Keyed-hash Message Authentication Code(*HMAC*)

Taufik Ramadhany – NIM : 13503112

Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
E-mail : if13112@students.if.itb.ac.id

Abstrak

Makalah ini akan membahas deskripsi, algoritma umum, dan contoh penggunaan dari *Keyed-hash Message Authentication Code* (*HMAC*). *HMAC* adalah salah satu tipe dari *Message Authentication Code*(*MAC*) yang berbasis fungsi kriptografi *hash* satu arah. *MAC* sendiri adalah fungsi satu-arah yang menggunakan kunci privat dalam pembangkitan nilai *hash*. Seperti fungsi *MAC* yang lain, *HMAC* ini digunakan untuk memeriksa integritas data dan autentifikasi dari sebuah pesan yang dikirimkan, tapi di lain pihak tidak mendukung prinsip *non-repudiation*. Hal ini dikarenakan pemilikan kunci privat yang bisa diketahui oleh beberapa orang, sehingga tidak memungkinkan untuk mengetahui orang yang membangkitkan nilai *hash* yang sebenarnya.

Selain itu, akan dibahas juga kelebihan dan kekurangan *HMAC*, dan terakhir akan disertakan contoh implementasi dari *HMAC*.

Kata kunci: *Keyed-hash Message Authentication Code, hash function, Message Authentication Code*

1. Pendahuluan

Pada masa sekarang, informasi adalah sesuatu yang sangat mahal harganya. Informasi menjadi barang yang perlu dilindungi dan dijaga dari pihak-pihak yang tidak berhak memilikinya. Informasi yang dijaga tersebut mencakup dokumen atau pesan komunikasi antar beberapa orang. Untuk dokumen, perlindungan yang perlu dilakukan adalah mencegah orang lain mendapatkan akses ke dokumen yang dimaksud. Sedangkan untuk pesan komunikasi, hal yang harus dilakukan adalah mencegah adanya penyusup ketika pesan dikirim, baik itu mengetahui isi pesan atau mengubah pesan asli. Untuk itulah, menyediakan cara untuk memeriksa integritas dari informasi yang telah ditransmisi dari jalur yang tidak dijamin aman adalah hal penting di dunia komunikasi terbuka seperti saat ini.

Mekanisme yang menyediakan pemeriksaan integritas pesan berdasar atas kunci rahasia/privat biasa disebut juga *Message Authentication Code*. Biasanya, *Message Authentication Code* digunakan ketika dua pihak membagi sebuah kunci rahasia/privat untuk melakukan autentikasi terhadap pesan yang ditransmisikan antar pihak tersebut. Ada beberapa macam *Message Authentication Code*(*MAC*), salah satunya adalah *Keyed-hash Message Authentication Code*(*HMAC*). Secara umum, *Keyed-hash Message Authentication Code* adalah teknik *MAC* yang memanfaatkan fungsi *hash* terhadap pesan dan kemudian mengenkripsi pesan tersebut dengan sebuah kunci privat. *MAC* sendiri adalah teknik autentikasi pesan dengan membandingkan nilai *authentication tag* yang telah dihitung oleh pengirim dengan *authentication tag* yang dihitung sendiri oleh penerima.

Keyed-hash Message Authentication Code adalah teknik *Message Authentication Code* yang dibuat oleh Mihir Bellare, Ran Canetti, Hugo Krawczyk pada tahun 1996. Hal ini didasari karena fungsi *hash* itu sendiri tidak memungkinkan penggunaan kunci ketika menghitung nilai *digest*-nya.

Tujuan dari dibangunnya algoritma *HMAC* ini adalah:

a. Untuk menggunakan fungsi *hash*, tanpa modifikasi, yang telah tersedia dan mudah didapatkan.
b. Untuk mempertahankan performansi dari algoritma *hash* yang sudah ada
c. Untuk menggunakan dan mengatur kunci secara mudah
d. Untuk mendapatkan pengertian yang lebih dalam dari analisis kriptografi mengenai kekuatan mekanisme

autentikasi yang berdasarkan fungsi *hash*

e. Untuk mempermudah pengubahan atau penggantian fungsi *hash* yang digunakan apabila algoritma *hash* baru yang lebih cepat atau lebih aman ditemukan.

2. Algoritma *HMAC*

Secara umum, algoritma *HMAC* ini dapat dijelaskan dengan persamaan di bawah ini:

$$HMAC_K(m) = h\left((K \oplus opad) \square h\left((K \oplus ipad) \square m\right)\right)$$

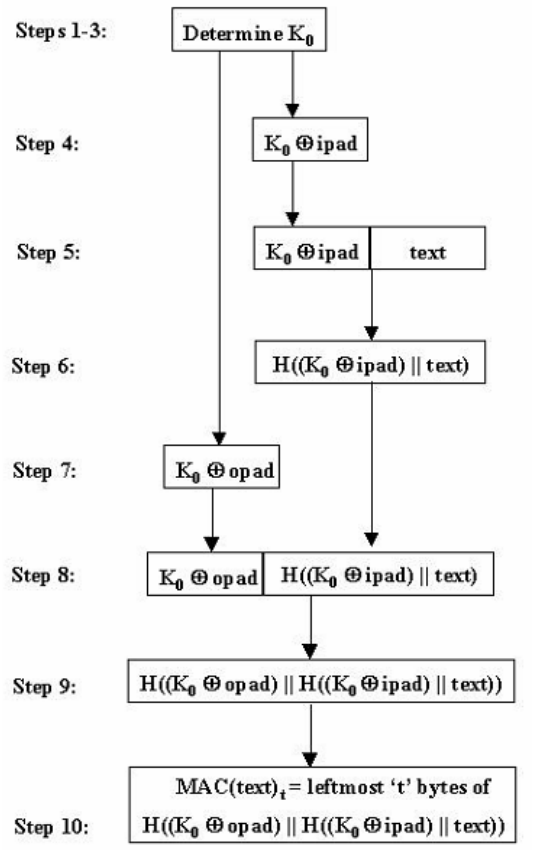
dengan K adalah kunci privat yang diketahui oleh pengirim dan penerima, h adalah fungsi *hash* yang digunakan, `m` adalah pesan yang akan diautentikasi, `opad` adalah 0x5c5c5c...5c dan `ipad` adalah 0x363636...36 dengan panjang yang sama.

2.1 Algoritma *HMAC* Secara Umum

Dari definisi di atas, algoritma *HMAC* dapat dijabarkan menjadi 10 langkah, yaitu:

Step 1. Jika panjang `K = B`, set K_0 = `K`. Kemudian lanjut ke *Step 4*.
Step 2. Jika panjang `K > B`, *hash* `K` untuk mendapatkan `L` *string byte*, kemudian *append* dengan (`B-L`) angka 0 untuk mendapatkan *string byte* K_0 yang panjangnya sama dengan `B`. Kemudian lanjut ke *Step 4*.
Step 3. Jika panjang `K < B`, *append* angka 0 sebanyak (`B-K`) untuk mendapatkan *string byte* K_0 yang panjangnya sama dengan `B`. Kemudian lanjut ke *Step 4*.
Step 4. Lakukan XOR antara K_0 dengan `ipad` untuk menghasilkan *string byte* sepanjang `B`.
Step 5. *Append string* *'text'* ke dalam hasil *string* dari *Step 4* tadi.
Step 6. Lakukan `H` untuk *string* yang dihasilkan oleh *Step 5*.
Step 7. Lakukan XOR antara K_0 dengan *opad*.
Step 8. Lakukan *append string* dari *Step 6* ke dalam *string* dari *Step 7*.
Step 9. Lakukan `H` untuk *string* yang dihasilkan dari *Step 8*.
Step 10. Ambil *leftmost byte* sebanyak `t` dari *string* yang dihasilkan *Step 9*.

Algoritma di atas dapat digambarkan seperti diagram di samping:



Adapun penjelasan dari variabel yang digunakan di dalam diagram dan algoritma di atas adalah:

Variabel	Deskripsi
`B`	Ukuran block(dalam *byte*) dari input ke fungsi *Hash*
`H`	Fungsi *Hash* yang digunakan
`ipad`	inner pad(0x3636...36)
`K`	Kunci privat yang diketahui oleh pengirim dan penerima
K_0	Kunci yang telah diproses seperlunya supaya panjangnya B.
`L`	Ukuran block(dalam *byte*) dari output fungsi *Hash*
`opad`	outer pad(0x5c5c...5c)
`t`	Jumlah *byte* dari *MAC* yang diinginkan
`text`	Pesan/informasi yang akan dimanipulasi

2.2 Penjelasan Algoritma *HMAC*

Langkah pertama yang harus dilakukan pada algoritma *HMAC* ini adalah menormalisasi panjang kunci yang diberikan, sehingga panjangnya sama dengan B. Ukuran kunci K harus lebih dari/sama dengan L/2, dengan L yaitu ukuran keluaran dari fungsi *hash*. Kunci yang lebih besar tidak serta merta meningkatkan keamanan dari fungsi *hash*, hal ini karena kunci tersebut akan di-*hash* terlebih dahulu untuk mendapatkan ukuran yang lebih dari/sama dengan L/2. Tapi sangat disarankan untuk menggunakan kunci yang panjang jika tingkat keacakan dari kunci tersebut lumayan lemah.

Jika panjang kunci telah sama dengan B, maka nilai K_0(kunci yang akan digunakan untuk fungsi *Hash* dan telah dinormalisasi) diisi dengan K. Sedangkan jika panjang kunci kurang dari B, maka kunci K di-append dengan 0 supaya panjangnya sama dengan B. Untuk kasus ketika panjang K melebihi panjang B, maka kunci tersebut di-*hash* terlebih dahulu. Kemudian hasil *hash* ini akan di-append dengan 0 supaya panjangnya sama dengan B, sama dengan halnya jika panjang kunci lebih pendek dari B.

Langkah selanjutnya adalah melakukan XOR antara K_0 dengan ipad. Hasil operasi ini disebut juga dengan inner key.

Langkah kelima adalah melakukan operasi konkatenasi `text` ke string `inner key`.

Langkah keenam adalah menghitung nilai *hash* dari hasil konkatenasi yang baru saja dilakukan.

Langkah berikutnya adalah menghitung nilai outer key, yang didapatkan dari operasi XOR antara K_0 dengan `opad`.

Kemudian dilakukan operasi konkatenasi lagi antara `outer key` dengan nilai *hash* yang dihitung pada langkah keenam.
Langkah kesembilan adalah menghitung nilai *hash* dari hasil konkatenasi tersebut.

Kemudian langkah terakhir adalah mengambil *byte* terkiri sebanyak t dari nilai *hash* dari langkah kesembilan. Langkah terakhir ini sendiri sebenarnya bersifat opsional. Hal ini karena hasil dari fungsi *hash* pada langkah kesembilan sudah dapat diambil sebagai nilai *MAC* dari pesan tersebut, dengan panjang L. Langkah kesepuluh ini dapat dilakukan apabila pengguna menginginkan *authentication tag* yang lebih pendek, dengan hanya mengambil t *byte* saja. Tapi dari spesifikasi *HMAC* disarankan bahwa nilai t ini tidak boleh kurang dari setengah L atau tidak boleh kurang dari 80 untuk kasus tertentu. Jika nilai t kecil, maka *HMAC* ini kemungkinan besar menjadi tidak aman.

2.3 Catatan Implementasi dari *HMAC*

Algoritma *HMAC* ini dispesifikasikan untuk fungsi *hash* yang direkomendasikan, H. Dengan perubahan yang sedikit saja, fungsi *hash* H dapat digantikan dengan fungsi *hash* baru H' dari sebuah implementasi dari *HMAC*.

Sesuai dengan konsep dari algoritma *HMAC* di atas, nilai dari $K_0 \oplus$ `ipad`(inner key) dan $K_0 \oplus$ `opad`(outer key) dapat dihitung sekali saja, yaitu ketika kunci dibangkitkan, atau ketika digunakan untuk pertama kalinya. Hasil operasi XOR yang menghasilkan inner key dan outer key tersebut dapat disimpan dan kemudian digunakan untuk menginisialisasi fungsi *hash* H setiap kali sebuah pesan perlu diautentikasi menggunakan kunci yang sama. Setiap kali mengautentikasi pesan dengan kunci yang sama, hasil operasi XOR tersebut dapat langsung dipakai. Hal ini sangat berguna ketika harus melakukan autentikasi data dalam bentuk aliran yang pendek.

Tapi, sesuai dengan prinsip keamanan, kedua hasil (inner key dan outer key) tersebut harus dijaga dan dilindungi seperti halnya kunci. Hal ini jelas sekali karena `inner key` dan `outer key` tersebut merupakan kunci yang diubah menggunakan sebuah konstanta. Sehingga jika inner key dan outer key diketahui oleh pihak lain, pihak tersebut dapat dengan mudah mendapatkan kunci yang sebenarnya.

2.4 Implementasi Sederhana dari Algoritma *HMAC*

Di bawah ini merupakan implementasi dari algoritma *HMAC* dengan menggunakan bahasa pemrograman C#. Implementasi berikut terbatas pada fungsi dasar dan tidak menyertakan kasus kesalahan tertentu seperti kesalahan masukan dan sebagainya.

```
/*
 * Created by SharpDevelop.
 * User: Taufik Ramadhany
 * Date: 12/26/2006
 * Time: 1:07 PM
 *
 */
using System;
using System.Text;
using System.Collections.Generic;
using System.Security.Cryptography;

namespace My HMAC
{
	/// <summary>
	/// Kelas ini berfungsi untuk melakukan simulasi sederhana
	/// terhadap Keyed-hash Message Authentication Code(HMAC).
	/// </summary>
	class My HMAC
	{
		private static int B;
		private byte[] ipad = new byte[B];
		private byte[] opad = new byte[B];
		byte nol = 0;
		SHA1 md = new SHA1CryptoServiceProvider();

		/// <summary>
		/// Konstruktor dari kelas My HMAC
		/// </summary>
		public My HMAC(){
			init();
		}

		/// <summary>
		/// Method ini berfungsi untuk menginisialisasi nilai B,
		/// ipad dan juga opad, sebelum nilai MAC dihitung
		/// </summary>
		public void init(){
			B = 64;
			for (int i=0;i<B;i++){
				ipad[i] = (byte) 0x36;
				opad[i] = (byte) 0x5c;
			}
		}

		/// <summary>
		/// Method ini berfungsi untuk mengubah sebuah string
		/// menjadi array of byte
		/// </summary>
		/// <param name="text">String yang akan diubah</param>
		/// <returns>Nilai array of byte dari text</returns>
		public byte[] stringToByte(string text){
			byte[] buffer;
			ASCIIEncoding ascii = new ASCIIEncoding();
			buffer = ascii.GetBytes(text);
			return buffer;
```

```
}

/// <summary>
/// Method ini berfungsi untuk mengubah array of byte
/// menjadi sebuah string. Kebalikan dari method
/// stringToByte di atas
/// </summary>
/// <param name="hex">Array of byte yang akan
/// diubah</param>
/// <returns>Nilai string dari hex</returns>
public string byteToString(byte[] hex){
      string result = "";
      for (int i=0;i<hex.Length;i++){
            result += hex[i];
      }
      return result;
}

/// <summary>
/// Method ini berfungsi untuk menghitung nilai MAC dari
/// sebuah message
/// </summary>
/// <param name="message">String yang akan di-HMAC</param>
/// <param name="key">Kunci privat untuk melakukan
/// HMAC</param>
/// <param name="t">Jumlah leftmost byte yang akan
/// diambil</param>
/// <returns>Array of byte nilai MAC yang telah diubah
/// menjadi string</returns>
public string MAC(string message, string key, int t){
      /** Mengubah kunci menjadi array of byte terlebih
       *  dahulu*/
      byte[] k = stringToByte(key);
      /** Mengubah message menjadi array of byte terlebih
       *  dahulu*/
      byte[] text = stringToByte(message);
      byte[] k0;        //kunci yang telah dinormalisasi
      string result;    //string hasil HMAC

      /* Step 1: Jika panjang kunci K = B
       *          maka K0 langsung diisi dengan K*/
      if (k.Length == B){
            k0 = k;
      } else {
            /* Step 2: Jika panjang kunci K > B
             *          maka kunci K di-hash terlebih
             *          dahulu*/
            if (k.Length > B){
                  k = md.ComputeHash(k);
            }
            /* Step 3: Jika panjang kunci K < B
             *          maka kunci K di-append dengan 0*/
            if (k.Length < B){
                  k0 = new byte[B];
                  /* Mengisi nilai K0 dengan nilai K
                   * terlebih dahulu*/
```

```
            for (int i=0;i<k.Length;i++){
                  k0[i] = k[i];
            }
            /* Mengisi byte sisa dengan 0*/
            for (int i=k.Length;i<B;i++){
                  k0[i] = nol;
            }
      }
}

/* Step 4: Operasi XOR antara kunci K dengan ipad
 *          Menghasilkan inner key*/
byte[] innerKey;
for (int i=0;i<B;i++){
      innerKey[i] = k0[i] ^ ipad[i];
}

/* Step 5: Operasi append string text ke inner key*/
int j = 0;
for(int i=innerKey.Length; i<innerKey.Length +
      text.Length; i++){
      innerKey[i] = text[j];
      j++;
}

/* Step 6: Menghitung nilai hash dari hasil append
 * sebelumnya*/
byte[] hash1 = md.ComputeHash(innerKey);

/* Step 7: Operasi XOR antara kunci K dengan opad
 *          Menghasilkan outer key*/
byte[] outerKey;
for (int i=0;i<B;i++){
      outerKey[i] = k0[i] ^ ipad[i];
}

/* Step 8: Operasi append nilai hash yang didapatkan
** dari Step 6 ke outer key*/
j = 0;
for (int i=outerKey.Length; i<outerKey.Length +
      hash1.Length; i++){
      outerKey[i] = hash1[j];
      j++;
}

/* Step 9: Menghitung nilai hash dari hasil append
 * Step 8*/
byte[] hashFinal = md.ComputeHash(outerKey);

/* Step 10: Mengambil leftmost byte sebanyak t*/
/* Memeriksa nilai t terlebih dahulu*/
if (t < hashFinal.Length/2){
      t = hashFinal.Length/2;
} else if (t > hashFinal.Length){
      t = hashFinal.Length;
}
```

```
      /* Mengambil t leftmost byte*/
      byte[] MACValue = new byte[t];
      for(int i=0;i<hashFinal.Length;i++){
            MACValue[i] = hashFinal[i];
      }

      /* Mengubah byte yang telah diambil menjadi string*/
      result = byteToString(MACValue);

      return result;
}

/// <summary>
/// Method ini berfungsi untuk melakukan verifikasi
/// terhadap sebuah nilai MAC dari string tertentu dengan
/// kunci privatnya
/// </summary>
/// <param name="oMAC">Nilai MAC dari string yang
/// diketahui</param>
/// <param name="message">String yang akan
/// diverifikasi</param>
/// <param name="key">Kunci privat untuk HMAC</param>
/// <param name="t">Jumlah leftmost byte yang akan
/// diambil</param>
/// <returns>Nilai boolean apakah nilai MAC valid atau
/// tidak</returns>
public bool verify(string oMAC, string message, string key,
      int t){
      bool valid = false;
      string newMAC = MAC(message, key, t);
      if (newMAC.Equals(oMAC)){
            valid = true;
      }
      return valid;
}

public static void Main(string[] args)
{
      string message = "contoh message yang akan di-HMAC
            oleh kelas ini, panjang string ini tidak
            ditentukan";
      string key = "contoh key yang digunakan";
      MyHMAC HMAC = new MyHMAC();
      Console.WriteLine("String message = " + message);
      Console.WriteLine("String key = " + key);
      string MACValue = HMAC.MAC(message, key, 32);
      Console.WriteLine("String hasil MAC = " + MACValue);
      string testMAC = "tes validasi";
      if (HMAC.verify(testMAC, message, key, 32)){
            Console.WriteLine(testMAC + " adalah nilai MAC
                  dari " + message);
      } else {
            Console.WriteLine(testMAC + " bukan nilai MAC
                  dari " + message);
      }
      Console.ReadLine();
}
```

```
        }
}
```

2.5 Fungsi *Hash* untuk *HMAC*

Ada beberapa fungsi *hash* yang mungkin digunakan untuk algoritma *HMAC*. Fungsi *hash* tersebut adalah:

a. *SHA1* dan variannya

Fungsi *hash SHA1* dan keluarganya seperti *SHA224, SHA256, SHA512* adalah salah satu algoritma yang paling banyak digunakan dalam implementasi *Keyed-hash Message Authentication Code* atau *HMAC* ini. Fungsi *hash SHA1* menghasilkan output sebanyak 160 bit, *SHA224* menghasilkan output sebanyak 224 bit, *SHA256* menghasilkan output sepanjang 256, dan *SHA512* menghasilkan output sepanjang 512 bit. Walaupun banyak digunakan, tetapi pada beberapa tahun belakangan pada fungsi *hash SHA1* ini ditemukan adanya *collision* yang dianggap sudah memasuki tahap fatal oleh Christian Rechberger and Christophe De Cannière pada tahun 2006. Sehingga, penggunaan *SHA1* ini tidak dianjurkan lagi oleh kalangan kriptografi. Untuk keluarga fungsi *hash SHA* yang lain, serangan yang ampuh belum bisa dianggap menghancurkan fungsi ini, sehingga masih layak digunakan untuk algoritma *HMAC*.

b. *MD5*

Bersama dengan fungsi *hash SHA1*, fungsi *MD5* adalah fungsi *hash* yang banyak digunakan dalam pengimplementasian algoritma *HMAC*. *MD5* sendiri adalah *hash* yang diciptakan oleh Ronald Rivest pada tahun 1991. *MD5* ini menghasilkan output *hash* sepanjang 128 bit, lebih pendek daripada keluarga *hash SHA*. Hal ini menyebabkan keamanan yang diberikan oleh *MD5* kurang terjamin dibandingkan dengan *SHA* dan variannya. Adanya *collision* juga telah ditemukan dari percobaan oleh Xiaoyun Wang dan beberapa peneliti lainnya pada tahun 2005. Bahkan pada Maret 2006, peneliti bernama Vlastimil Klima telah berhasil menemukan algoritma yang bisa menemukan *collision* pada *MD5* dalam waktu yang kurang dari satu menit. Penggunaan MD5 juga sudah tidak dianjurkan lagi.

c. *Tiger Hash*

Tiger hash adalah fungsi kriptografi *hash* yang diciptakan oleh Ross Anderson dan Eli Biham pada tahun 1995. Fungsi *hash* ini menghasilkan output *digest* sepanjang 192 bit. Algoritma *Tiger hash* ini dikembangkan menggunakan paradigma Merkle-Damgard. *Collision* pada algoritma ini juga sudah ditemukan oleh peneliti bernama John Kelsey dan Stefan Lucks pada *Tiger hash*.

3. Penggunaan *HMAC*

Seperti yang telah dijelaskan pada bagian sebelumnya, *Keyed-hash Message Authentication Code* menggunakan fungsi *hash* untuk menghitung nilai dari isi pesan. Fungsi *hash* yang bisa digunakan oleh *HMAC* tidak terbatas, bergantung kepada keinginan dari orang yang ingin mengimplementasikan *HMAC* itu sendiri. Fungsi *hash* sendiri awalnya tidak didesain untuk digunakan bersama dengan kunci, karena itu cara pengoptimalan penggunaan fungsi *hash* ini juga tidak pasti. Oleh sebab itu, sangat disarankan untuk berhati-hati dalam menggunakan fungsi *hash* di dalam *MAC*.

3.1 Contoh penggunaan *HMAC*

Pada contoh di bawah ini, *HMAC* diaplikasikan dengan menggunakan fungsi *hash* SHA-1.

a. Contoh tanpa pemotongan nilai *MAC* dengan kunci 64-*byte*

Pesan (Text) : `Sample #1`

Key :
```
00010203 04050607 08090a0b 0c0d0e0f
10111213 14151617 18191a1b 1c1d1e1f
20212223 24252627 28292a2b 2c2d2e2f
30313233 34353637 38393a3b 3c3d3e3f
```
K0 :
```
00010203 04050607 08090a0b 0c0d0e0f
10111213 14151617 18191a1b 1c1d1e1f
20212223 24252627 28292a2b 2c2d2e2f
30313233 34353637 38393a3b 3c3d3e3f
```

K0 ⊕ ipad :
```
36373435 32333031 3e3f3c3d 3a3b3839
26272425 22232021 2e2f2c2d 2a2b2829
16171415 12131011 1e1f1c1d 1a1b1819
06070405 02030001 0e0f0c0d 0a0b0809
```

K0 ⊕ ipad || Text :
```
36373435 32333031 3e3f3c3d 3a3b3839
26272425 22232021 2e2f2c2d 2a2b2829
16171415 12131011 1e1f1c1d 1a1b1819
06070405 02030001 0e0f0c0d 0a0b0809
53616d70 6c652023 31
```

Hash(K0 ⊕ ipad || Text):
```
bcc2c68c abbbf1c3 f5b05d8e 7e73a4d2
7b7e1b20
```

K0 ⊕ opad :
```
5c5d5e5f 58595a5b 54555657 50515253
4c4d4e4f 48494a4b 44454647 40414243
7c7d7e7f 78797a7b 74757677 70717273
6c6d6e6f 68696a6b 64656667 60616263
```

(K0 ⊕ opad) || *Hash*(K0 ⊕ ipad || Text):
```
5c5d5e5f 58595a5b 54555657 50515253
4c4d4e4f 48494a4b 44454647 40414243
7c7d7e7f 78797a7b 74757677 70717273
6c6d6e6f 68696a6b 64656667 60616263
bcc2c68c abbbf1c3 f5b05d8e 7e73a4d2
7b7e1b20
```

Hash((K0 ⊕ opad) || *Hash*(K0 ⊕ ipad || Text)):
```
4f4ca3d5 d68ba7cc 0a1208c9 c61e9c5d
a0403c0a
```

Maka nilai *MAC*-nya adalah :
```
4f4ca3d5 d68ba7cc 0a1208c9 c61e9c5d
```

b. Contoh dengan pemotongan nilai *MAC* dengan kunci 49-*byte*

Pesan (Text) : `Sample #4`

Key :
```
70717273 74757677 78797a7b 7c7d7e7f
80818283 84858687 88898a8b 8c8d8e8f
90919293 94959697 98999a9b 9c9d9e9f
a0
```

K0 :
```
70717273 74757677 78797a7b 7c7d7e7f
80818283 84858687 88898a8b 8c8d8e8f
90919293 94959697 98999a9b 9c9d9e9f
a0000000 00000000 00000000 00000000
```

K0 ⊕ ipad :
```
46474445 42434041 4e4f4c4d 4a4b4849
b6b7b4b5 b2b3b0b1 bebfbcbd babbb8b9
a6a7a4a5 a2a3a0a1 aeafacad aaaba8a9
96363636 36363636 36363636 36363636
```

K0 ⊕ ipad || Text :
```
46474445 42434041 4e4f4c4d 4a4b4849
b6b7b4b5 b2b3b0b1 bebfbcbd babbb8b9
a6a7a4a5 a2a3a0a1 aeafacad aaaba8a9
96363636 36363636 36363636 36363636
53616d70 6c652023 34
```

Hash(K0 ⊕ ipad || Text):
```
bf1e889d 876c34b7 bef3496e d998c8d1
16673a2e
```

K0 ⊕ opad :
```
2c2d2e2f 28292a2b 24252627 20212223
dcdddedf d8d9dadb d4d5d6d7 d0d1d2d3
cccdcecf c8c9cacb c4c5c6c7 c0c1c2c3
fc5c5c5c 5c5c5c5c 5c5c5c5c 5c5c5c5c
```

(K0 ⊕ opad) || *Hash*(K0 ⊕ ipad || Text):
```
2c2d2e2f 28292a2b 24252627 20212223
dcdddedf d8d9dadb d4d5d6d7 d0d1d2d3
cccdcecf c8c9cacb c4c5c6c7 c0c1c2c3
fc5c5c5c 5c5c5c5c 5c5c5c5c 5c5c5c5c
bf1e889d 876c34b7 bef3496e d998c8d1
16673a2e
```

Hash((K0 ⊕ opad) || *Hash*(K0 ⊕ ipad || Text)):
```
9ea886ef e268dbec ce420c75 24df32e0
751a2a26
```

Maka nilai *MAC*-nya adalah(karena hanya 12 *byte* yang diambil) :
```
9ea886ef e268dbec ce420c75
```

3.2 *Distinguisher* dari *HMAC*

Algoritma *HMAC* ini sendiri bisa dioptimalkan dengan mengubah struktur umum dari *HMAC* untuk mengetahui adanya *collison* pada algoritma *HMAC*. Optimalisasi ini bisa dicapai dengan dua macam pembeda(*distinguisher*). Dua macam tersebut adalah:

a. *Differential distingusiher*

Untuk mendesain *differential distinguisher* dari *HMAC* dibutuhkan sepasang pesan yang akan mengakibatkan *collision* dari hasil nilai *MAC* kedua pesan tersebut.

Caranya adalah:

1. Pilih sebuah pesan secara acak M. Kemudian bangkitkan pesan lain M' dari hasil operasi XOR antara M dengan sebuah variabel A. Panjang A harus sama dengan M.(M' = M ⊕ A)
2. Dengan skenario *chosen message*, cari nilai *HMAC* C1 dari pesan M dan C2 dari pesan M'.
3. Periksa apakan hasil C1 ⊕ C2 = 0.

b. *Rectangle distinguisher*

Untuk mendesain *rectangle distinguisher* dari *HMAC* dibutuhkan 4 buah pesan yang akan mengakibatkan 2 buah *collision* dari hasil nilai *MAC* kedua pesan tersebut. Pada *MAC*, ada kemungkinan terjadinya *non-bijectivity*, yaitu dua pesan yang berbeda memiliki nilai *MAC* yang sama atau nilai antara(*intermediate*) yang sama.

Cara untuk *rectangle distinguisher*:
1. Pilih 2 buah pesan secara acak M dan N. Kemudian bangkitkan dua pesan lain M' dan N' yang didapatkan dari hasil operasi XOR, M' = M XOR dengan A, N' = N XOR dengan A. Panjang M, N dan A harus sama.(M' = M $\oplus$ A, N' = N $\oplus$ A)
2. Dengan skenario *chosen message*, cari nilai *HMAC* C1 dari pesan M, C1' dari pesan M', C2 dari pesan N, dan C2' dari pesan N'.
3. Periksa apakah hasil XOR C1 $\oplus$ C2 = C1' $\oplus$ C2' = 0 atau C1 $\oplus$ C2' = C2 $\oplus$ C1' = 0.

Distinguisher ini digunakan untuk memeriksa terjadinya *collision* pada algoritma *HMAC*. Penggunaannya dijelaskan pada bagian di bawah ini dan menggunakan *forgery attack*(pemalsuan) sebagai perbandingan:

Pada bagian ini, variabel p menunjukkan serangan *distinguisher*, sedangkan q untuk *forgery*. Serangan ini akan dilakukan pada *HMAC* yang berbasis *MD5* dan *SHA1*. Skenario percobaan pertama dengan menggunakan p dan *rectangle distinguish* adalah:

1. Kumpulkan pasangan pesan sebanyak $2^{(l+1)/2} \cdot p^{-1}$(M_i, M_i') dengan perbedaan A, dan panjang $M_i = M_i' = t$.
2. Dengan skenario *chosen message attack*, hitung nilai *MAC*-nya. Untuk setiap pasangan pesan (M_i, M_i'), maka pasangan nilainya adalah (C_i, C_i').
3. Periksa apakah $C_i \oplus C_j = C_i' \oplus C_j' = 0$ atau $C_i \oplus C_j' = C_i' \oplus C_j = 0$ dengan $1 \le i < j \le 2^{(l+1)/2}.p^{-1}$.

   Jika ada salah satu dari kuartet pesan yang memenuhi persamaan di atas, maka algoritma *MAC* = *HMAC*, jika tidak maka algoritma *MAC* = sebuah fungsi random.

Tingkat kesuksesan dari serangan ini adalah:

$$\frac{1-(1-2^{-l}\cdot\hat{p}^2)^{2^{l+1}\cdot\hat{p}^{-2}}}{2}+\frac{(1-2^{-2l})^{2^{l+1}\cdot\hat{p}^{-2}}}{2}\approx\frac{1-e^{-2}}{2}+\frac{e^{-2^{-l+1}\cdot\hat{p}^{-2}}}{2}.$$

Perkiraan nilai kesuksesan pada percobaan pertama ini adalah 0.43.
Untuk percobaan kedua, yang digunakan adalah q dan *differential distinguisher* dengan skenario:

1. Kumpulkan 2. q^{-1} pasangan pesan (M_i, M_i') dengan perbedaan A, dan panjang $M_i = M_i' = t$.
2. Dengan skenario *chosen message attack*, hitung nilai *MAC*-nya. Untuk setiap pasangan pesan (M_i, M_i'), maka pasangan nilainya adalah (C_i, C_i').
3. Periksa apakah $C_i \oplus C_i' = 0$. Jika ada minimal satu buah pesan yang memenuhi persamaan di atas, maka algoritma *MAC* = *HMAC*, jika tidak maka algoritma *MAC* = sebuah fungsi random.

Tingkat kesuksesan dari percobaan kedua ini adalah

$$\frac{1-(1-q)^{2\cdot q^{-1}}}{2}+\frac{(1-2^{-l})^{2\cdot q^{-1}}}{2}\approx\frac{1-e^{-2}}{2}+\frac{e^{-2^{-l+1}\cdot q^{-1}}}{2}.$$

Terakhir, percobaan serangan forgery dengan menggunakan q dan differential distinguisher:

1. Kumpulkan 2. q^{-1} pasangan pesan (M_i, M_i') dengan perbedaan A, dan panjang $M_i = M_i' = t$.
2. Dengan skenario *chosen message attack*, hitung nilai *MAC*-nya. Untuk setiap pasangan pesan (M_i, M_i'), maka pasangan nilainya adalah (C_i, C_i').
3. Periksa apakah $C_i \oplus C_i' = 0$ dan cari pasangan nilai MAC dari $M_i \| P$ dan juga $M_i' \| P$ di mana M dan M_i' adalah pesan yang memiliki nilai MAC yang sama, dan P adalah sebuah string tidak-kosong.

Dari percobaan di atas, diketahui bahwa tingkat kesuksesan untuk MD5 adalah 0.92 dan untuk SHA1 adalah 0.93.

4. Aspek Keamanan pada *HMAC*

4.1 Hal-hal yang patut diperhatikan

Ada beberapa hal yang perlu diperhatikan dalam aspek keamanan dari *HMAC*. Hal-hal tersebut antara lain:

a. Keamanan kunci
Keamanan *HMAC* sangat tergantung kepada tingkat kerahasiaan dari kunci. Maka itu, pengguna harus menjaga kunci ini dari tindakan yang membahayakan. Standar *HMAC* ini juga tidak menjamin bahwa implementasinya aman. Keamanan adalah tanggung jawab sepenuhnya dari pengguna standar *HMAC* untuk memastikan implementasinya aman.

b. Keamanan fungsi *hash*
Faktor yang sangat penting dalam aspek keamanan dari *HMAC* adalah keamanan fungsi *hash* yang digunakan itu sendiri. Hal ini disebabkan satu-satunya cara *HMAC* gagal adalah ketika fungsi *hash* yang digunakan juga gagal. Fungsi *hash* sendiri sangat rentan terhadap *collision*. Collision akan dijelaskan lebih lanjut di bagian 4.2.b.

c. Pemalsuan
Keamanan dari *MAC* berarti keamanannya dari pemalsuan. *MAC* dikatakan gagal jika seorang penyusup yang tidak memiliki kunci K, bisa menemukan beberapa pesan `Text` bersama dengan nilai *MAC*-nya. Penyusup tersebut diasumsikan dapat mengumpulkan sejumlah contoh dari teks dan nilai *MAC*-nya yang valid dengan melakukan observasi terhadap jalur aliran data antara pengirim dan penerima. Bahkan sang penyusup bisa melakukan chosen message attack dengan mempengaruhi pemilihan pesan yang akan dihitung nilai *MAC*-nya oleh pengirim. Karena itulah, keamanan dari *MAC* ini bisa dianggap sebagai ukuran kemungkinan terjadinya pemalsuan yang berhasil dengan melakukan serangan seperti itu.

4.2 Jenis Serangan/Ancaman pada *HMAC*

a. *Birthday attack*
Birthday attack adalah sebuah jenis serangan di dalam konteks kriptografi yang menggunakan perhitungan matematis. Tujuan dari serangan ini adalah mencari 2 buah pesan yang hampir sama yang memiliki nilai fungsi *hash* yang sama. Setelah salah satu pesan ditandatangani, pihak penyusup menggunakan nilai fungsi *hash* tersebut untuk ditempelkan pada pesan yang lain.

Contohnya:
Alice ingin menjebak Bob untuk menandatangani kontrak yang telah dicurangi. Alice menyiapkan kontrak yang benar M dan kontrak yang salah M'. Kemudian Alice akan menemukan posisi pada pesan M yang bisa diubah tanpa mengubah artinya, seperti menambahkan tanda baca koma, baris kosong, dua spasi dan sebagainya. Dengan mengombinasikan perubahan tesebut, Alice dapat menciptakan banyak variasi dari M. Hampir sama dengan tindakan sebelumnya, Alice juga melakukan perubahan pada kontrak M'. Dia kemudian menghitung nilai fungsi *hash* pada kedua pesan kontrak tersebut sehingga nilai kontrak yang benar dan kontrak yang salah memiliki nilai fungsi *hash* yang sama. Kemudian dia memberikan kontrak yang benar untuk ditandatangani oleh Bob. Bob yang tidak mengetahui hal ini kemudian memberikan tanda tangan pada kontrak yang benar. Setelah itu, Alice mengambil tanda tangan dari kontrak tersebut dan menambahkannya kepada kontrak yang salah. Dengan demikian secara tidak langsung Bob telah menandatangani kontrak tersebut.

Ada beberapa cara untuk menghindari serangan semacam ini. Yang pertama adalah dengan memilih panjang output dari fungsi *hash* sepanjang mungkin. Sehingga kemungkinan dari serangan ini secara komputasi tidak masuk akal. Cara yang kedua adalah Bob harus mengubah beberapa bagian dari kontrak yang disodori oleh Alice sebelum ditandatangani. Tetapi, cara yang kedua ini juga tidak menyelesaikan permasalahan dari *Birthday attack* ini. Hal ini disebabkan bahwa Alice bisa mencurigai Bob melakukan *Birthday attack* kepada dirinya.

Serangan *Birthday attack* ini merupakan serangan yang paling berbahaya yang bisa dilakukan kepada sebuah *Message Authentication Code.*

b. *Collision attack*
Collision attack adalah celah pada fungsi kriptografi *hash* yang menghasilkan nilai *hash* yang sama untuk dua buah pesan yang jauh berbeda. Pada fungsi *hash* MD5 dan SHA1, celah *collision* ini telah ditemukan oleh beberapa orang peneliti kriptografi. Contohnya *collision* pada SHA1 ditemukan oleh Xiaoyun Wang, Yiqun Lisa Yin dan Hongbu Yo pada bulan Februari 2005.

c. *Preimage attack*
Preimage attack memiliki jenis serangan yang hampir sama dengan *collision attack*. Bedanya, pada *preimage* dibutuhkan 2^n operasi supaya serangan pada nilai *hash* sebanyak n-bit sukses. Sedangkan pada *collision attack*, jumlah operasi yang dibutuhkan adalah $2^{n/2}$ operasi.
Ada dua jenis *preimage attack* yang biasa dilakukan. Pertama, mencari sebuah pesan sehingga nilai *hash* dari pesan tersebut sama dengan nilai H yang telah didefiniskan. Yang kedua adalah mencari pesan M2 sehingga nilai *hash*-nya sama dengan nilai *hash* dari pesan M1 yang telah didefinisikan.

d. *Brute Force attack*
Jenis serangan *brute force* ini adalah salah satu jenis serangan yang paling umum dalam dunia kriptografi. Tipikal dari serangan *brute force* adalah mencoba semua kemungkinan yang ada untuk mendapatkan nilai tertentu. Pada *HMAC* ini, serangan ini akan mencoba kemungkinan pesan yang benar untuk sebuah nilai *hash* yang diketahui.

5. Kelebihan dan Kekurangan *HMAC*

Kelebihan:
1. Keamanannya terjamin
2. Fungsi *hash* bisa diganti dengan yang lain
3. Adanya pemotongan output mengurangi kemungkinan serangan terhadap algoritma *HMAC*

Kekurangan:
1. Keamanannya sangat bergantung kepada fungsi *hash* yang digunakan
2. Operasi yang digunakan masih sederhana

6. Perbandingan dengan *MAC* Lain

Karena algoritma *MAC* yang lain banyak dan umumnya kompleks, maka perbandingan yang dilakukan hanya sebatas pengertian dan konsep dari algoritma *MAC* tersebut. Algoritma *MAC* yang dibandingkan:

a. *PMAC*(*Paralellizeble MAC*)
Paralellizable *MAC* adalah algoritma *MAC* yang diciptakan oleh Phillip Rogaway. Berbeda dengan *HMAC* yang berbasis fungsi *hash*, *PMAC* ini adalah *MAC* yang berbasis kepada *block cipher*. Nilai *MAC* yang dihasilkan efisien dan telah terbukti aman terhadap *block cipher* yang digunakan.

b. *CMAC*(*Cipher-based MAC*)
Sama seperti *PMAC* di atas, *CMAC* juga merupakan algoritma *MAC* yang berbasis kepada *block cipher*, bukan fungsi *hash*. Algoritma ini biasanya digunakan memberikan jaminan autentikasi dan integritas dari data biner.
Algoritma ini juga aman digunakan untuk data pesan yang panjangnya tidak tetap. Pada algoritma ini kunci yang digunakan juga dipecah menjadi dua buah kunci k_1 dan k_2. Karena berbasis cipher blok, maka pesan dibagi terlebih dahulu menjadi beberapa blok yang ukurannya telah ditentukan. Kemudian untuk setiap blok tersebut akan dilakukan operasi XOR dengan kunci k_1 atau k_2.

c. *UMAC*(*MAC based on Universal Hashing*)
Seperti yang telah dijelaskan pada namanya, *UMAC* ini menggunakan fungsi *hash* sebagai basisnya. Perbedaannya dengan *HMAC* adalah pada *UMAC* ini, fungsi *hash* yang digunakan lebih dari satu. Dan ketika mencari nilai fungsi *hash* dari sebuah pesan, fungsi yang digunakan dipilih secara acak. Dengan demikian, jika seorang kriptanalis ingin mengubah pesan, ia harus mengetahui fungsi *hash* yang digunakan beserta kunci privatnya. Karena itu, peluang *UMAC* gagal mengidentifikasi perubahan ini paling besar 1/D, dimana D adalah jumlah nilai *hash* yang mungkin. Adapun universal di sini maksudnya ialah kemungkinan pesan adalah 0 dan 1, D={0,1}, dan fungsi *hash* H memiliki operasi identitas dan negasi.

d. *NMAC*
Algoritma *NMAC* ini adalah generalisasi dari *Keyed-hash Message Authentication Code*. Pada algoritma ini, kunci privat yang digunakan adalah 2 buah(K_1, K_2). Struktur umum dari algoritma ini juga sama dengan *HMAC*. Perbedaan mendasar antara *NMAC* dengan *HMAC* adalah penggantian *inner key* dan *outer key* pada *HMAC* dengan K_1 dan K_2 pada *NMAC*. Pada beberapa tulisan, *NMAC* ini disebut sebagai dasar dari *HMAC*, yang berarti *HMAC* sebenarnya dikembangkan berdasarkan *NMAC*. *NMAC* ini disebut juga sebagai *two-key HMAC*, yang berarti algoritma *HMAC* bisa saja menggunakan dua kunci daripada satu kunci seperti yang telah dijelaskan. Kunci ini kemudian akan dianggap sebagai kunci K_1 dan kunci K_2. Sedangkan algoritma standar *HMAC* yang hanya menggunakan satu buah kunci akan disebut juga sebagai *one-key HMAC*.

7. Kesimpulan

Keyed-hash Message Authetication Code adalah salah satu algoritma *Message Authetication Code* yang umum digunakan. Biasanya *HMAC* ini diaplikasikan pada aplikasi *web* yang membutuhkan koneksi yang aman, contohnya jual-beli *online*. *HMAC* ini memiliki keunggulan pada pemakaian kunci kriptografi dan juga *truncated output*(pemotongan nilai *MAC*).

HMAC juga memungkinkan penggantian fungsi *hash* yang digunakan secara mudah. Walaupun begitu, keamanan dari *HMAC* ini sangat

bergantung kepada fungsi *hash* yang digunakan. Jika fungsi *hash* yang dipakai mempunyai banyak celah keamanan, maka *HMAC* yang diimplementasikan juga akan memiliki banyak celah yang bisa dimanfaatkan oleh pihak penyusup.

DAFTAR PUSTAKA

[1]http://csrc.nist.gov/publications/
[2] http://doc.astro-wise.org/HMAC.html
[2] http://en.wikipedia.org/wiki/HMAC
[3]http://homes.esat.kuleuven.be/%7Ekjongsun/papers/scn02006.pdf
[4]http://the.jhu.edu/upe/2002/01/18/HMAC1-the-keyed-hash-based-MAC-function/
[5] http://tools.ietf.org/html/rfc2104
[6]http://www-cse.ucsd.edu/~mihir/papers/HMAC.html