Otentikasi Komunikasi *Server-to-Server* Menggunakan HMAC

Steve Andreas Immanuel 13517039
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13517039@std.stei.itb.ac.id

Abstrak—Komunikasi antar *server* sekarang adalah suatu hal yang sangat sering ditemui pada praktek implementasi *service*. Oleh karena itu, untuk memastikan bahwa data yang dipertukarkan tetap aman dan terjaga diperlukan suatu protokol komunikasi tertentu yang disetujui oleh semua pihak yang berkomunikasi. Biasanya komunikasi antar *server* dilakukan menggunakan protokol HTTP lewat Internet secara global. SSL menggunakan protokol HTTPS telah menangani permasalahan seperti *confidentiality* dan *integrity*, namun belum menangani otentikasi *client* (pihak pengirim) oleh *server* (pihak pemroses). Salah satu cara untuk melakukan otentikasi tersebut adalah dengan menggunakan *keyed-hash message authentication code* (HMAC). Selain melakukan otentikasi, HMAC juga memberikan tambahan keamanan di bidang *data integrity*. Pada makalah ini, diimplementasikan otentikasi menggunakan HMAC pada komunikasi antar server.

Kata Kunci—otentikasi, *integrity*, HMAC, *server-to-server*.

I. PENDAHULUAN

Microservice merupakan salah satu arsitektur yang sekarang sedang marak digunakan oleh banyak perusahaan berbasis teknologi. Ide dasarnya adalah dengan memecah satu *service* besar menjadi *service-service* kecil dengan tanggung jawab masing-masing. Namun dengan menggunakan arsitektur seperti ini, berarti muncul adanya tambahan *overhead* untuk komunikasi antar *service* satu dengan yang lain. Kasus yang paling sering terjadi adalah *service-service* tersebut terletak tersebar dan saling berkomunikasi menggunakan protokol HTTP.

Dengan banyaknya proses komunikasi yang harus dilakukan, tentu saja dibutuhkan mekanisme keamanan tertentu untuk menjaga keamanan data yang dipertukarkan. Salah satu upaya untuk menangani hal tersebut adalah dengan menggunakan SSL (*Secure Socket Layers*) pada protokol komunikasi HTTPS. Dengan demikian, semua data akan dipertukarkan dalam bentuk terenkripsi dan baru didekripsi pada penerimanya sehingga akan menghindari adanya serangan yang dilakukan pada saat pengiriman data / *Man in The Middle Attack*.

SSL baru memberikan otentikasi kepada *client* bahwa *server* yang sedang diajak berkomunikasi memang terbukti identitasnya berdasarkan *certificate*. Namun hal ini tidak berlaku kebalikannya. *Server* tidak melakukan otentikasi terhadap *client* mengenai identitasnya [1]. Hal tersebut masuk akal jika *client* yang melakukan *request* adalah *browser* biasa. Namun pada komunikasi antar *server* di mana satu bertindak sebagai "*server*" dan satu bertindak sebagai "*client*" tentu saja identitas keduanya perlu diotentikasi kebenarannya.

Otentikasi dapat dilakukan dengan menggunakan *keyed-hash message authentication code* (HMAC). Pada HMAC, terdapat *secret key* yang hanya dimiliki oleh pihak-pihak yang saling berkomunikasi. Kemudian setiap *request*/pertukaran data antar pihak akan ditambahkan HMAC *signature* yang didapatkan dengan mengombinasikan *secret key* dengan konten *request*. *Signature* tersebut kemudian dapat digunakan untuk memverifikasi pengirim dari *request* dan juga kebenaran isi konten *request*.

Pada prakteknya, komunikasi menggunakan protokol HTTPS tidak selalu diimplementasikan dikarenakan berbagai macam alasan misalnya *developer* tidak mau menangani masalah *certificate* dan konfigurasinya, tidak mau keluar ongkos untuk membeli *certificate*, atau mungkin tidak mau adanya *overhead* komunikasi karena pada HTTPS diperlukan *initial handshake* untuk *generate shared secret key* antar pihak yang berkomunikasi [2]. Mungkin juga ada kasus di mana *data confidentiality* bukanlah hal yang diprioritaskan namun hanya *data integrity* yang diinginkan sehingga penggunaan HMAC menjadi cocok.

II. DASAR TEORI

Pada komunikasi antar *server*, salah satu akan bertindak seolah sebagai "*client*" sedangkan satunya bertindak seolah sebagai "*server*". Untuk membedakan istilah *client* yang berarti *end user* atau *browser*, maka akan digunakan istilah *server requestor* untuk *server* yang bertindak sebagai "*client*" dan *server processor* untuk *server* yang bertindak sebagai "*server*".

A. Otentikasi

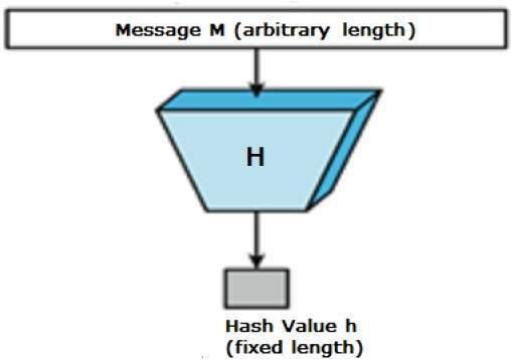
Sebelum dapat dilakukan komunikasi antar *server*, langkah pertama yang perlu dilakukan adalah memverifikasi siapa identitas dari masing-masing pihak yang melakukan komunikasi. [3] Otentikasi adalah suatu proses yang dilakukan untuk memvalidasi identitas. Dalam hal ini, *server processor* memvalidasi apakah pengirim *request* memang benar *server requestor* yang dikenalinya.

Pada komunikasi menggunakan protokol HTTP, untuk melakukan otentikasi biasanya *client* akan memberikan sepotong informasi identitas dalam bentuk tertentu bersama dengan *request* yang ia kirimkan ke *server*. Kemudian *server* akan memverifikasi informasi identitas tersebut. Informasi identitas ini dapat berupa *token*, *signature*, dan lain-lain dan dapat diletakkan pada *request header* atau langsung pada *request body*.

B. Fungsi Hash

[4] Fungsi *hash* adalah suatu fungsi yang melakukan kompresi terhadap suatu *arbritary input* dan menghasilkan *string* dengan panjang tertentu. Keluaran fungsi *hash* disebut sebagai *message-digest* atau *hash-value*. Fungsi hash adalah salah satu kakas yang paling penting di dunia kriptografi karena banyak digunakan untuk mengimplementasikan berbagai kebutuhan keamanan seperti otentikasi, *digital signature*, *pseudo number generator*, dan lain-lain. Secara garis besar, terdapat dua tipe fungsi *hash*, yaitu:

- *Keyed Hash Function*, fungsi *hash* yang menggunakan *secret key* untuk melakukan kompresi.
- *Un-keyed Hash Function*, fungsi *hash* yang tidak memerlukan *secret key* untuk melakukan kompresi.



Gambar 1. Ilustrasi fungsi *hash*
Sumber: https://www.tutorialspoint.com/cryptography/images/hash_functions.jpg

Berbeda dengan algoritma enkripsi dan dekripsi pada umumnya, fungsi *hash* memiliki sifat satu arah. Artinya hasil keluaran dari fungsi *hash* tidak dapat diolah sedemikian rupa agar kembali ke bentuk semulanya. [5] Selain itu, fungsi *hash* juga memiliki beberapa sifat berikut:

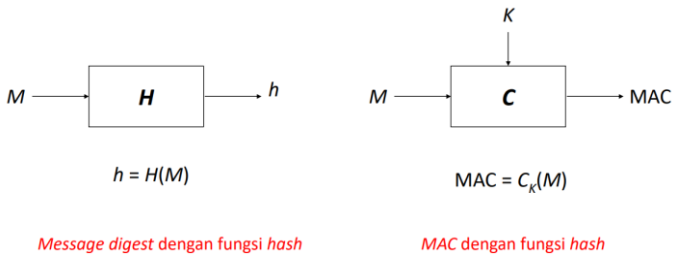
- *Collision Resistance*,
  Sulit ditemukan adanya *input* A dan B sedemikian rupa sehingga $H(A) = H(B)$.
- *Preimage Resistance*
  Sangat sulit/*infeasible* untuk melakukan inversi dari hasil keluaran agar didapatkan input awalnya.
- *Second Preimage Resistance*
  Untuk sembarang *input* A, sangat sulit untuk menemukan suatu *input* B sehingga $H(A) = H(B)$.

Penerapan fungsi memiliki banyak keuntungan, di antaranya:

- Menjaga *data integrity*
  Fungsi *hash* sangat peka terhadap *input* yang dimasukkan. Akibatnya perubahan satu bit saja pada *input* akan menghasilkan keluaran yang jauh berbeda. Dengan menambahkan nilai *hash* dari suatu data/pesan/informasi lainnya, dapat dilakukan verifikasi terhadap isinya dengan cara membandingkan nilai *hash* tersebut dengan nilai *hash* hasil komputasi. Jika nilainya sama, maka dapat dipastikan bahwa data tersebut isinya memang benar dan tidak terjadi perubahan.
- Memudahkan verifikasi
  Misalkan banyak pihak yang saling berbagi pesan, kemudian satu pihak melakukan *broadcast* suatu pesan. Untuk meyakinkan pihak-pihak penerima pesan menerima pesan yang benar, daripada membandingkan secara langsung isinya dengan pihak yang melakukan *broadcast*, dapat dibandingkan saja langsung nilai *hash* dari pesan yang asli dengan pesan yang diterima.
- Normalisasi panjang data
  Misalkan suatu informasi seperti *password* dari pengguna yang berbeda-beda. Sangat dimungkinkan informasi memiliki panjang yang berbeda. Hal tersebut akan merepotkan pada proses penyimpanan. Karena hasil keluaran fungsi *hash* memiliki panjang yang sama dan sudah kehilangan informasi dari *input*-nya, maka fungsi *hash* sangat cocok digunakan sebelum informasi disimpan pada basis data. Saat ingin melakukan pengecekan apakah suatu *password* benar, dapat dihitung nilai *hash*-nya dan dibandingkan dengan nilai *hash* yang tersimpan pada basis data.

C. Message Authentication Code

MAC adalah suatu kode berupa *string arbritrary* yang dikirimkan bersamaan dengan suatu pesan. Bisa dikatakan bahwa MAC adalah nilai *hash* dari suatu pesan. Namun perbedaannya dengan nilai *hash* biasa terletak pada fungsi *hash* yang digunakan. MAC dibuat dari fungsi *hash* dengan tipe *keyed hash function*. Oleh karena itu dibutuhkan suatu *secret key* agar algoritmanya dapat bekerja. [6] MAC sering digunakan pada kasus ketika terdapat dua atau lebih pihak yang ingin berkomunikasi lewat media yang tidak aman.



Gambar 2. Perbedaan fungsi *hash* dan MAC
Sumber: [7]

Karena MAC memerlukan adanya *secret key*, maka tidak sembarang pihak dapat membuat nilai *hash* yang benar. Hal tersebut merupakan perbedaan utama MAC dengan fungsi *hash* biasa. Dengan demikian, MAC dapat menyelesaikan kelemahan yang ada pada fungsi *hash* biasa. Misalkan dilakukan pertukaran pesan beserta nilai *hash*-nya menggunakan fungsi *hash* biasa. Seorang *attacker* tetap dapat mengubah pesan dengan cara menghitung ulang nilai *hash* dari pesan yang sudah diubah dan

mengirimkan hasil perubahannya ke penerima pesan. Akibatnya, penerima pesan akan merasa seolah pesan tersebut memang benar dari hasil verifikasi nilai *hash*-nya. Namun pada MAC, hal tersebut tidak dapat dilakukan karena adanya *secret key*. *Secret key* hanya dimiliki oleh pihak-pihak yang saling percaya saja. Jadi, dengan selain keuntungan yang didapatkan dari penerapan fungsi *hash*, MAC juga menawarkan adanya otentikasi.

[7] Selain menggunakan fungsi *hash*, MAC juga dapat diimplementasikan mengggunakan algoritma *block cipher* seperti AES, DES, atau bahkan *block cipher* buatan sendiri. Caranya adalah dengan menggunakan hanya sebagian hasil enkripsi saja. Misalkan suatu algoritma *block cipher* memiliki panjang *block* sebesar x bit. Untuk mendapatkan nilai MAC dengan panjang x bit, dapat dilakukan enkripsi terhadap seluruh pesan dan diambil hasil enkripsi pada *block* terakhir saja.

MAC biasanya didapatkan dengan menerapkan menggunakan persamaan berikut:

$$MAC_K(m) = H(m + K),$$

Dengan H adalah suatu fungsi *hash* tertentu (MD5, SHA-256, SHA-512, dan lain-lain), m adalah isi pesan asli, s adalah secret key yang digunakan, dan $+$ merupakan operasi *append*.

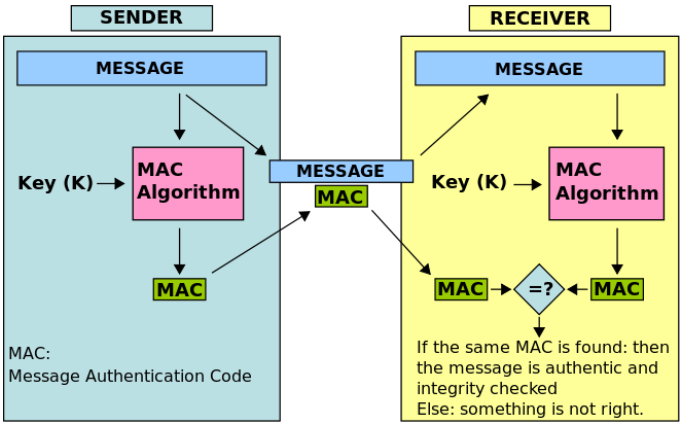
Namun setelah dilakukan penelitian lebih lanjut, ditemukan bahwa MAC memiliki kelemahan terhadap *length extension attack* [8]. Pada intinya, jika diketahui suatu MAC dari pesan m dan panjang pesan m, maka *attacker* dapat menghitung MAC dari pesan lain m' tanpa perlu mengetahui isi dari m sama sekali.

D. Keyed-Hash Message Authentication Code

Untuk menangani *vulnerabilities* MAC terhadap *length extension attack*, diperlukan suatu algoritma yang lebih kompleks lagi untuk menghindari serangan tersebut. Oleh karena itu, dibuatlah suatu mekanisme untuk membuat MAC yang lebih aman, salah satunya adalah HMAC.

Salah satu kelebihan HMAC adalah fungsi *hash* yang digunakan bersifat *black box* sehingga dapat diubah-ubah sekiranya dibutuhkan dengan mudah. Namun, kelebihan utama dari HMAC terletak pada *cryptographic strengths* yang ditawarkan. Pada intinya, dapat dibuktikan bahwa jika hasil HMAC ternyata tidak aman, dapat disimpulkan bahwa bagian yang salah terletak pada fungsi *hash* yang digunakan, bukan pada mekanisme pembuatan MAC-nya [9].

Baik MAC maupun HMAC akan melindungi pesan dari serangan *unforgeable under chosen-message attacks* (UF-CMA). Artinya adalah misalkan Alice dan Bob saling bertukar pesan menggunakan MAC dan keduanya sudah memiliki *secret key* yang sama. Jika Eve menyadap pesan yang dipertukarkan Alice dan Bob, ia tidak akan dapat membuat suatu MAC meskipun ia dapat meyakinkan salah satu dari Alice maupun Bob untuk mengirimkan suatu pesan spesifik. HMAC memiliki tambahan fitur keamanan yang lebih kuat lagi yaitu *pseudo-random function* (PRF). Hal ini artinya jika Eve tidak memiliki *secret key* yang dimiliki Alice dan Bob, semua pesan yang ia sadap akan memiliki MAC yang benar-benar *random* meskipun ia mengetahui isi pesannya (karena mungkin saja MAC selalu memiliki awalan misalnya berupa angka 0).



Gambar 3. Mekanisme penggunaan MAC
Sumber:
https://upload.wikimedia.org/wikipedia/commons/thumb/0/08/MAC.svg/661px-MAC.svg.png

Menurut dokumen RFC 2104, terdapat empat tujuan utama digunakannya HMAC yaitu:

- Agar dapat menggunakan fungsi *hash* apapun yang tersedia tanpa melakukan modifikasi.
- Untuk menjaga kinerja asli dari fungsi *hash* tanpa menimbulkan banyak degradasi yang signifikan.
- Untuk mendapatkan pemahaman *cryptographic analysis* yang jelas mengenai kekuatan mekanisme otentikasi berdasarkan asumsi pada fungsi *hash* yang digunakan.
- Memungkinkan kemudahan penggantian fungsi *hash* jika dibutuhkan atau ditemukan fungsi yang lebih aman nantinya.

Dengan demikian, untuk mewujudkan tujuan tersebut, maka mekanisme dari HMAC didefinisikan sebagai berikut:

- Terdapat fungsi *hash* H, *secret key* K. H adalah fungsi hash yang melakukan iterasi terhadap *block* data dengan panjang B *byte* dan memiliki keluaran dengan panjang L *byte*.
- Panjang $L \leq B$
- Didefinisikan dua konstanta yaitu $ipad$ (*inner padding*) dan $opad$ (*outer padding*) dengan rumus:
  $ipad = byte\ 0x36$ diulang sebanyak B kali
  $opad = byte\ 0x5c$ diulang sebanyak B kali
- Untuk menghitung HMAC dari suatu pesan m digunakan rumus berikut:
  $$H(K \oplus opad, H(K \oplus ipad, m)$$

[10] Langkah rinci untuk pembuatan HMAC adalah sebagai berikut:

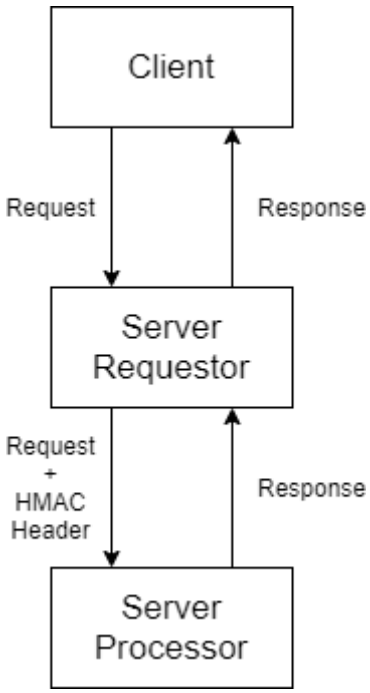
(1) Append bit 0 ke K hingga panjangnya B.
(2) Lakukan XOR hasil (1) dengan $ipad$.
(3) Append pesan m ke hasil yang didapat dari (2).
(4) Lakukan fungsi hash H terhadap hasil (3).
(5) Lakukan XOR hasil (1) dengan $opad$.
(6) Append hasil (4) ke hasil (5).
(7) Lakukan fungsi hash H terhadap hasil (6).

Jadi hal utama yang membedakan HMAC adalah adanya dua ronde fungsi hash yang dilakukan terhadap pesan asli.

III. IMPLEMENTASI

A. Flow Komunikasi

Dalam implementasi HMAC untuk komunikasi antar *server*, dibuat sebuah jaringan sederhana seperti yang terlihat pada Gambar 4.



Gambar 4. Arsitektur jaringan yang diimplementasikan
Sumber: Dokumen penulis

Arsitektur ini dibuat untuk merepresentasikan kondisi yang sering terjadi di praktek dunia nyata. Dalam hal ini, *server requestor* dapat disebut juga sebagai API *Gateway* yang berguna untuk menyaring semua *request* yang dikirimkan client. Arsitektur ini tampak sederhana namun cukup menggambarkan keadaan yang aslinya. Pada prakteknya, biasanya terdapat lebih dari satu *server processor* dan mungkin terjadi komunikasi antar satu dengan yang lain.

Flow komunikasi yang terjadi adalah pertama *client* akan mengirimkan *request* ke *server requestor*. Kemudian, *server requestor* akan mengecek apakah *client* terotentikasi atau tidak menggunakan mekanisme tertentu (bagian ini tidak diimplementasikan karena pada makalah ini hanya fokus di otentikasi antar *server*-nya). Jika *client* terotentikasi, maka *server requestor* akan menambahkan HMAC pada *header request* berdasarkan *request body* dan meneruskan *request* tersebut ke *server processor* yang sesuai. *Server processor* akan melakukan verifikasi nilai HMAC dengan menghitung sendiri nilai tersebut dan membandingkannya dengan nilai HMAC yang ia terima di *header request*. Hal ini dapat dilakukan karena *server requestor* dan *server processor* telah memiliki *shared secret key*. Jika nilainya sama, maka *server processor* dapat yakin bahwa pengirim request memang berasal dari *server requestor* yang ia kenal dan bukan hasil pemalsuan pihak lain. Terakhir, *server processor* akan mengirimkan *response* yang sesuai ke *server requestor* dan *server requestor* melanjutkan *response*-nya ke *client*.

Dengan arsitektur seperti ini, semua *request* yang berasal dari *client* harus dilewatkan *server requestor* terlebih dahulu baru dapat dianggap valid sehingga akan memudahkan melakukan *control flow* dari semua *request* yang ada. Jika *client* mencoba melakukan *request* secara langsung ke *server processor*, ia tidak dapat menambahkan HMAC sendiri dan *server processor* akan membalas dengan *response unauthorized*.

Jadi, dengan adanya HMAC akan membuat semua *server* yang saling berkomunikasi yakin bahwa pihak yang diajak berkomunikasi itu memang *server* lain yang sudah terpercaya karena memiliki *shared secret key*.

B. HMAC Header

HMAC akan didapatkan dengan melakukan kalkulasi terhadap *request body* dan menggunakan *secret key* tertentu. Setelah nilai HMAC didapatkan, hasilnya akan dimasukkan ke *header authentication* pada *request*. Berikut adalah potongan kode dalam bahasa Javascript untuk melakukan perhitungan HMAC yang digunakan pada implementasi:

```
calculateHMAC = (text, secretKey) => {
    const asciiText = text.split('').map(
        char => char.charCodeAt(0)
    );

    let asciiKey = secretKey.split('').map(
        char => char.charCodeAt(0)
    );
    asciiKey = [...asciiKey, ...Array(64 -
asciiKey.length)
        .fill(0)];

    const ipadKey = asciiKey.map(x => x ^ 0x36);
    const innerInput = ipadKey.concat(asciiText);
    const innerHash = crypto.createHash('sha256')
        .update(Uint8Array.from(innerInput))
        .digest('hex').split('');


    const opadKey = asciiKey.map(x => x ^ 0x5C);
    const outerInput = opadKey.concat(innerHash);
    const outerHash = crypto.createHash('sha256')
        .update(Uint8Array.from(outerInput))
        .digest('hex');

    return outerHash;
}
```

C. Lingkungan Implementasi

Implementasi dilakukan secara lokal pada satu mesin untuk mengurangi *overhead* pengiriman data pada saat melakukan eksperimen. Adapun spesifikasi mesin tersebut adalah:

- Processor: AMD Ryzen 7 4800U
- RAM: 16GB
- OS: Ubuntu 20.04

Program dibuat menggunakan bahasa Javascript dengan *library* ExpressJS. Jadi akan ada dua *server* yang berjalan pada *port* yang berbeda pada mesin yang sama. Kedua *server* ini saling berkomunikasi sebagaimana yang digambarkan pada Gambar 4.

IV. EKSPERIMEN DAN ANALISIS

A. Eksperimen

Pada eksperimen untuk menguji otentikasi menggunakan HMAC, digunakan fungsi *hash* yaitu SHA-256 yang memiliki *block size* sebesar 64 *byte*. *Secret key* yang digunakan oleh *server* yang berkomunikasi adalah `VeRYs3cr3tK318472!!`. Semua *server* dijalankan di mesin yang sama dan saling berkomunikasi menggunakan protokol HTTP. C*lient* akan melakukan HTTP *request* ke *server requestor* dengan menggunakan *command* `curl` dari *terminal* dengan *method* POST.

Tabel 1 menunjukkan beberapa contoh *header* dari *request* yang dikirimkan oleh *server requestor* sebelum dan sesudah ditambahkan HMAC *header*.

Tabel 1. Contoh *request header* sebelum dan sesudah ditambahkan HMAC

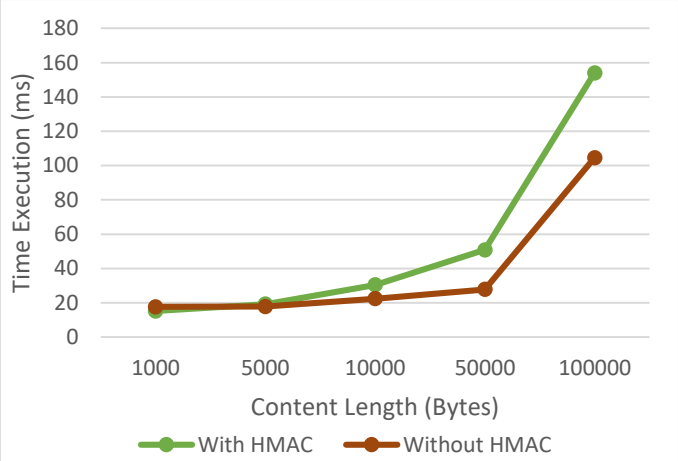
Body	*Header* Sebelum	*Header* Sesudah
{ “content”:”hello” }	{ 'user-agent': 'curl/7.68.0', accept: '*/*', 'content-type': 'application/json', connection: 'close' }	{ 'user-agent': 'curl/7.68.0', accept: '*/*', 'content-type': 'application/json', connection: 'close', **Authentication: '7c5de2f0ec4cb84c7a0e75754c99032ffa0b6b25630bc5b1f925d201593eb91b'** }
{ “content”:”tugas makalah uas kriptografi 2020” }	{ 'user-agent': 'curl/7.68.0', accept: '*/*', 'content-type': 'application/json', connection: 'close' }	{ 'user-agent': 'curl/7.68.0', accept: '*/*', 'content-type': 'application/json', connection: 'close', **Authentication: 'fbeea16574a29c4b4096f7d28e552086020744c3da51c8eb51937b09844735d9'** }

Untuk menguji apakah proses otentikasi berhasil atau tidak, dilakukan pengujian dengan beberapa kondisi yaitu dengan mengubah *secret key* pada salah satu *server*, mencoba melakukan *request* langsung ke *server processor* dari *client*, dan mengubah *body* dari *request* di *server requestor* ke *server processor*. Pada pengujian ini digunakan *request body* yang sama yaitu `{“content”:”test otentikasi”}`.

Tabel 2. Hasil pengujian otentikasi

Kasus Uji	Kondisi	Response
1	Mengikuti *flow* yang benar dan sesuai	200 OK
2	Melakukan *request* langsung ke *server processor* dari *client*	401 Unauthorized
3	Mengubah *secret key* pada *server requestor* dari `VeRYs3cr3tK318472!!` menjadi `VeRYs3cr3tK31847`**`1`**`!!`	401 Unauthorized
4	Mengubah pesan yang dikirimkan dari *server requestor* dari `{“content”:”test otentikasi”}` menjadi `{“content”:”test otentikasi`**`!`**`”}`	401 Unauthorized

Untuk menguji kinerja dari penggunaan HMAC, dilakukan perbandingan lama waktu eksekusi pemrosesan dari suatu *request* jika menggunakan HMAC dan jika tidak menggunakan HMAC pada berbagai panjang pesan. Waktu pemrosesan dihitung dari sejak *client* melakukan *request* hingga mendapatkan *response* yang sesuai. Pada pengujian ini, semua *request* dilakukan sesuai *flow* yang benar agar semua *server* mengeksekusi perhitungan HMAC dan mengembalikan *response* 200. Untuk setiap pesan dengan panjang tertentu, *request* diulang sebanyak lima kali dan dilakukan rata-rata waktu pemrosesannya. Hasilnya tampak pada Gambar 5.



Gambar 5. Perbandingan waktu pemrosesan *request* dengan dan tanpa HMAC
Sumber: Dokumen penulis

B. Analisis

Berdasarkan eksperimen yang telah dilakukan, semua *request header* yang dikirimkan ke *server requestor* berhasil ditambahkan HMAC *header* berupa *string* dengan panjang 64 *byte* seperti yang terlihat pada Tabel 1. *Request* yang diteruskan *server requestor* ke *server processor* kemudian akan diverifikasi apakah memiliki nilai HMAC yang valid berdasarkan *request body*-nya.

Dari Tabel 2 terlihat bahwa *server* hanya akan mengembalikan *response* 200 apabila *flow request* benar (HMAC bernilai valid).

Pada kasus uji-2, dilakukan pengujian otentikasi *client*. Jika *client* mencoba untuk langsung melakukan *request* ke *server processor*, maka *request* tersebut tidak mungkin dapat mengandung nilai HMAC yang valid, akibatnya *server* akan mengembalikan *response* 401.

Pada kasus uji-3, dilakukan pengujian otentikasi *server*. Hanya *server-server* yang memiliki *secret key* yang sama saja yang dapat melakukan komunikasi yang terotentikasi. Hal ini terbukti bahwa hanya dengan mengubah satu karakter saja *secret key* dari salah satu *server*, nilai HMAC langsung menjadi tidak valid dan *server processor* mengembalikan *response* 401.

Pada kasus uji-4, dilakukan pengujian terhadap *Man in The Middle Attack* (MITM). Dilakukan perubahan satu karakter *request body* pada *request* yang dikirim dari *server requestor* ke *server processor*. Akibatnya ketika dilakukan verifikasi, nilai HMAC yang ada tidak sesuai dengan *request body* yang diterima. Sesuai dugaan, *server processor* kemudian akan mengembalikan *response* 401.

Selain dilakukan pengujian otentikasi, dilakukan pengujian kinerja HMAC. Seperti yang terlihat pada Gambar 5, menggunakan HMAC tentu saja memperlama waktu pemrosesan suatu *request*. Perlu diperhatikan bahwa waktu eksekusi yang ditunjukkan dihitung dari mulai *client* melakukan *request* dan mendapatkan *response*-nya. Hal ini berarti sebenarnya dilakukan kalkulasi HMAC dua kali yaitu pada saat *server requestor* menambahkan HMAC *header* dan pada saat *server processor* memverifikasi HMAC.

Selisih waktu antara menggunakan HMAC dan tidak menggunakan berbanding lurus terhadap panjang pesan yang dikirimkan. Hal tersebut baru tampak pada *request* yang memiliki panjang pesan ≥ 50 KB karena pada *request* yang berukuran < 50 KB waktu kalkulasi HMAC jauh lebih cepat dibandingkan *overhead* lain yang pasti selalu dilakukan pada saat melakukan *request*. Pada *request* dengan panjang pesan 50 KB selisih waktu sebesar 23 ms, sedangkan pada *request* dengan panjang pesan 100 KB selisih waktu sebesar 49,4 ms (dua kali lipat sesuai dengan perbandingan panjang pesannya). Dari hasil eksperimen, tampak bahwa penggunakan HMAC memiliki dampak yang cukup signifikan pada waktu pemrosesannya jika *request* memiliki panjang yang besar. Namun, pada *request* dengan panjang pesan kecil (< 10 KB) selisih waktu ini dapat diabaikan.

Sebagai alternatif, proses kalkulasi HMAC dapat dilakukan bukan berdasarkan *request body* namun berdasarkan informasi trivial lain yang selalu ada pada saat melakukan request, misalnya *timestamp request*. Dengan demikian kalkulasi HMAC dapat dilakukan dengan cepat tanpa berdampak signifikan terhadap terhadap pemrosesan *request* dan otentikasi tetap dapat dilakukan. Namun alternatif ini tidak menjamin *data integrity* dari pesan yang dikirimkan.

Jadi terbukti bahwa dengan menggunakan HMAC dapat dilakukan otentikasi pada komunikasi antar *server* dengan cara berbagi *secret key*. Penggunaan HMAC juga dapat memberikan keamanan *data integrity* pada saat berbagi pesan. Selain itu, dengan menggunakan HMAC sebagai otentikasi antar *server*, *flow request* menjadi lebih mudah di-*control* karena hanya *server* yang terotentikasi saja yang dapat saling berkomunikasi.

V. KESIMPULAN

HMAC dapat digunakan untuk melakukan otentikasi komunikasi antar *server*. Dengan berbagi *secret key*, *server* yang saling berkomunikasi dapat menjamin bahwa pesan yang diterima berasal dari *server* yang terpercaya. Dengan melakukan modifikasi, HMAC juga dapat digunakan untuk menjamin *data integrity* pesan.

VI. UCAPAN TERIMA KASIH

Ucapan terima kasih penulis sampaikan kepada Tuhan yang Maha Esa karena berkat dan rahmatnya, penulis dapat menyelesaikan tugas makalah Kriptografi ini. Penulis juga ingin mengucapkan banyak terima kasih pada Dr. Ir. Rinaldi Munir, MT. sebagai dosen penulis karena sudah mengajarkan sangat banyak hal mengenai kriptografi, masalah keamanan yang ada, serta mekanisme penyelesaiannya. selama satu semester ini. Terakhir, penulis juga berterima kasih kepada semua teman dan keluarga yang membantu penulis menyelesaikan makalah ini.

REFERENSI

[1] "Security issue, HMAC in header vs https, or both?" https://stackoverflow.com/questions/22683952/security-issue-hmac-in-header-vs-https-or-both/ (accessed Dec. 19, 2020).

[2] "HTTP vs HTTPS performance." https://stackoverflow.com/questions/149274/http-vs-https-performance (accessed Dec. 19, 2020).

[3] N. A. Lal, S. Prasad, and M. Farik, "A Review Of Authentication Methods," *Int. J. Sci. Technol. Res.*, vol. 4, no. 8, pp. 246–249, 2015.

[4] R. Sobti and G. Geetha, "Cryptographic Hash functions - a review," *IJCSI Int. J. Comput. Sci. Issues*, vol. 9, no. 2, pp. 461–479, 2012.

[5] R. Munir, "Fungsi Hash," 2020. https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2020-2021/Fungsi-hash-2020.pdf (accessed Dec. 19, 2020).

[6] H. Krawczyk, "Keying Hash Functions for Message Authentication," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1995. .

[7] R. Munir, "Message - Authentication Code Algorithms," *Cryptography for Developers*, 2020. https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2020-2021/MAC-2020.pdf (accessed Dec. 19, 2020).

[8] Y. Sasaki, "Cryptanalyses on a Merkle-Damg ˚ ard Based MAC — Almost Universal Forgery and Distinguishing- H Attacks."

[9] M. Bellare, R. Canetti, and H. Krawczyk, "Message authentication using hash functions—the HMAC construction," *RSA Lab. CryptoBytes*, vol. 2, no. 1, pp. 1–5, 1996, [Online]. Available: http://charlotte.ucsd.edu/~mihir/papers/hmac-cb.pdf.

[10] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," 1997. https://cseweb.ucsd.edu/~mihir/papers/rfc2104.txt.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Klaten, 20 Desember 2020

Steve Andreas Immanuel 13517039