in the notation of [SP800-108].

In the following summary, | indicates concatenation. The random-to-key function is the identity function, as defined in Section 3. The k-truncate function is defined in [RFC3961], Section 5.1.

When the encryption type is aes128-cts-hmac-sha256-128, the output key length k is 128 bits for all applications of KDF-HMAC-SHA2(key, constant) which is computed as follows:

```
K1 = HMAC-SHA-256(key, 00 00 00 01 | constant | 0x00 | 00 00 00 80)
KDF-HMAC-SHA2(key, constant) = random-to-key(k-truncate(K1))
```

When the encryption type is aes256-cts-hmac-sha384-192, the output key length k is 256 bits when computing the base-key and Ke, and the output key length k is 192 bits when deriving Kc and Ki. KDF-HMAC-SHA2(key, constant) is computed as follows:

```
If deriving Kc or Ki (the constant ends with 0x99 or 0x55):
k = 192
K1 = HMAC-SHA-384(key, 00 00 00 01 | constant | 0x00 | 00 00 00 C0)
KDF-HMAC-SHA2(key, constant) = random-to-key(k-truncate(K1))

Otherwise (if deriving Ke or deriving the base-key from a
           passphrase as described in Section 3):
k = 256
K1 = HMAC-SHA-384(key, 00 00 00 01 | constant | 0x00 | 00 00 01 00)
KDF-HMAC-SHA2(key, constant) = random-to-key(k-truncate(K1))
```

The constants used for key derivation are the same as those used in the simplified profile.

5. Kerberos Algorithm Protocol Parameters

In cases where the plaintext length is greater than the block size:

Each encryption will use a 16-octet nonce generated at random by the message originator. The initialization vector (IV) used by AES is obtained by xoring the random nonce with the cipherstate.

The ciphertext is the concatenation of the random nonce, the output of AES in CBC-CS3 mode, and the HMAC of the nonce concatenated with the AES output. The HMAC is computed using either SHA-256 or SHA-384. The output of SHA-256 is truncated to 128 bits and the output of SHA-384 is truncated to 192 bits. Sample test vectors are given in Appendix A.

Decryption is performed by removing the HMAC, verifying the HMAC against the remainder, and then decrypting the remainder if the HMAC is correct.

In cases where the plaintext length is less than or equal to the block size, a different algorithm is specified.

Each encryption will use a 16-octet nonce generated at random by the message originator. The initialization vector (IV) used by AES is obtained by xoring the random nonce with the cipherstate.

The plaintext is padded with zeros so the length of the result is one block length (no zeros are added if the plaintext length equals the block length). The padded plaintext is xored with the IV, then encrypted using AES in ECB mode. The output of AES is split into two parts, so that the length of the first part equals the length of the unpadded plaintext. The nonce is also split into two parts, so that the length of the first part equals the length of the unpadded plaintext.

The ciphertext is the concatenation of the first part of the random nonce, the second part of the AES output followed by the first part of the AES output, and the HMAC of the concatenation of the first part of the random nonce, the second part of the AES output followed by the first part of the AES output. The HMAC is computed using either SHA-256 or SHA-384. The output of SHA-256 is truncated to 128 bits and the output of SHA-384 is truncated to 192 bits. Sample test vectors are given in Appendix A.

Decryption is performed by first removing the HMAC, and verifying the HMAC against the remainder. If the HMAC is correct, separate the remainder into N' and C' by taking the first 16 bytes as N', and the following bytes as C'. Split N' into two parts, so that the length of the first part equals the length of C'. Decrypt the concatenation of C' with the second part of N' using ECB mode to get a value P' whose length is one block length. The nonce is recovered by taking the concatenation of the first part of N' with the second part of P' xored with the cipherState (where again, the length of the first part equals the length of C'). The IV is recovered as the nonce xored with cipherState, and the plaintext is recovered as the first part of P' xored with the IV.

The following parameters apply to the encryption types aes128-cts-hmac-sha256-128 and aes256-cts-hmac-sha384-192.

protocol key format: as defined in Section 2.

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

required checksum mechanism: as defined in Section 6.

key-generation seed length: key size (128 or 256 bits).

string-to-key function: as defined in Section 3.

default string-to-key parameters: 00 00 80 00.

random-to-key function: identity function.

key-derivation function: KDF-HMAC-SHA2 as defined in Section 4. The key usage number is expressed as four octets in big-endian order.

```
Kc = KDF-HMAC-SHA2(base-key, usage | 0x99)
Ke = KDF-HMAC-SHA2(base-key, usage | 0xAA)
Ki = KDF-HMAC-SHA2(base-key, usage | 0x55)
```

cipherState: a 128-bit random nonce.

initial cipherState: all bits zero.

encryption function: as follows. When the plaintext length is greater than the block size, CTS mode is used. When the plaintext is less than or equal to the block size, ECB mode is used.

```
h = size of truncated HMAC
E() = encryption function
D() = decryption function
c = block size of the encryption algorithm
L(x) = length of x
< = less-than operator; true == 1, false == 0
zeroblock = one block (length c) of zeros
o[start:len] = sub-string operation returning the substring of
               length len of string o starting at byte start
               (zero-based)

encryption function:
   N = random nonce of length 128 bits
   IV = N XOR cipherState
   if (L(P) > c)
       PC = 0
       P' = P
       C = E(Ke, P', IV)
           // using CBC-CS3-Encrypt defined
           // in [SP800-38A+]
       N' = N
       C' = C
   else
       PC = c - L(P)
       P' = P | zeroblock[0:PC]
       C = E(Ke, P' XOR IV)
           // using ECB mode
       N' = N[0:c - PC] | C[c - PC:PC]
       C' = C[0:c - PC]
   H = HMAC(Ki, N' | C')
   ciphertext =  N' | C' | H[1..h]
   cipherState = N
```

```
decryption function:
   (N', C', H) = ciphertext
   if (H != HMAC(Ki, N' | C')[1..h])
       stop, report error

   if (L(C') > c)
       // Not short-plaintext
       IV = N' XOR cipherState
       P = D(Ke, C', IV)
           // using CBC-CS3-Decrypt defined
           // in [SP800-38A+]
       cipherState = N'
       stop, output P, success
   else
   // Short plaintext
   PC = c - L(C')
   C = C' | N'[c - PC:PC]
   P' = D(Ke, C)
        // using ECB mode

   // P' here == (P | zeroblock[0:PC]) XOR IV
   // so IV[c - PC:PC] == P'[c - PC:PC]
   // In the non-short-pt case we'd recover
   // IV as N XOR cipherState, but here we only know
   // a head of N and tail of IV.

   N = N'[0:c -PC] | (P' XOR cipherState)[c - PC:PC]
   IV = N XOR cipherState
   P = (P' XOR IV)[0:PC]
   cipherState = N
   stop, output P, success

pseudo-random function:
   Kp  = KDF-HMAC-SHA2(protocol-key, "prf")
   PRF = HMAC(Kp, octet-string)
```

6. Checksum Parameters

The following parameters apply to the checksum types hmac-sha256-128-aes128 and hmac-sha384-192-aes256, which are the associated checksums for aes128-cts-hmac-sha256-128 and aes256-cts-hmac-sha384-192, respectively.

associated cryptosystem: AES-128-CTS or AES-256-CTS as appropriate

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

verify_mic: get_mic and compare

7. IANA Considerations

IANA is requested to assign:

Encryption type numbers for aes128-cts-hmac-sha256-128 and aes256-cts-hmac-sha384-192 in the Kerberos Encryption Type Numbers registry.

Etype	encryption type	Reference
TBD1	aes128-cts-hmac-sha256-128	[this document]
TBD2	aes256-cts-hmac-sha384-192	[this document]

Checksum type numbers for hmac-sha256-128-aes128 and hmac-sha384-192-aes256 in the Kerberos Checksum Type Numbers registry.

Sumtype	Checksum type	Size	Reference
TBD3	hmac-sha256-128-aes128	16	[this document]
TBD4	hmac-sha384-192-aes256	24	[this document]

8. Security Considerations

This specification requires implementations to generate random values. The use of inadequate pseudo-random number generators (PRNGs) can result in little or no security. The generation of quality random numbers is difficult. NIST Special Publication 800-90 [SP800-90] and [RFC4086] offer random number generation guidance.

This document specifies a mechanism for generating keys from pass phrases or passwords. The salt and iteration count resist brute force and dictionary attacks, however, it is still important to choose or generate strong passphrases.

8.1. Random Values in Salt Strings

NIST guidance in Section 5.1 of [SP800-132] requires the salt used as input to the PBKDF to contain at least 128 bits of random. Some known issues with including random values in Kerberos encryption type salt strings are:

* Cross-realm TGTs are currently managed by entering the same password at two KDCs to get the same keys. If each KDC uses a random salt, they won't have the same keys.

* The string-to-key function as defined in [RFC3961] requires the salt to be valid UTF-8 strings. Not every 128-bit random string will be valid UTF-8.

* Current implementations of password history checking will not work.

* ktutil's add_entry command assumes the default salt.

9. References

9.1. Normative References

[RFC3961] Raeburn, K., "Encryption and Checksum Specifications for Kerberos 5", RFC 3961, February 2005.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.

[FIPS197] National Institute of Standards and Technology, "Advanced Encryption Standard (AES)", FIPS PUB 197, November 2001.

9.2. Informative References

[SP800-38A+] National Institute of Standards and Technology, "Recommendation for Block Cipher Modes of Operation: Three Variants of Ciphertext Stealing for CBC Mode", Addendum to NIST Special Publication 800-38A, October 2010.

[SP800-90] National Institute of Standards and Technology, Recommendation for Random Number Generation Using Deterministic Random Bit Generators (Revised), NIST Special Publication 800-90, March 2007.

[SP800-108] National Institute of Standards and Technology, "Recommendation for Key Derivation Using Pseudorandom Functions", NIST Special Publication 800-108, October 2009.

[SP800-132] National Institute of Standards and Technology, "Recommendation for Password-Based Key Derivation, Part 1: Storage Applications", NIST Special Publication 800-132, June 2010.

[suiteb-kerberos] Burgin, K. and K. Igoe, "Suite B Profile for Kerberos 5", internet-draft draft-burgin-kerberos-suiteb-01, 2012.

Appendix A. Test Vectors

```
Sample results for string-to-key conversion:
--------------------------------------------

Iteration count = 32768
Pass phrase = "password"
Saltp for creating 128-bit master key:
   61 65 73 31 32 38 2D 63 74 73 2D 68 6D 61 63 2D
   73 68 61 32 35 36 2D 31 32 38 00 10 DF 9D D7 83
   E5 BC 8A CE A1 73 0E 74 35 5F 61 41 54 48 45 4E
   41 2E 4D 49 54 2E 45 44 55 72 61 65 62 75 72 6E
(The saltp is "aes128-cts-hmac-sha256-128" | 0x00 |
 random 16 byte valid UTF-8 sequence | "ATHENA.MIT.EDUraeburn")
128-bit master key:
   3C 44 03 85 28 06 BF 5C EE E6 36 48 6C 29 2F D6

Saltp for creating 256-bit master key:
   61 65 73 32 35 36 2D 63 74 73 2D 68 6D 61 63 2D
   73 68 61 33 38 34 2D 31 39 32 00 10 DF 9D D7 83
   E5 BC 8A CE A1 73 0E 74 35 5F 61 41 54 48 45 4E
   41 2E 4D 49 54 2E 45 44 55 72 61 65 62 75 72 6E
(The saltp is "aes256-cts-hmac-sha384-192" | 0x00 |
 random 16 byte valid UTF-8 sequence | "ATHENA.MIT.EDUraeburn")
256-bit master key:
   53 96 0C AF 44 D5 57 4D FF 4D 44 37 38 75 22 B0
   7F 5B 02 5C 5E 65 BF EF 29 C2 B4 28 98 3B 37 08

Sample results for key derivation:
----------------------------------

enctype aes128-cts-hmac-sha256-128:
128-bit master key:
   37 05 D9 60 80 C1 77 28 A0 E8 00 EA B6 E0 D2 3C
Kc value for key usage 2 (constant = 0x0000000299):
   B3 1A 01 8A 48 F5 47 76 F4 03 E9 A3 96 32 5D C3
Ke value for key usage 2 (constant = 0x00000002AA):
   9B 19 7D D1 E8 C5 60 9D 6E 67 C3 E3 7C 62 C7 2E
Ki value for key usage 2 (constant = 0x0000000255):
   9F DA 0E 56 AB 2D 85 E1 56 9A 68 86 96 C2 6A 6C

enctype aes256-cts-hmac-sha384-192:
256-bit master key:
   6D 40 4D 37 FA F7 9F 9D F0 D3 35 68 D3 20 66 98
   00 EB 48 36 47 2E A8 A0 26 D1 6B 71 82 46 0C 52
Kc value for key usage 2 (constant = 0x0000000299):
   EF 57 18 BE 86 CC 84 96 3D 8B BB 50 31 E9 F5 C4
   BA 41 F2 8F AF 69 E7 3D
```

```
   Ke value for key usage 2 (constant = 0x00000002AA):
      56 AB 22 BE E6 3D 82 D7 BC 52 27 F6 77 3F 8E A7
      A5 EB 1C 82 51 60 C3 83 12 98 0C 44 2E 5C 7E 49
   Ki value for key usage 2 (constant = 0x0000000255):
      69 B1 65 14 E3 CD 8E 56 B8 20 10 D5 C7 30 12 B6
      22 C4 D0 0F FC 23 ED 1F

   Sample encryptions (using the default cipher state):
   ----------------------------------------------------

   128-bit AES key:
      2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
   128-bit HMAC key:
      67 C3 31 A4 D7 AB 52 EF 3A A9 73 E0 39 AD D3 32
   Nonce:
      7E 58 95 EA F2 67 24 35 BA D8 17 F5 45 A3 71 48
   Plaintext: (length less than block size)
      49 6E 63 6F 6E 63 65 69 76 61 62 6C 65
   AES Output:
      1C 17 3E AD FC 67 C8 BC B3 A5 93 02 98 CB FC 60
   HMAC Output (truncated):
      35 E8 32 B2 EB F4 6A 46 C2 E6 50 D2 50 AB 84 43
   Ciphertext: (Nonce* | AES Output** | Truncated HMAC Output)
      7E 58 95 EA F2 67 24 35 BA D8 17 F5 45 CB FC 60
      1C 17 3E AD FC 67 C8 BC B3 A5 93 02 98 35 E8 32
      B2 EB F4 6A 46 C2 E6 50 D2 50 AB 84 43

   *   Only the first 13 bytes of Nonce are sent.
   **  The AES Output is split and rearranged as described in Section 5
       since the plaintext length is less than the block size.

   128-bit AES key:
      2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
   128-bit HMAC key:
      67 C3 31 A4 D7 AB 52 EF 3A A9 73 E0 39 AD D3 32
   Nonce:
      7E 58 95 EA F2 67 24 35 BA D8 17 F5 45 A3 71 48
   Plaintext: (length equals block size)
      67 61 73 74 72 6F 69 6E 74 65 73 74 69 6E 61 6C
   AES Output:
      F6 71 0B 75 0C 60 65 E8 2E BF F8 9D DC E0 C9 B9
   HMAC Output (truncated):
      7B 2C D9 70 E6 DF 18 F5 E0 3D 8B 8E 40 02 F4 C0
   Ciphertext: (Nonce | AES Output | Truncated HMAC Output)
      7E 58 95 EA F2 67 24 35 BA D8 17 F5 45 A3 71 48
      F6 71 0B 75 0C 60 65 E8 2E BF F8 9D DC E0 C9 B9
      7B 2C D9 70 E6 DF 18 F5 E0 3D 8B 8E 40 02 F4 C0
```

```
256-bit AES key:
   60 3D EB 10 15 CA 71 BE 2B 73 AE F0 85 7D 77 81
   1F 35 2C 07 3B 61 08 D7 2D 98 10 A3 09 14 DF F4
192-bit HMAC key:
   37 16 14 EB 62 24 E1 F0 C4 72 6E E6 BE A7 A3 D2
   F4 62 C6 AC 66 42 A6 AC
Nonce:
   7E 58 95 EA F2 67 24 35 BA D8 17 F5 45 A3 71 48
Plaintext: (length less than block size)
   49 6E 63 6F 6E 63 65 69 76 61 62 6C 65
AES Output:
   BD AE EC 5C F9 C9 B6 3C 9D DB A2 B7 9D 5C 6C 0B
HMAC Output (truncated):
   65 D4 C7 07 8E 14 65 8B C9 B3 C4 EA F5 F7 C2 6F
   ED 36 AC 7A CD 59 19 2B
Ciphertext: (Nonce* | AES Output* | Truncated HMAC Output)
   7E 58 95 EA F2 67 24 35 BA D8 17 F5 45 5C 6C 0B
   BD AE EC 5C F9 C9 B6 3C 9D DB A2 B7 9D 65 D4 C7
   07 8E 14 65 8B C9 B3 C4 EA F5 F7 C2 6F ED 36 AC
   7A CD 59 19 2B

*   Only the first 13 bytes of Nonce are sent.
**  The AES Output is split and rearranged as described in Section 5
    since the plaintext length is less than the block size.

256-bit AES key:
   60 3D EB 10 15 CA 71 BE 2B 73 AE F0 85 7D 77 81
   1F 35 2C 07 3B 61 08 D7 2D 98 10 A3 09 14 DF F4
192-bit HMAC key:
   37 16 14 EB 62 24 E1 F0 C4 72 6E E6 BE A7 A3 D2
   F4 62 C6 AC 66 42 A6 AC
Nonce:
   7E 58 95 EA F2 67 24 35 BA D8 17 F5 45 A3 71 48
Plaintext: (length equals block size)
   67 61 73 74 72 6F 69 6E 74 65 73 74 69 6E 61 6C
AES Output:
   5D E5 49 BE D6 50 23 18 78 8F 14 D2 E1 17 E0 5A
HMAC Output (truncated):
   2C EA DF D5 B0 60 38 DE A9 22 29 2D 7C 56 50 10
   C5 D6 D2 8D F6 21 E9 7A
Ciphertext: (Nonce | AES Output | Truncated HMAC Output)
   7E 58 95 EA F2 67 24 35 BA D8 17 F5 45 A3 71 48
   5D E5 49 BE D6 50 23 18 78 8F 14 D2 E1 17 E0 5A
   2C EA DF D5 B0 60 38 DE A9 22 29 2D 7C 56 50 10
   C5 D6 D2 8D F6 21 E9 7A

128-bit AES key:
   9B 19 7D D1 E8 C5 60 9D 6E 67 C3 E3 7C 62 C7 2E
```

```
128-bit HMAC key:
   9F DA 0E 56 AB 2D 85 E1 56 9A 68 86 96 C2 6A 6C
Nonce:
   8D 32 50 F6 36 AB 81 02 BE 6F AB 1E 57 D8 F8 17
Plaintext: (length greater than the block size)
   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
   10 11 12 13 14
AES Output:
   13 64 FB 39 DC C0 E3 D9 83 A7 DB 5B 4B 9F FB CA
   42 F6 65 88 29
HMAC Output (truncated):
   F2 1F C8 95 75 AE 93 C7 57 18 AB 3C 7C FB 28 E1
Ciphertext: (Nonce | AES Output | HMAC Output)
   8D 32 50 F6 36 AB 81 02 BE 6F AB 1E 57 D8 F8 17
   13 64 FB 39 DC C0 E3 D9 83 A7 DB 5B 4B 9F FB CA
   42 F6 65 88 29 F2 1F C8 95 75 AE 93 C7 57 18 AB
   3C 7C FB 28 E1

256-bit AES key:
   56 AB 22 BE E6 3D 82 D7 BC 52 27 F6 77 3F 8E A7
   A5 EB 1C 82 51 60 C3 83 12 98 0C 44 2E 5C 7E 49
192-bit HMAC key:
   69 B1 65 14 E3 CD 8E 56 B8 20 10 D5 C7 30 12 B6
   22 C4 D0 0F FC 23 ED 1F
Nonce:
   8D 32 50 F6 36 AB 81 02 BE 6F AB 1E 57 D8 F8 17
Plaintext: (length greater than the block size)
   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
   10 11 12 13 14
AES Output:
   50 CB FF DC DF 38 69 D7 0B EA FF C3 2C 47 0B C6
   5B 72 C3 37 2D
HMAC Output (truncated):
   6E D7 B3 47 E9 0B BD 8F 31 F5 79 58 F9 69 50 BA
   A1 41 64 6E 65 6C F6 7C
Ciphertext: (Nonce | AES Output | HMAC Output)
   8D 32 50 F6 36 AB 81 02 BE 6F AB 1E 57 D8 F8 17
   50 CB FF DC DF 38 69 D7 0B EA FF C3 2C 47 0B C6
   5B 72 C3 37 2D 6E D7 B3 47 E9 0B BD 8F 31 F5 79
   58 F9 69 50 BA A1 41 64 6E 65 6C F6 7C

Sample checksums:
-----------------

Checksum type: hmac-sha256-128-aes128
128-bit master key:
   37 05 D9 60 80 C1 77 28 A0 E8 00 EA B6 E0 D2 3C
128-bit HMAC key (Kc, key usage 2):
```

```
      B3 1A 01 8A 48 F5 47 76 F4 03 E9 A3 96 32 5D C3
   Plaintext:
      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
      10 11 12 13 14
   Checksum:
      D7 83 67 18 66 43 D6 7B 41 1C BA 91 39 FC 1D EE

   Checksum type: hmac-sha384-192-aes256
   256-bit master key:
      6D 40 4D 37 FA F7 9F 9D F0 D3 35 68 D3 20 66 98
      00 EB 48 36 47 2E A8 A0 26 D1 6B 71 82 46 0C 52
   192-bit HMAC key (Kc, key usage 2):
      EF 57 18 BE 86 CC 84 96 3D 8B BB 50 31 E9 F5 C4
      BA 41 F2 8F AF 69 E7 3D
   Plaintext:
      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
      10 11 12 13 14
   Checksum:
      45 EE 79 15 67 EE FC A3 7F 4A C1 E0 22 2D E8 0D
      43 C3 BF A0 66 99 67 2A
```

Authors' Addresses

Kelley W. Burgin
National Security Agency

EMail: kwburgi@tycho.ncsc.mil

Michael A. Peck
The MITRE Corporation

EMail: mpeck@mitre.org