Evaluasi Kinerja dan Keamanan Penggunaan Algoritma Kriptografi HMAC SHA-3 dan RSA dalam Implementasi JSON Web Token (JWT)

Gibran Fasha Ghazanfar (18221069)
Program Studi Sistem dan Teknologi Informasi
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: gibran.fsh@gmail.com

***Abstrak*—JSON Web Token (JWT) adalah metode populer untuk memastikan pertukaran data yang aman dalam lingkungan terdistribusi. Dalam dunia yang semakin terhubung, keamanan dan kinerja data menjadi prioritas utama, dan JWT menawarkan solusi yang efisien dan terukur. Penelitian ini menganalisis penggunaan dua algoritma kriptografi yang berbeda dalam pembuatan dan verifikasi JWT, yaitu menggunakan algoritma RSA dan HMAC SHA-3. RSA dengan panjang kunci 2048-bit dan HMAC SHA-3 dengan panjang kunci 256-bit dipilih untuk mengukur kinerja dalam proses pembuatan dan verifikasi token, serta mengevaluasi kemampuan sistem dalam menghadapi berbagai jenis serangan. Hasil pengujian menunjukkan bahwa kedua algoritma memiliki keunggulan dan kelemahan masing-masing, dengan HMAC SHA-3 memberikan kinerja yang lebih baik dalam hal kecepatan, sementara RSA menunjukkan keandalan yang lebih tinggi dalam memastikan integritas data. Penelitian ini memberikan wawasan yang berharga bagi pengembang dalam memilih algoritma kriptografi yang sesuai dengan kebutuhan keamanan dan kinerja aplikasi mereka. Penelitian lebih lanjut disarankan untuk menjelajahi implikasi keamanan dan kinerja dari menggunakan kunci dengan panjang yang berbeda, serta untuk menguji algoritma kriptografi lainnya yang mungkin menjadi alternatif yang baik untuk penggunaan JWT.**

Kata Kunci*—*JSON Web Token (JWT), algoritma kriptografi, RSA, HMAC SHA-3, keamanan data, kinerja sistem, integritas data, pertukaran data terdistribusi

I. Pendahuluan

Dalam era digital saat ini, keamanan informasi menjadi salah satu perhatian utama dalam pengembangan dan implementasi sistem berbasis web. Seiring dengan meningkatnya penggunaan aplikasi web untuk berbagai kebutuhan, dari layanan keuangan hingga media sosial, kebutuhan akan mekanisme autentikasi yang andal dan efisien semakin mendesak. Keamanan autentikasi tidak hanya memastikan bahwa data pengguna terlindungi, tetapi juga menjaga integritas dan kepercayaan terhadap layanan yang diberikan oleh penyedia aplikasi.

Keamanan data dalam aplikasi web adalah faktor yang sangat krusial mengingat banyaknya informasi sensitif yang dipertukarkan secara *online*. Dengan semakin meningkatnya pengguna aplikasi web, tantangan untuk menjaga keamanan data juga semakin kompleks. Berbagai ancaman keamanan, seperti *man-in-the-middle attack*, *phishing*, dan pemalsuan token, membuat pengembang aplikasi harus menerapkan mekanisme autentikasi yang kuat dan dapat diandalkan. Dalam konteks ini, memilih algoritma kriptografi yang tepat untuk autentikasi menjadi sangat penting untuk memastikan bahwa sistem dapat melindungi data pengguna dengan efektif.

Salah satu mekanisme autentikasi yang banyak digunakan adalah JSON Web Token (JWT). JWT adalah standar terbuka yang menyediakan cara ringkas dan *self-contained* untuk mengirimkan informasi antara pihak-pihak sebagai objek JSON. JWT memungkinkan data yang dikirimkan untuk ditandatangani secara digital menggunakan algoritma kriptografi sehingga integritas dan autentikasi data dapat dipastikan. Dua algoritma kriptografi yang umum digunakan dalam implementasi JWT adalah HMAC (Hash-based Message Authentication Code) dengan SHA-3 dan RSA (Rivest-Shamir-Adleman).

Dalam implementasi JWT, pemilihan antara HMAC SHA-3 dan RSA dapat berdampak signifikan pada kinerja dan tingkat keamanan sistem. HMAC SHA-3 umumnya lebih cepat dalam proses enkripsi dan dekripsi dibandingkan RSA karena tidak memerlukan operasi matematis yang kompleks, tetapi memerlukan manajemen kunci yang lebih ketat. Sebaliknya, RSA, meskipun lebih lambat, menawarkan keuntungan dalam manajemen kunci karena hanya memerlukan distribusi kunci publik untuk verifikasi tanda tangan digital.

Oleh karena itu, akan dilakukan evaluasi kinerja algoritma HMAC SHA-3 dan RSA dalam proses pembuatan dan verifikasi JWT, analisis tingkat keamanan yang ditawarkan oleh algoritma HMAC SHA-3 dan RSA dalam menjaga integritas dan autentikasi JWT, serta membandingkan hasil evaluasi kinerja dan keamanan kedua algoritma tersebut untuk memberikan rekomendasi penggunaan yang optimal dalam konteks yang berbeda. Dengan memahami kelebihan dan kelemahan masing-masing algoritma dalam konteks JWT, pengembang aplikasi dapat membuat keputusan yang lebih baik terkait pilihan algoritma kriptografi yang digunakan

untuk memastikan keamanan dan efisiensi sistem. Evaluasi ini diharapkan dapat memberikan panduan yang komprehensif bagi pengembang dalam memilih algoritma kriptografi yang paling sesuai untuk kebutuhan autentikasi aplikasi web mereka sehingga dapat meningkatkan kepercayaan pengguna dan melindungi data sensitif dari berbagai ancaman keamanan.

II. Metodologi Penelitian

A. *Literature Review*

Langkah pertama dalam pembuatan makalah ini adalah melakukan *literature review* dengan mencari informasi dari sumber digital berupa jurnal, *paper*, dan materi perkuliahan mengenai algoritma RSA, fungsi hash, SHA-3 (Keccak), HMAC (Hash-based Message Authentication Code), Autentikasi Berbasis Token, dan JSON Web Token (JWT), serta penerapannya pada sistem keamanan dan autentikasi dalam aplikasi web. Pencarian dilakukan melalui *database* seperti IEEE Xplore, SpringerLink, dan Google Scholar menggunakan kata kunci terkait. Informasi dari jurnal, studi kasus, modul kuliah, dan dokumentasi teknis dari organisasi standar seperti NIST dan IETF digunakan untuk memahami mekanisme kerja, implementasi, serta isu-isu keamanan dan performa terkait algoritma tersebut. Hasil dari *literature review* ini menjadi dasar teori dan referensi utama dalam penelitian ini, yang memungkinkan penulis merancang eksperimen yang tepat dan menyusun rekomendasi untuk implementasi optimal algoritma kriptografi dalam konteks JWT.

B. *Experiment*

Bagian eksperimen dari penelitian ini melibatkan beberapa langkah kunci untuk mengevaluasi kinerja dan keamanan algoritma HMAC SHA-3 dan RSA dalam implementasi JSON Web Token (JWT). Berikut adalah tahapan eksperimen yang dilakukan :

1. Persiapan *Experiment Environment*

   Menyiapkan lingkungan pengujian dengan menginstal Go (Go Lang) dan library yang diperlukan seperti `github.com/golang-jwt/jwt/v4` untuk JWT, `crypto` untuk algoritma kriptografi, dan pembuatan beberapa fungsi untuk pengujian kinerja.

2. Implementasi Algoritma

   Mengimplementasikan HMAC SHA-3 dan RSA untuk pembuatan dan verifikasi JWT mencakup penulisan kode Go untuk :

   a) Menghasilkan dan menandatangani token JWT menggunakan HMAC SHA-3
   b) Menghasilkan dan menandatangani token JWT menggunakan RSA
   c) Memverifikasi token JWT dengan kedua algoritma tersebut

3. Pengujian Kinerja dan Keamanan

   Pengujian kinerja dan keamanan dilakukan dengan cara sebagai berikut.

   a) *Benchmarking Test*

      Pengujian ini digunakan untuk mengukur kinerja pembuatan dan verifikasi token JWT menggunakan algoritma RSA dan HMAC SHA-3. Hal ini membantu mengevaluasi efisiensi dan kecepatan operasi kunci dalam sistem.

   b) Algorithm-Confusion Test

      Pengujian ini bertujuan untuk memeriksa apakah sistem dapat mendeteksi dan menolak token yang dibuat menggunakan algoritma yang salah. Hal ini membantu memastikan bahwa mekanisme verifikasi token bekerja secara efektif dalam mengenali algoritma yang benar.

   c) Integrity Checking Test

      Pengujian ini melibatkan mencoba memanipulasi token yang dihasilkan, seperti mengubah *payload*, dan memastikan bahwa sistem dapat mendeteksi perubahan yang tidak sah pada token tersebut.

   d) Payload Size Impact Test

      Pengujian ini mengukur kinerja pembuatan dan verifikasi token JWT dengan ukuran *payload* yang berbeda-beda. Hal ini membantu mengevaluasi dampak ukuran *payload* terhadap kinerja sistem.

   e) Stress Testing

      Pengujian ini dilakukan untuk menguji kinerja sistem dalam kondisi beban kerja yang tinggi, dengan membuat dan memverifikasi sejumlah besar token JWT secara bersamaan. Hal ini membantu mengevaluasi keandalan sistem dalam menangani beban kerja yang tinggi dan melihat apakah sistem tetap berjalan secara stabil dalam situasi tersebut.

4. Analisis dan Perbandingan

   a) Menganalisis data yang diperoleh dari pengujian kinerja dan keamanan untuk menilai efektivitas masing-masing algoritma
   b) Membandingkan hasil pengujian untuk menentukan kelebihan dan kelemahan HMAC SHA-3 dan RSA dalam konteks penggunaan JWT

III. Landasan Teori

A. Algoritma RSA (Rivest-Shamir-Adleman)

Algoritma RSA (Rivest-Shamir Adleman) adalah algoritma yang dibuat oleh Ronald Rivest, Adi Shamir, dan Leonard Adleman, pada tahun 1976 yang merupakan peneliti dari MIT. Algoritma RSA ini merupakan algoritma kunci-publik yang saat ini paling terkenal dan paling banyak aplikasinya. Keamanan algoritma RSA terletak pada sulitnya

memfaktorkan bilangan bulat yang besar menjadi faktor-faktor prima [1].

Properti-properti dari algoritma RSA adalah sebagai berikut :

1. p dan q adalah bilangan prima (Rahasia)
2. $n = p \cdot q$ (Tidak Rahasia)
3. $\phi(n) = (p-1)(q-1)$ (Rahasia)
4. e, Kunci Enkripsi (Tidak Rahasia), dengan syarat PBB$(e, \phi(n)) = 1$, PBB = Pembagi Bersama Terbesar, atau bisa disebut sebagai *gcd*.
5. d, Kunci Dekripsi (Rahasia), yang mana d itu dihitung dari $d \equiv e^{-1} mod(\phi(n))$ dan harus memenuhi persamaan $ed \equiv mod\,(\phi(n))$
6. m, plainteks (Rahasia)
7. c, cipherteks (Tidak Rahasia)

Dari properti-properti tersebut, pasangan kunci privatnya adalah (d, n) dan pasangan kunci publiknya adalah (e, n). Proses enkripsi dapat dilakukan dengan membagi menjadi blok-blok yang lebih kecil jika memang memungkinkan. Kemudian, cipherteks, c, dihitung dengan plainteks, m, menggunakan kunci publik, e, dengan persamaan sebagai berikut :

$$c = m^e \, mod \, n$$

Sementara itu, proses dekripsi dilakukan dengan menghitung blok plainteks, m, dengan blok chiperteks, c, dengan menggunakan kunci privat , d, dengan persamaan berikut :

$$m = c^d \, mod \, n$$

B. Fungsi Hash

Fungsi hash dalam kriptografi adalah algoritma matematis yang digunakan untuk mengubah data dengan ukuran berapa pun (*message*) menjadi sebuah *array* dengan ukuran tetap, yang sering disebut sebagai *hash value* atau *message digest* [2].

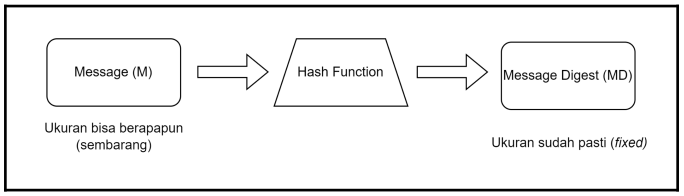

Fig. 1 Cara Fungsi *Hash* Bekerja (*high level*)

Sumber : Dokumentasi Penulis

Fungsi *hash* memiliki sifat *irreversible*, artinya nilai *hash* yang dihasilkan tidak dapat dikembalikan ke bentuk data asli. Oleh karena itu, fungsi *hash* sering disebut sebagai fungsi satu arah (*one-way function*). Idealnya, setiap *message* yang berbeda akan menghasilkan *message digest* yang unik sehingga berbagai jenis fungsi *hash* digunakan secara luas dalam implementasi keamanan kriptografi modern.

Fungsi *hash* digunakan dalam berbagai aplikasi, seperti memverifikasi integritas dokumen atau *file*, membuat tanda tangan digital, verifikasi kata sandi, dan pengecekan data *checksum*. Untuk mengimplementasikan fungsi *hash*, tersedia berbagai algoritma seperti MD5, SHA-1, SHA-2, SHA-3, RIPEMD, dan lain-lain. Namun, tidak semua fungsi *hash* ini aman digunakan karena beberapa di antaranya telah ditemukan rentan terhadap kolisi. Kolisi terjadi ketika dua pesan berbeda menghasilkan nilai *hash* yang sama, yang tentunya dapat mengancam keamanan informasi yang dihasilkan oleh algoritma tersebut. Contoh fungsi hash yang telah terbukti memiliki kolisi adalah MD5 dan RIPEMD. Oleh karena itu, penting untuk memilih algoritma *hash* yang lebih modern dan aman seperti SHA-3 dalam implementasi yang memerlukan keamanan tinggi.

Fungsi *hash* memainkan peran penting dalam berbagai skenario keamanan, seperti memastikan bahwa dokumen tidak berubah selama transmisi, memberikan otentikasi yang kuat melalui tanda tangan digital, dan menyimpan kata sandi dengan aman dalam bentuk *hash* yang tidak dapat dibalik. Dengan perkembangan teknologi dan meningkatnya ancaman keamanan, penting untuk terus mengevaluasi dan memperbarui algoritma *hash* yang digunakan untuk memastikan bahwa sistem tetap terlindungi dari potensi serangan.

C. SHA-3

SHA-3 (Secure Hash Algorithm 3) atau dikenal juga dengan Keccak adalah salah satu varian dari fungsi hash yang melengkapi fungsi SHA-1 dan SHA-2. Algoritma ini dikembangkan oleh Guido Bertoni, Joan Daemen, Michaël Peeters, dan Gilles Van Assche, dan dirilis oleh NIST (National Institute of Standards and Technology) pada tanggal 5 Agustus 2015.

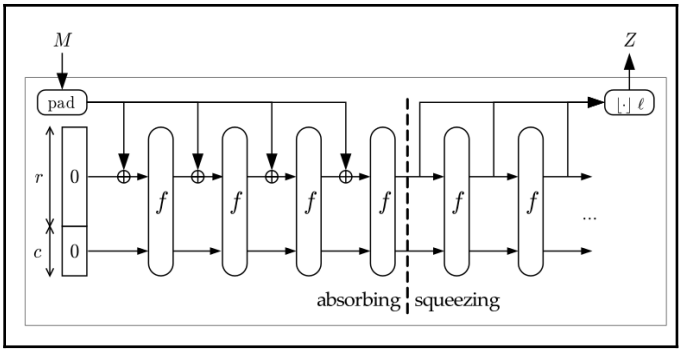

Fig. 2 Ilustrasi Mekanisme *Hash* pada SHA-3

Sumber : [3]

SHA-3 bekerja menggunakan konstruksi *sponge* yang memungkinkan penyesuaian panjang *message digest* (*output length*). Proses *hashing* dimulai dengan menambahkan *padding* pada pesan sesuai dengan panjang *output* yang diinginkan. Setelah proses *padding*, pesan dibagi menjadi blok-blok berukuran r-bit [4]. Tabel berikut memberikan contoh panduan untuk panjang *output* standar dan nilai r.

TABEL 1. TIPE-TIPE SHA-3

Tipe	***Output Length***	***Rate* (*r*)**	***Capacity* (*c*)**

SHA3-224	224	1152	448
SHA3-256	256	1088	512
SHA3-384	384	832	768
SHA3-512	512	576	1024

Sumber : [5]

Perlu dicatat bahwa jumlah nilai r dan c harus sama dengan 1600 bit. Pada awal proses, *state* awal dari blok berukuran r+c bit diinisialisasi menjadi nol. Algoritma kemudian menjalankan dua fase utama: fase penyerapan (*absorbing*) dan fase pemerasan (*squeezing*) [5]. Dalam fase penyerapan, setiap blok pesan yang masuk diproses melalui fungsi permutasi. Selanjutnya, dalam fase pemerasan, *message digest* dihasilkan sesuai dengan panjang *output* yang telah ditentukan di awal [6].

Konstruksi *sponge* memungkinkan SHA-3 untuk menawarkan fleksibilitas dan keamanan yang tinggi. Algoritma ini dirancang untuk lebih tahan terhadap serangan kriptografi modern dibandingkan dengan pendahulunya, SHA-1 dan SHA-2. Fase penyerapan mengintegrasikan data masuk dengan *state* internal, sedangkan fase pemerasan menghasilkan nilai *hash* akhir. Fleksibilitas dalam menentukan panjang *output* menjadikan SHA-3 sangat berguna dalam berbagai aplikasi keamanan, termasuk tanda tangan digital, fungsi kunci derivasi, dan *checksum* data.

Dengan memanfaatkan konstruksi *sponge* dan teknik permutasi, SHA-3 tidak hanya meningkatkan keamanan tetapi juga memberikan kinerja yang lebih baik dalam berbagai skenario aplikasi kriptografi. Penggunaan SHA-3 menjadi pilihan yang tepat dalam lingkungan yang membutuhkan tingkat keamanan tinggi dan fleksibilitas dalam pengaturan panjang *output hash*.

D. HMAC

HMAC, atau Hash-based Message Authentication Codes, merupakan salah satu jenis Message Authentication Code (MAC) yang menggunakan algoritma *hash* seperti MD5, SHA-1, dan SHA-2, bersama dengan sebuah *secret key*. HMAC dirancang untuk menyediakan autentikasi pesan dan memastikan integritas data dengan cara yang lebih efisien. Dengan menggabungkan fungsi *hash* dan *secret key*, HMAC dapat menghasilkan kode autentikasi yang unik untuk setiap pesan.

Berbeda dengan pendekatan tanda tangan digital yang memanfaatkan kriptografi kunci publik, HMAC menggunakan kriptografi kunci simetris di mana *secret key* yang sama digunakan oleh pengirim dan penerima pesan. Hal ini membuat HMAC lebih cepat dan lebih sederhana dalam implementasi, terutama dalam lingkungan di mana kunci dapat disimpan dengan aman.

Dalam prosesnya, HMAC menggabungkan pesan asli dengan *secret key*, lalu menerapkan algoritma *hash* pada kombinasi tersebut untuk menghasilkan kode autentikasi. Proses ini dilakukan dalam dua tahap utama: pertama, *secret key* dan pesan di-*hash* bersama-sama, kemudian hasil *hash* ini digabungkan lagi dengan *secret key* dan di-*hash* untuk kedua kalinya. Metode dua tahap ini meningkatkan keamanan HMAC dengan mengurangi risiko serangan terhadap fungsi *hash* yang digunakan.

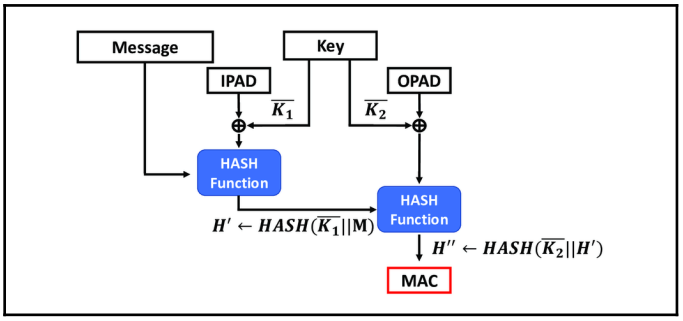

Fig. 3 Ilustrasi Mekanisme HMAC

Sumber : [7]

HMAC sangat efektif dalam berbagai aplikasi keamanan, termasuk verifikasi integritas data, autentikasi pesan dalam protokol jaringan, dan proteksi terhadap manipulasi data. Algoritma ini digunakan secara luas dalam protokol keamanan seperti TLS (Transport Layer Security), IPsec, dan berbagai layanan web yang memerlukan komunikasi yang aman.

Keunggulan HMAC termasuk ketahanannya terhadap serangan *brute force* dan analisis kriptografi, selama *secret key* dijaga kerahasiaannya. Selain itu, fleksibilitas HMAC dalam menggunakan berbagai algoritma *hash* membuatnya dapat disesuaikan dengan kebutuhan spesifik dari berbagai sistem keamanan. Penggunaan algoritma *hash* yang kuat dan kunci yang panjang dapat lebih meningkatkan keamanan HMAC dalam aplikasi praktis [8].

E. Token-based Authentication

Token-based Authentication atau autentikasi berbasis token adalah sebuah mekanisme autentikasi yang umum digunakan dalam aplikasi web, di mana sistem menghasilkan sebuah *access token* yang unik setelah identitas pengguna berhasil diverifikasi. Setiap *access token* memiliki masa aktif tertentu, selama masa tersebut, pengguna dapat mengakses halaman web yang dilindungi dengan menyertakan token tersebut tanpa perlu *login* ulang setiap kali mengakses [9].

Penggunaan token dalam metode autentikasi ini menawarkan beberapa keuntungan. Pertama, token bersifat *stateless*, yang berarti tidak ada informasi pengguna yang disimpan di *server* sehingga mengurangi risiko kebocoran data pengguna. Kedua, token memiliki masa aktif tertentu. Ketika pengguna *logout*, token tersebut dihancurkan sehingga mengurangi risiko serangan siber yang mungkin terjadi jika token tetap aktif. Ketiga, token dihasilkan secara unik menggunakan algoritma kriptografi yang aman sehingga memastikan bahwa setiap token sulit untuk dipalsukan.

Menurut [10], terdapat lima proses utama dalam autentikasi berbasis token:

1. *Request*: Pengguna melakukan *login* dengan memasukkan kredensial mereka dan mengirimkan permintaan akses ke *server*.

2. *Verification*: *Server* memverifikasi kredensial pengguna. Jika pengguna valid, proses berlanjut ke tahap berikutnya.
3. *Token Submission*: *Server* menghasilkan token autentikasi untuk pengguna, yang dapat digunakan dalam jangka waktu tertentu.
4. *Storage*: Token dikirimkan kembali kepada pengguna dan disimpan untuk digunakan dalam akses selanjutnya ke halaman web yang dilindungi.
5. *Expiration*: Token tetap aktif sampai pengguna melakukan *logout* atau sampai masa aktif token berakhir.

Metode autentikasi berbasis token meningkatkan fleksibilitas dan keamanan dalam manajemen sesi pengguna. Misalnya, dalam aplikasi *mobile* dan *single-page applications* (SPA), token dapat dengan mudah disimpan dan digunakan untuk melakukan permintaan berulang ke *server* tanpa memerlukan autentikasi ulang. Selain itu, karena token tidak menyimpan informasi sensitif tentang pengguna, risiko eksposur data berkurang signifikan.

Dengan menggunakan algoritma kriptografi yang kuat, token autentikasi tidak hanya memastikan bahwa hanya pengguna yang terverifikasi yang mendapatkan akses, tetapi juga menjaga integritas dan keamanan komunikasi antara pengguna dan *server*. Implementasi autentikasi berbasis token ini telah menjadi standar dalam pengembangan aplikasi web modern karena keunggulannya dalam efisiensi dan keamanan.

F. JSON Web Token (JWT)

JSON Web Token (JWT) adalah standar terbuka (RFC 7519) yang mendefinisikan mekanisme untuk mengirimkan informasi secara ringkas antara dua pihak dalam bentuk objek JSON. Informasi ini dapat diverifikasi karena ditandatangani secara digital menggunakan *secret key* (dengan algoritma HMAC) atau pasangan kunci publik/pribadi (dengan algoritma RSA atau ECDSA) [11]. Selain itu, JWT dapat ditransmisikan melalui *query string*, *header*, dan *body* dari POST *request* karena ukurannya yang kompak.

JWT terdiri dari tiga bagian utama yang dipisahkan oleh titik (.), yaitu *header*, *payload*, dan *signature*. Secara umum, format JWT terlihat seperti berikut.

ppppp.qqqqq.rrrrr

(*header.payload.signature*)

Fig. 4 Format Umum dari JWT

Sumber : Dokumentasi Penulis

Setiap bagian tersebut di-*encode* menggunakan format base64. Untuk menghasilkan token dalam format JWT, diperlukan input dari setiap bagian yang ditunjukkan pada format berikut.

Header

```
{
 "alg": "HS256",
 "typ": "JWT"
}
```

Payload

```
{
 "sub": "1234567890123",
 "name": "Alice Bob Charlie",
 "admin": true
}
```

Signature

```
HMACSHA256(
 base64UrlEncode(header) + "." +
 base64UrlEncode(payload),
 secret)
```

Fig. 5 Konten dari Tiap Bagian Penyusun JWT

Sumber : Dokumentasi Penulis

Secara ringkas, untuk menghasilkan token, digunakan rumus berikut.

$$Token = f(Base64Encode) \sum_{n=\alpha,\beta}^{\infty} (header, payload, signature)$$

Fig. 6 Rumus untuk Membangkitkan JWT

Sumber : [12]

Bagian *header* biasanya berisi informasi tentang tipe token dan algoritma yang digunakan untuk *signing*. Bagian *payload* berisi data pengguna umum dan informasi tambahan lainnya. Bagian *signature* berisi tanda tangan yang dihasilkan dengan menggabungkan *encoded header* (base64), *encoded payload* (base64), dan *secret key* menggunakan algoritma yang ditentukan dalam *header*. Berikut ini adalah contoh final dari JWT.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

Fig. 7 Bentuk Final JWT

Sumber : [13]

IV. Implementasi dan Eksperimen

A. Lingkungan Implementasi

Untuk melakukan eksperimen ini, program akan dibuat menggunakan bahasa Go dengan *library* dan *tools* yang relevan untuk mengevaluasi kinerja dan keamanan algoritma HMAC SHA-3 dan RSA dalam implementasi JSON Web Token (JWT). Lingkungan implementasi mencakup perangkat keras dengan prosesor Intel Core i7, RAM 16GB, dan penyimpanan SSD 1TB, serta perangkat lunak berupa sistem operasi Windows 10 Home Single Language 64-bit (10.0,

Build 19045), library `github.com/golang-jwt/jwt/v4` untuk JWT, `crypto` untuk algoritma kriptografi, dan beberapa fungsi yang dibuat untuk mengukur kinerja JWT.

B. Implementasi Pembangkitan Kunci dan JWT dengan Algoritma RSA

Berikut ini adalah `rsa.go`, yang isinya adalah implementasi dari fungsi pembangkitan kunci *public-key* dan *private-key* dengan algoritma RSA, dan dilakukan implementasi JWT beserta tanda tangan digitalnya.

```
// File: rsa.go
// Berikut adalah program untuk membuat dan memverifikasi token JWT dengan algoritma RSA.

package main

import (
	"crypto"
	"crypto/rand"
	"crypto/rsa"
	"crypto/sha256"
	"encoding/base64"
	"errors"

	"github.com/golang-jwt/jwt/v4"
)

func generateRSAKeys() (*rsa.PrivateKey, *rsa.PublicKey, error) {
    privateKey, err := rsa.GenerateKey(rand.Reader, 2048)
    if err != nil {
        return nil, nil, err
    }
    return privateKey, &privateKey.PublicKey, nil
}

type RSASigningMethod struct {
    Name string
}

func (m *RSASigningMethod) Alg() string {
    return m.Name
}

func (m *RSASigningMethod) Sign(signingString string, key interface{}) (string, error) {
    var rsaPrivateKey *rsa.PrivateKey

    switch k := key.(type) {
    case *rsa.PrivateKey:
        rsaPrivateKey = k
    default:
        return "", errors.New("invalid key type")
    }

    hash := sha256.New()
    hash.Write([]byte(signingString))
    hashed := hash.Sum(nil)

    signature, err := rsa.SignPKCS1v15(rand.Reader, rsaPrivateKey, crypto.SHA256, hashed)
    if err != nil {
        return "", err
    }
    return base64.RawURLEncoding.EncodeToString(signature), nil
}

func (m *RSASigningMethod) Verify(signingString, signature string, key interface{}) error {
    var rsaPublicKey *rsa.PublicKey

    switch k := key.(type) {
    case *rsa.PublicKey:
        rsaPublicKey = k
    default:
        return errors.New("invalid key type")
    }

    sig, err := base64.RawURLEncoding.DecodeString(signature)
    if err != nil {
        return err
    }

    hash := sha256.New()
    hash.Write([]byte(signingString))
    hashed := hash.Sum(nil)
```

```
    return rsa.VerifyPKCS1v15(rsaPublicKey,
crypto.SHA256, hashed, sig)
}

func init() {
    jwt.RegisterSigningMethod("RS256",
func() jwt.SigningMethod {
        return &RSASigningMethod{Name:
"RS256"}
    })
}

func createRSAToken(privateKey
*rsa.PrivateKey) (string, error) {
    token :=
jwt.New(jwt.GetSigningMethod("RS256"))
    claims := jwt.MapClaims{
        "foo": "bar", // ini adalah contoh
payload
    }
    token.Claims = claims
    return token.SignedString(privateKey)
}

func verifyRSAToken(tokenString string,
publicKey *rsa.PublicKey) (*jwt.Token,
error) {
    return jwt.Parse(tokenString,
func(token *jwt.Token) (interface{}, error)
{
        if token.Method.Alg() != "RS256" {
            return nil,
jwt.ErrSignatureInvalid
        }
        return publicKey, nil
    })
}
```

Fig. 8 Blok Kode `rsa.go`

Sumber : Dokumentasi Penulis

Alur dari algoritma `rsa.go` di atas secara lebih detailnya adalah sebagai berikut. Pertama, fungsi `generateRSAKeys` digunakan untuk menghasilkan sepasang kunci RSA (privat dan publik) dengan panjang 2048 bit. Fungsi ini menggunakan generator angka acak kriptografis untuk memastikan keamanan kunci yang dihasilkan. Struktur `RSASigningMethod` kemudian didefinisikan untuk mewakili metode penandatanganan RSA dengan SHA-256, menyediakan metode `Alg` untuk mengembalikan nama algoritma, serta metode `Sign` dan `Verify` untuk penandatanganan dan verifikasi token.

Dalam metode `Sign`, string yang akan ditandatangani terlebih dahulu di-hash menggunakan SHA-256. Hash ini kemudian ditandatangani menggunakan kunci privat RSA dan fungsi `rsa.SignPKCS1v15`, yang menghasilkan tanda tangan digital yang dikodekan dengan base64.

Untuk verifikasi tanda tangan, metode `Verify` digunakan. Pertama, tanda tangan yang diterima di-decode dari format base64, kemudian string asli di-hash lagi menggunakan SHA-256. Hash ini diverifikasi terhadap tanda tangan yang diberikan menggunakan kunci publik RSA dan fungsi `rsa.VerifyPKCS1v15`.

Fungsi `init` mendaftarkan metode penandatanganan `RS256` menggunakan JWT library. Fungsi `createRSAToken` membuat token JWT baru dengan klaim sederhana, yang kemudian ditandatangani menggunakan kunci privat RSA. Fungsi `verifyRSAToken` memverifikasi token JWT yang diberikan menggunakan kunci publik RSA untuk memastikan bahwa token tersebut sah dan belum dimodifikasi.

Secara keseluruhan, alur kerja ini mencakup pembuatan kunci RSA, penandatanganan token JWT menggunakan SHA-256 sebagai fungsi hash, dan verifikasi tanda tangan untuk memastikan integritas dan keaslian token.

C. Implementasi Pembangkitan JWT dengan Algoritma HMAC SHA-3

Berikut ini adalah `hmac-sha-3.go`, yang berisikan implementasi algoritma HMAC SHA-3 yang digunakan untuk menghasilkan dan memverifikasi JSON Web Token (JWT) dengan metode penandatanganan HMAC menggunakan SHA-3.

```
// File: hmac-sha-3.go
// Berikut adalah program untuk membuat dan
memverifikasi token JWT dengan algoritma
HMAC SHA-3.

package main

import (
    "crypto/hmac"
    "encoding/base64"
    "errors"
    "github.com/golang-jwt/jwt/v4"
    "golang.org/x/crypto/sha3"
)

type HMACSHA3SigningMethod struct {
    Name string
}

func (m *HMACSHA3SigningMethod) Alg()
```

```
string {
    return m.Name
}

func (m *HMACSHA3SigningMethod)
Sign(signingString string, key interface{})
(string, error) {
    var keyBytes []byte

    switch k := key.(type) {
    case []byte:
        keyBytes = k
    case string:
        keyBytes = []byte(k)
    default:
        return "", errors.New("invalid key
type")
    }

    mac := hmac.New(sha3.New256, keyBytes)
    mac.Write([]byte(signingString))
    return
base64.RawURLEncoding.EncodeToString(mac.Su
m(nil)), nil
}

func (m *HMACSHA3SigningMethod)
Verify(signingString, signature string, key
interface{}) error {
    var keyBytes []byte

    switch k := key.(type) {
    case []byte:
        keyBytes = k
    case string:
        keyBytes = []byte(k)
    default:
        return errors.New("invalid key
type")
    }

    sig, err :=
base64.RawURLEncoding.DecodeString(signatur
e)
    if err != nil {
        return err
    }
    mac := hmac.New(sha3.New256, keyBytes)
    mac.Write([]byte(signingString))
    if !hmac.Equal(sig, mac.Sum(nil)) {
        return errors.New("signature is
invalid")
    }

    return nil
}

func init() {
    jwt.RegisterSigningMethod("HS3", func()
jwt.SigningMethod {
        return &HMACSHA3SigningMethod{Name:
"HS3"}
    })
}

func createHMACSHA3Token(secret []byte)
(string, error) {
    token :=
jwt.New(jwt.GetSigningMethod("HS3"))
    claims := jwt.MapClaims{
        "foo": "bar",
    }
    token.Claims = claims
    return token.SignedString(secret)
}

func verifyHMACSHA3Token(tokenString
string, secret []byte) (*jwt.Token, error)
{
    return jwt.Parse(tokenString,
func(token *jwt.Token) (interface{}, error)
{
        if token.Method.Alg() != "HS3" {
            return nil,
jwt.ErrSignatureInvalid
        }
        return secret, nil
    })
}
```

Fig. 9 Blok Kode `hmac-sha-3.go`

Sumber : Dokumentasi Penulis

Alur dari algoritma `hmac-sha-3.go` di atas secara lebih detailnya adalah sebagai berikut. Pertama, tipe `HMACSHA3SigningMethod` didefinisikan untuk mewakili metode penandatanganan HMAC dengan SHA-3. Metode `Alg` mengembalikan nama algoritma. Metode `Sign`

digunakan untuk menandatangani string dengan menggunakan kunci yang diberikan. Kunci ini dapat berupa string atau byte array dan dikonversi ke byte array jika perlu. Kemudian, algoritma HMAC dengan SHA-3 (SHA3-256) digunakan untuk menghasilkan tanda tangan dengan menulis string yang akan ditandatangani ke dalam fungsi hash. Hasilnya dikodekan dengan base64.

Metode `Verify` memverifikasi tanda tangan dengan mendekode tanda tangan dari base64 dan menghitung HMAC baru dengan menggunakan string yang diberikan dan kunci yang sama. Jika HMAC yang dihasilkan cocok dengan tanda tangan yang didekode, verifikasi berhasil; jika tidak, verifikasi gagal.

Fungsi `init` mendaftarkan metode penandatanganan `HS3` menggunakan library JWT. Fungsi `createHMACSHA3Token` membuat token JWT baru dengan klaim sederhana, yang kemudian ditandatangani menggunakan kunci rahasia dengan metode HMAC SHA-3. Fungsi `verifyHMACSHA3Token` memverifikasi token JWT yang diberikan dengan memastikan bahwa metode penandatanganan yang digunakan adalah `HS3` dan kunci rahasia yang sesuai.

Secara keseluruhan, alur kerja ini mencakup penandatanganan string dengan HMAC SHA-3, verifikasi tanda tangan, pembuatan token JWT, dan verifikasi token JWT untuk memastikan integritas dan keaslian token.

D. Implementasi Fungsi-fungsi untuk Dilakukannya Benchmarking

Untuk membandingkan kinerja dan efisiensi antara algoritma RSA dan HMAC SHA-3 dalam menghasilkan dan memverifikasi JSON Web Token (JWT), suatu kasus uji, yang dibuat pada file `benchmark.go`, dibuat untuk mengevaluasi waktu yang dibutuhkan dalam operasi pembuatan dan verifikasi sebanyak 1000 JWT, serta untuk menganalisis struktur token yang dihasilkan oleh masing-masing algoritma.

```
// File: benchmark.go
// Berikut adalah program untuk
benchmarking pembuatan dan verifikasi token
JWT dengan algoritma RSA dan HMAC SHA-3.

package main

import (
	"crypto/rsa"
	"fmt"
	"time"
)

// Function to benchmark RSA token creation
and verification
func benchmarkRSA(privateKey
*rsa.PrivateKey, publicKey *rsa.PublicKey)
{
	// Benchmark token creation
	start := time.Now()
	for i := 0; i < 1000; i++ {
		createRSAToken(privateKey)
	}
	creationTime := time.Since(start)

	// Benchmark token verification
	tokenString, _ :=
createRSAToken(privateKey)

	fmt.Println("JWT hasil generasi
menggunakan RSA: ", tokenString+"\n")

	start = time.Now()
	for i := 0; i < 1000; i++ {
		verifyRSAToken(tokenString,
publicKey)
	}
	verificationTime := time.Since(start)

	fmt.Println("1000x JWT Creation using
RSA Time:", creationTime)
	fmt.Println("1000x JWT Verification
using RSA Time:", verificationTime)
}

// Function to benchmark HMAC SHA-3 token
creation and verification
func benchmarkHMACSHA3(hmacSecret []byte) {
	// Benchmark token creation
	start := time.Now()
	for i := 0; i < 1000; i++ {

createHMACSHA3Token(hmacSecret)
	}
	creationTime := time.Since(start)

	// Benchmark token verification
	tokenString, _ :=
createHMACSHA3Token(hmacSecret)

	fmt.Println("\nJWT hasil generasi
menggunakan HMAC SHA-3: ",
tokenString+"\n")

	start = time.Now()
```

```
	for i := 0; i < 1000; i++ {

verifyHMACSHA3Token(tokenString,
hmacSecret)
	}
	verificationTime := time.Since(start)

	fmt.Println("1000x JWT Creation using
HMAC SHA-3 Time:", creationTime)
	fmt.Println("1000x JWT Verification
using HMAC SHA-3 Time:", verificationTime)
}
```

Fig. 10 Blok Kode `benchmark.go`

Sumber : Dokumentasi Penulis

E. Implementasi Fungsi-fungsi untuk Dilakukannya Algorithm-Confusion Test

Setelah itu, akan dilakukan *Algorithm-Confusion Test* untuk memastikan bahwa token yang dibuat dengan algoritma yang satu (RSA/HMAC SHA-3) tidak dapat diverifikasi dengan algoritma lainnya (RSA/HMAC SHA-3). Berikut adalah *file* `algorithmConfusion.go`.

```
// File: algorithmConfusion.go
// Berikut adalah program untuk melakukan
uji algorithm confusion pada JWT yang
dibuat dengan algoritma HMAC SHA-256 dan
di-parse dengan algoritma RSA SHA-256.
package main

import (
	"crypto/rsa"
	"fmt"
)

// Function for algorithm confusion test
func algorithmConfusion(privateKey
*rsa.PrivateKey, publicKey *rsa.PublicKey,
hmacSecret []byte) {
	// Create RSA token
	rsaToken, err :=
createRSAToken(privateKey)
	if err != nil {
		panic(err)
	}
	fmt.Println("Generated RSA Token:",
rsaToken)

	// Try to verify RSA token with HMAC
SHA-3
	if _, err :=
verifyHMACSHA3Token(rsaToken, hmacSecret);
err != nil {
		fmt.Println("HMAC SHA-3
verification failed as expected:", err)
	} else {
		fmt.Println("HMAC SHA-3
verification should have failed but
didn't")
	}

	// Create HMAC SHA-3 token
	hmacToken, err :=
createHMACSHA3Token(hmacSecret)
	if err != nil {
		panic(err)
	}
	fmt.Println("Generated HMAC SHA-3
Token:", hmacToken)

	// Try to verify HMAC SHA-3 token
with RSA
	if _, err :=
verifyRSAToken(hmacToken, publicKey); err
!= nil {
		fmt.Println("RSA verification
failed as expected:", err)
	} else {
		fmt.Println("RSA verification
should have failed but didn't")
	}
}
```

Fig. 11 Blok Kode `algorithmConfusion.go`

Sumber : Dokumentasi Penulis

F. Implementasi Fungsi-fungsi untuk Dilakukannya Integrity Checking Test

Setelah itu, akan dilakukan *Integrity Checking Test* untuk memastikan bahwa token yang diubah tidak dapat diverifikasi, yang menunjukkan bahwa integritas token telah terjaga. Berikut adalah *file* `integrityCheck.go`.

```
// File: integrityCheck.go
// Berikut adalah program untuk melakukan
integrity check pada token JWT dengan
algoritma RSA dan HMAC SHA-3.

package main
```

```
import (
	"crypto/rsa"
	"fmt"
)

// Function for integrity check with RSA
func integrityCheckRSA(publicKey
*rsa.PublicKey) {
	privateKey, _, err :=
generateRSAKeys()
	if err != nil {
		panic(err)
	}

	// Create a valid token
	tokenString, err :=
createRSAToken(privateKey)
	if err != nil {
		panic(err)
	}

	fmt.Println("Generated RSA JWT for
Integrity Check:", tokenString)

	// Tamper the token
	tamperedToken := tokenString +
"tampered"

	// Verify the original token
	_, err = verifyRSAToken(tokenString,
publicKey)
	if err != nil {
		fmt.Println("Original token
verification failed:", err)
	} else {
		fmt.Println("Original token
verification succeeded")
	}

	// Verify the tampered token
	_, err =
verifyRSAToken(tamperedToken, publicKey)
	if err != nil {
		fmt.Println("Tampered token
verification failed as expected:", err)
	} else {
		fmt.Println("Tampered token
verification succeeded unexpectedly")
	}
}

// Function for integrity check with HMAC
func integrityCheckHMAC() {
	// Create a valid token
	secret := []byte("secret")
	tokenString, err :=
createHMACSHA3Token(secret)
	if err != nil {
		panic(err)
	}

	fmt.Println("Generated HMAC JWT for
Integrity Check:", tokenString)

	// Tamper the token
	tamperedToken := tokenString +
"tampered"

	// Verify the original token
	_, err =
verifyHMACSHA3Token(tokenString, secret)
	if err != nil {
		fmt.Println("Original token
verification failed:", err)
	} else {
		fmt.Println("Original token
verification succeeded")
	}

	// Verify the tampered token
	_, err =
verifyHMACSHA3Token(tamperedToken, secret)
	if err != nil {
		fmt.Println("Tampered token
verification failed as expected:", err)
	} else {
		fmt.Println("Tampered token
verification succeeded unexpectedly")
	}
}
```

Fig. 12 Blok Kode `integrityCheck.go`

Sumber : Dokumentasi Penulis

G. Implementasi Fungsi-fungsi untuk Dilakukannya Payload Size Impact Test

Setelah itu, akan dilakukan *Payload Size Impact Test*, yaitu uji performa pembuatan dan verifikasi token dengan berbagai ukuran *payload* untuk melihat bagaimana ukuran data yang disematkan dalam JWT memengaruhi kinerja. Caranya adalah dengan membuat token dengan *payload* yang berbeda-beda ukurannya dan ukur waktu yang diperlukan untuk pembuatan dan verifikasinya. Berikut adalah *file* `payloadSizeImpact.go`.

```
// File: payloadSizeImpact.go
// Berikut adalah program untuk menguji pengaruh ukuran payload pada pembuatan dan verifikasi token JWT dengan algoritma RSA dan HMAC SHA-3.
package main

import (
	"crypto/rsa"
	"fmt"
	"time"

	"github.com/golang-jwt/jwt/v4"
)

func payloadSizeImpact(privateKey *rsa.PrivateKey, publicKey *rsa.PublicKey, hmacSecret []byte) {
	sizes := []int{100, 1000, 10000}
	for _, size := range sizes {
		payload := generatePayload(size)
		// RSA
		start := time.Now()

		createCustomRSAToken(privateKey, payload)
		fmt.Printf("RSA token creation time with payload size %d: %s\n", size, time.Since(start))

		tokenString, _ := createCustomRSAToken(privateKey, payload)
		start = time.Now()
		verifyRSAToken(tokenString, publicKey)
		fmt.Printf("RSA token verification time with payload size %d: %s\n", size, time.Since(start))

		// HMAC SHA-3
		start = time.Now()

		createCustomHMACSHA3Token(hmacSecret, payload)
		fmt.Printf("HMAC SHA-3 token creation time with payload size %d: %s\n", size, time.Since(start))

		tokenString, _ = createCustomHMACSHA3Token(hmacSecret, payload)
		start = time.Now()

		verifyHMACSHA3Token(tokenString, hmacSecret)
		fmt.Printf("HMAC SHA-3 token verification time with payload size %d: %s\n", size, time.Since(start))
	}
}

// Function to generate a payload of specified size
func generatePayload(size int) map[string]interface{} {
	payload := make(map[string]interface{})
	for i := 0; i < size; i++ {
		payload[fmt.Sprintf("key%d", i)] = fmt.Sprintf("value%d", i)
	}
	return payload
}

// Function to create RSA token with custom payload
func createCustomRSAToken(privateKey *rsa.PrivateKey, payload map[string]interface{}) (string, error) {
	token := jwt.NewWithClaims(jwt.SigningMethodRS256, jwt.MapClaims(payload))
	tokenString, err := token.SignedString(privateKey)
	return tokenString, err
}

// Function to create HMAC SHA-3 token with
```

```
custom payload
func createCustomHMACSHA3Token(secret
[]byte, payload map[string]interface{})
(string, error) {
	token :=
jwt.NewWithClaims(jwt.SigningMethodHS256,
jwt.MapClaims(payload))
	tokenString, err :=
token.SignedString(secret)
	return tokenString, err
}
```

Fig. 13 Blok Kode `payloadSizeImpact.go`

Sumber : Dokumentasi Penulis

H. Implementasi Fungsi-fungsi untuk Dilakukannya Stress Test

Setelah itu, akan dilakukan *Stress Test* untuk menyimulasikan skenario beban tinggi dengan membuat dan memverifikasi sejumlah besar token secara bersamaan untuk melihat bagaimana sistem menangani beban berat. Tekniknya adalah dengan cara menjalankan banyak GoRoutine (semacam *thread*) yang melakukan pembuatan dan verifikasi token secara bersamaan dan mengukur kinerja serta kestabilan sistem. Berikut adalah *file* `stressTesting.go`.

```
// File: stressTesting.go
// Berikut adalah program untuk melakukan
stress testing dengan membuat dan
memverifikasi token secara parallel.

package main

import (
	"crypto/rsa"
	"fmt"
	"time"
)

// Function to test stress testing by
creating and verifying tokens in parallel
func stressTesting(privateKey
*rsa.PrivateKey, publicKey *rsa.PublicKey,
hmacSecret []byte) {
	numGoroutines := 100
	done := make(chan bool,
numGoroutines)

	// Stress test RSA
	start := time.Now()
	for i := 0; i < numGoroutines; i++ {
		go func() {
			for j := 0; j < 1000;
j++ {
				tokenString, _ :=
createRSAToken(privateKey)

verifyRSAToken(tokenString, publicKey)
			}
			done <- true
		}()
	}
	for i := 0; i < numGoroutines; i++ {
		<-done
	}
	fmt.Println("RSA stress test time:",
time.Since(start))

	// Stress test HMAC SHA-3
	start = time.Now()
	for i := 0; i < numGoroutines; i++ {
		go func() {
			for j := 0; j < 1000;
j++ {
				tokenString, _ :=
createHMACSHA3Token(hmacSecret)

verifyHMACSHA3Token(tokenString,
hmacSecret)
			}
			done <- true
		}()
	}
	for i := 0; i < numGoroutines; i++ {
		<-done
	}
	fmt.Println("HMAC SHA-3 stress test
time:", time.Since(start))
}
```

Fig. 14 Blok Kode `stressTesting.go`

Sumber : Dokumentasi Penulis

V. PENGUJIAN

Untuk menguji kinerja dan keamanan JWT yang dibuat dengan RSA dan HMAC SHA-3, kita dapat menggunakan program-program uji yang sudah diimplementasikan di atas, yaitu program *benchmarking*, *algorithm-confusion test*, *integrity checking test*, *payload size impact test*, dan *stress testing*. Untuk mengeksekusi semua program uji dalam satu waktu, telah dibuat *file* `main.go`, yang mana jika dijalankan

akan otomatis menjalankan seluruh program uji yang telah dibuat. Berikut adalah *file* `main.go`.

```
// File: main.go
// Berikut adalah program utama yang
menjalankan semua program uji yang telah
diimplementasikan.

package main

import (
	"fmt"
)

func main() {
	// Generate RSA keys
	privateKey, publicKey, err :=
generateRSAKeys()
	if err != nil {
		panic(err)
	}

	// Generate HMAC SHA-3 secret
	hmacSecret := []byte("secret")

	// Benchmarking RSA
	fmt.Println("=== Benchmark JWT (RSA)
===")
	benchmarkRSA(privateKey, publicKey)

	// Benchmarking HMAC SHA-3
	fmt.Println("=== Benchmark JWT (HMAC
SHA-3) ===")
	benchmarkHMACSHA3(hmacSecret)

	fmt.Println("")

	// Integrity check for RSA
	fmt.Println("=== Integrity Check JWT
(RSA) ===")
	integrityCheckRSA(publicKey)

	fmt.Println("")

	// Integrity check for HMAC SHA-3
	fmt.Println("=== Integrity Check JWT
(HMAC SHA-3) ===")
	integrityCheckHMAC()

	fmt.Println("")

	// Test algorithm confusion
	fmt.Println("=== Algorithm Confusion
Test ===")
	algorithmConfusion(privateKey,
publicKey, hmacSecret)

	fmt.Println("")

	// Test payload size impact
	fmt.Println("=== Payload Size Impact
Test ===")
	payloadSizeImpact(privateKey,
publicKey, hmacSecret)

	fmt.Println("")

	// Stress testing
	fmt.Println("=== Stress Testing ===")
	stressTesting(privateKey, publicKey,
hmacSecret)
}
```

Fig. 15 Blok Kode `main.go`

Sumber : Dokumentasi Penulis

Ketika `main.go` dijalankan, *output* pada *terminal* adalah sebagai berikut.

```
PS
A:\penggunaan-algoritma-kriptografi-HMAC-SHA-3-dan-
RSA-untuk-JWT> go run main.go hmac-sha-3.go rsa.go
algorithmConfusion.go benchmark.go integrityCheck.go
payloadSizeImpact.go stressTesting.go

=== Benchmark JWT (RSA) ===
JWT hasil generasi menggunakan RSA:
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJi
YXIifQ.9GqYRhFjy9A6LVtdnM_uQwcPlVTmatftpzSUex
Nr5tHFmtc_UPQi84gjOgCfsA31PBzxqxuRwOu_TM5KQ
4Y9SuZPAGqw_5ZhDiJsmmBeWGCkYMBzo3AC1vZHz
wSEzNcQG_r0CYNS1b2rqMhCEbd-S_V5luZLXuOTLFa
EYu0nyFV3KmwuY0BwcQvH7CQr8Pi6_3slTBlRBmnZR
zuk_1qselSs4pq8-ZZ6YDZap4jFG3-nbAr71cP1dGGIpXT
VrjzxQ0QLs9ZiNzLvnNW9HaCxYHdWOtynGuKZuhi7-h
5xc2jJ8GdiMsheJRXyO_CWNqehq5hh7cDRBEAX-QPM
4jNFYg

1000x JWT Creation using RSA Time: 1.254932s
1000x JWT Verification using RSA Time: 152.4407ms
```

```
=== Benchmark JWT (HMAC SHA-3) ===
JWT hasil generasi menggunakan HMAC SHA-3:
eyJhbGciOiJIUzMiLCJ0eXAiOiJKV1QifQ.eyJmb28iOiJiYXIifQ.irFOB4hOLI2H3tM6HDtpt14iFp0-YrhxZzg8JWUZzPE

1000x JWT Creation using HMAC SHA-3 Time: 5.0199ms
1000x JWT Verification using HMAC SHA-3 Time: 5.0337ms

=== Integrity Check JWT (RSA) ===
Generated RSA JWT for Integrity Check:
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJiYXIifQ.IFqCIZblZnTx6meU3lUjyI7wANKA8vTQrSoi8qlssLdle3RNC2P4YsOcFKH5-c4AnR8V2ABiiyJf-7QBjdz6_KD2jD9ANZTVKBU3TX_MYEZmj9AIvPxnsXcDFwylzhNGa-oJF42NzylumRgdNKrZr3q9QA22-QAZrq0NwlZxCwVGUUFQRhYpoqQ_FoSAy-Tjw9kbaw1w6Zb1WI__Om9IBlTVPQNiLwjHKssHtx_czbWc-50kqINPnqMIDPZrjyUqXohhUz_kayFWL43CmjAVZnBh6JU8FOSurwWsGf0bhfleHDiBwFoqp6u35XHY2wYfQIba9fmuyExyGKU1-VtLfw
Original token verification failed: crypto/rsa: verification error
Tampered token verification failed as expected: crypto/rsa: verification error

=== Integrity Check JWT (HMAC SHA-3) ===
Generated HMAC JWT for Integrity Check:
eyJhbGciOiJIUzMiLCJ0eXAiOiJKV1QifQ.eyJmb28iOiJiYXIifQ.irFOB4hOLI2H3tM6HDtpt14iFp0-YrhxZzg8JWUZzPE
Original token verification succeeded
Tampered token verification failed as expected: signature is invalid

=== Algorithm Confusion Test ===
Generated RSA Token:
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJiYXIifQ.9GqYRhFjy9A6LVtdnM_uQwcPlVTmatftpzSUexNr5tHFmtc_UPQi84gjOgCfsA31PBzxqxuRwOu_TM5KQ4Y9SuZPAGqw_5ZhDiJsmmBeWGCkYMBzo3AC1vZHzwSEzNcQG_r0CYNS1b2rqMhCEbd-S_V5luZLXuOTLFaEYu0nyFV3KmwuY0BwcQvH7CQr8Pi6_3slTBlRBmnZRzuk_1qselSs4pq8-ZZ6YDZap4jFG3-nbAr71cP1dGGIpXTVrjzxQ0QLs9ZiNzLvnNW9HaCxYHdWOtynGuKZuhi7-h5xc2jJ8GdiMsheJRXyO_CWNqehq5hh7cDRBEAX-QPM4jNFYg
HMAC SHA-3 verification failed as expected: signature is invalid
Generated HMAC SHA-3 Token:
eyJhbGciOiJIUzMiLCJ0eXAiOiJKV1QifQ.eyJmb28iOiJiYXIifQ.irFOB4hOLI2H3tM6HDtpt14iFp0-YrhxZzg8JWUZzPE
RSA verification failed as expected: signature is invalid
```

```
=== Payload Size Impact Test ===
RSA token creation time with payload size 100: 997µs
RSA token verification time with payload size 100: 0s
HMAC SHA-3 token creation time with payload size 100: 0s
HMAC SHA-3 token verification time with payload size 100: 0s
RSA token creation time with payload size 1000: 1.9968ms
RSA token verification time with payload size 1000: 997.2µs
HMAC SHA-3 token creation time with payload size 1000: 997.9µs
HMAC SHA-3 token verification time with payload size 1000: 996.6µs
RSA token creation time with payload size 10000: 6.9805ms
RSA token verification time with payload size 10000: 7.4445ms
HMAC SHA-3 token creation time with payload size 10000: 5.8863ms
HMAC SHA-3 token verification time with payload size 10000: 4.9867ms

=== Stress Testing ===
RSA stress test time: 26.8158878s
HMAC SHA-3 stress test time: 437.1332ms
```

Fig. 16 Tampilan pada *Terminal* ketika `main.go` dijalankan

Sumber : Dokumentasi Penulis

Dari hasil tersebut, kita akan telaah satu per satu, yaitu sebagai berikut.

1. Hasil *Benchmark* JWT

```
=== Benchmark JWT (RSA) ===
JWT hasil generasi menggunakan RSA:
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJiYXIifQ.9GqYRhFjy9A6LVtdnM_uQwcPlVTmatftpzSUexNr5tHFmtc_UPQi84gjOgCfsA31PBzxqxuRwOu_TM5KQ4Y9SuZPAGqw_5ZhDiJsmmBeWGCkYMBzo3AC1vZHzwSEzNcQG_r0CYNS1b2rqMhCEbd-S_V5luZLXuOTLFaEYu0nyFV3KmwuY0BwcQvH7CQr8Pi6_3slTBlRBmnZRzuk_1qselSs4pq8-ZZ6YDZap4jFG3-nbAr71cP1dGGIpXTVrjzxQ0QLs9ZiNzLvnNW9HaCxYHdWOtynGuKZuhi7-h5xc2jJ8GdiMsheJRXyO_CWNqehq5hh7cDRBEAX-QPM4jNFYg

1000x JWT Creation using RSA Time: 1.254932s
1000x JWT Verification using RSA Time: 152.4407ms

=== Benchmark JWT (HMAC SHA-3) ===
JWT hasil generasi menggunakan HMAC SHA-3:
eyJhbGciOiJIUzMiLCJ0eXAiOiJKV1QifQ.eyJmb28iOiJiYXIifQ.irFOB4hOLI2H3tM6HDtpt14iFp0-YrhxZzg8JWUZzPE

1000x JWT Creation using HMAC SHA-3 Time: 5.0199ms
1000x JWT Verification using HMAC SHA-3 Time: 5.0337ms
```

Fig. 17 Tampilan pada *Terminal* untuk Hasil *Benchmark* JWT (RSA dan HMAC SHA-3)

Sumber : Dokumentasi Penulis

Dari hasil tersebut, kita akan telaah satu per satu, yaitu sebagai berikut. Pengujian ini mengukur waktu yang dibutuhkan untuk membuat dan memverifikasi 1000 token berturut-turut untuk setiap algoritma. Hasilnya adalah sebagai berikut.

TABEL 2. HASIL DARI *BENCHMARK* JWT

Algorithm	***Creation Time (1000x)***	***Verification Time (1000x)***
RSA	1.254932s	152.4407ms
HMAC SHA-3	5.0199ms	5.0337ms

Sumber : Dokumentasi Penulis

Dapat dilihat bahwa pembuatan token RSA membutuhkan waktu 1.254932 detik untuk 1000 token, dan verifikasi membutuhkan 152.4407 milidetik untuk 1000 token. Pembuatan token HMAC SHA-3 membutuhkan waktu 5.0199 milidetik untuk 1000 token, dan verifikasi membutuhkan 5.0337 milidetik untuk 1000 token.

Dari hasil tersebut, HMAC SHA-3 jauh lebih cepat dalam pembuatan dan verifikasi token dibandingkan RSA. Ini menunjukkan bahwa HMAC SHA-3 lebih efisien dalam skenario dengan beban yang tinggi.

2. Hasil *Integrity Checking Test*

```
=== Integrity Check JWT (RSA) ===
Generated RSA JWT for Integrity Check:
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJi
YXIifQ.IFqCIZblZnTx6meU3lUjyI7wANKA8vTQrSoi8ql
ssLdle3RNC2P4YsOcFKH5-c4AnR8V2ABiiyJf-7QBjdz6_
KD2jD9ANZTVKBU3TX_MYEZmj9AIvPxnsXcDFwylzh
NGa-oJF42NzylumRgdNKrZr3q9QA22-QAZrq0NwlZxC
wVGUUFQRhYpoqQ_FoSAy-Tjw9kbaw1w6Zb1WI__Om
9IBlTVPQNiLwjHKssHtx_czbWc-50kqINPnqMIDPZrjyU
qXohhUz_kayFWL43CmjAVZnBh6JU8FOSurwWsGf0bhf
leHDiBwFoqp6u35XHY2wYfQIba9fmuyExyGKU1-VtLf
w
Original token verification failed: crypto/rsa: verification
error
Tampered token verification failed as expected: crypto/rsa:
verification error

=== Integrity Check JWT (HMAC SHA-3) ===
Generated HMAC JWT for Integrity Check:
eyJhbGciOiJIUzMiLCJ0eXAiOiJKV1QifQ.eyJmb28iOiJiY
XIifQ.irFOB4hOLI2H3tM6HDtpt14iFp0-YrhxZzg8JWUZz
PE
Original token verification succeeded
Tampered token verification failed as expected: signature is
invalid
```

Fig. 18 Tampilan pada *Terminal* untuk Hasil *Integrity Checking Test*

Sumber : Dokumentasi Penulis

Dari hasil tersebut, dapat dilihat bahwa, token asli berhasil diverifikasi, sedangkan token yang dimodifikasi gagal diverifikasi dengan kesalahan "*signature is invalid*" atau "crypto/rsa: verification error". Pengujian ini memastikan bahwa setiap modifikasi pada token terdeteksi dan menyebabkan kegagalan verifikasi. Hal ini menunjukkan bahwa integritas token dijaga dengan baik.

3. Hasil *Algorithm-Confusion Test*

```
=== Algorithm Confusion Test ===
Generated RSA Token:
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJi
YXIifQ.9GqYRhFjy9A6LVtdnM_uQwcPlVTmatftpzSUex
Nr5tHFmtc_UPQi84gjOgCfsA31PBzxqxuRwOu_TM5KQ
4Y9SuZPAGqw_5ZhDiJsmmBeWGCkYMBzo3AC1vZHz
wSEzNcQG_r0CYNS1b2rqMhCEbd-S_V5luZLXuOTLFa
EYu0nyFV3KmwuY0BwcQvH7CQr8Pi6_3slTBlRBmnZR
zuk_1qselSs4pq8-ZZ6YDZap4jFG3-nbAr71cP1dGGIpXT
VrjzxQ0QLs9ZiNzLvnNW9HaCxYHdWOtynGuKZuhi7-h
5xc2jJ8GdiMsheJRXyO_CWNqehq5hh7cDRBEAX-QPM
4jNFYg
HMAC SHA-3 verification failed as expected: signature is
invalid
Generated HMAC SHA-3 Token:
eyJhbGciOiJIUzMiLCJ0eXAiOiJKV1QifQ.eyJmb28iOiJiY
XIifQ.irFOB4hOLI2H3tM6HDtpt14iFp0-YrhxZzg8JWUZz
PE
RSA verification failed as expected: signature is invalid
```

Fig. 19 Tampilan pada *Terminal* untuk Hasil *Algorithm-Confusion Test*

Sumber : Dokumentasi Penulis

Dari hasil tersebut, dapat dilihat bahwa, JWT yang dibuat menggunakan algoritma RSA tidak dapat diverifikasi menggunakan HMAC SHA-3 dan juga sebaliknya. Pengujian ini berhasil menunjukkan bahwa kebingungan algoritma tidak terjadi. Token yang ditandatangani dengan satu algoritma tidak dapat diverifikasi dengan algoritma lain, yang menunjukkan keamanan yang baik dalam pemisahan algoritma.

4. Hasil *Payload Size Impact Test*

```
=== Payload Size Impact Test ===
RSA token creation time with payload size 100: 997µs
RSA token verification time with payload size 100: 0s
HMAC SHA-3 token creation time with payload size 100:
0s
HMAC SHA-3 token verification time with payload size
100: 0s
RSA token creation time with payload size 1000: 1.9968ms
RSA token verification time with payload size 1000:
```

997.2µs
HMAC SHA-3 token creation time with payload size 1000: 997.9µs
HMAC SHA-3 token verification time with payload size 1000: 996.6µs
RSA token creation time with payload size 10000: 6.9805ms
RSA token verification time with payload size 10000: 7.4445ms
HMAC SHA-3 token creation time with payload size 10000: 5.8863ms
HMAC SHA-3 token verification time with payload size 10000: 4.9867ms

Fig. 20 Tampilan pada *Terminal* untuk Hasil *Payload Size Impact Test*

Sumber : Dokumentasi Penulis

TABEL 3. HASIL DARI *PAYLOAD SIZE IMPACT TEST*

Payload Size	***RSA Creation Time***	***RSA Verification Time***	***HMAC SHA-3 Creation Time***	***HMAC SHA-3 Verification Time***
100	997µs	0s	0s	0s
1000	1.9968ms	997.2µs	997.9µs	996.6µs
10000	6.9805ms	7.4445ms	5.8863ms	4.9867ms

Sumber : Dokumentasi Penulis

Dari hasil tersebut, dapat dilihat bahwa, waktu pembuatan dan verifikasi token meningkat dengan ukuran *payload* yang lebih besar, terutama untuk RSA. HMAC SHA-3 menunjukkan waktu yang lebih cepat dibandingkan RSA untuk *payload* besar sehingga membuatnya lebih efisien dalam konteks performa.

5. Hasil *Stress Test*

=== Stress Testing ===
RSA stress test time: 26.8158878s
HMAC SHA-3 stress test time: 437.1332ms

Fig. 21 Tampilan pada *Terminal* untuk Hasil *Stress Test*

Sumber : Dokumentasi Penulis

Dari hasil tersebut, dapat dilihat bahwa, HMAC SHA-3 jauh lebih cepat dalam pembuatan dan verifikasi token dibandingkan RSA. Hal ini menunjukkan bahwa HMAC SHA-3 lebih efisien dalam skenario dengan beban tinggi.

VI. KESIMPULAN DAN SARAN

Berdasarkan hasil pengujian yang telah dilakukan, beberapa kesimpulan penting dapat diambil. Pertama, implementasi JWT menggunakan RSA dan HMAC SHA-3 terbukti aman. *Algorithm-Confusion Test* dan *Integrity Checking Test* menunjukkan bahwa JWT yang ditandatangani dengan satu algoritma tidak dapat diverifikasi dengan algoritma lain, dan setiap modifikasi pada token terdeteksi dengan baik (algoritma di sini adalah RSA/HMAC SHA-3). Kedua, HMAC SHA-3 menunjukkan performa yang lebih unggul dibandingkan RSA dalam hal waktu pembuatan dan verifikasi token. Pengujian *benchmarking* menunjukkan bahwa HMAC SHA-3 jauh lebih cepat dalam pembuatan dan verifikasi token dibandingkan RSA. Ketiga, HMAC SHA-3 lebih efisien dalam menangani *payload* besar dan beban tinggi. Waktu pembuatan dan verifikasi token meningkat seiring dengan bertambahnya ukuran *payload*, tetapi HMAC SHA-3 tetap lebih cepat dibandingkan RSA, terutama untuk *payload* besar. Keempat, dalam *Stress Test*, HMAC SHA-3 menunjukkan ketahanan yang lebih baik dibandingkan RSA.

Berdasarkan kesimpulan tersebut, beberapa saran dapat diberikan untuk implementasi dan penggunaan JWT. Pertama, disarankan untuk menggunakan HMAC SHA-3 dalam aplikasi yang membutuhkan performa tinggi dan efisiensi, terutama untuk skenario dengan beban tinggi atau ukuran *payload* besar. HMAC SHA-3 menawarkan waktu pembuatan dan verifikasi token yang lebih cepat sehingga ideal untuk sistem dengan kebutuhan respons cepat. RSA disarankan untuk digunakan dalam aplikasi yang memerlukan tingkat keamanan yang sangat tinggi, terutama dalam konteks di mana enkripsi kunci publik/pribadi memberikan keuntungan tambahan. Namun, perlu diperhatikan bahwa RSA memiliki waktu pembuatan dan verifikasi yang lebih lambat sehingga kurang cocok untuk skenario dengan beban tinggi. Kedua, kombinasi penggunaan RSA dan HMAC SHA-3 dapat dipertimbangkan dalam beberapa kasus, misalnya, RSA dapat digunakan untuk menandatangani token yang sangat sensitif, sementara HMAC SHA-3 dapat digunakan untuk token dengan kebutuhan performa tinggi. Ketiga, untuk meminimalkan dampak ukuran *payload* terhadap performa, disarankan untuk mengoptimalkan isi *payload* token JWT dengan menghindari penyimpanan data yang tidak perlu di dalam token dan mempertimbangkan untuk menyimpan data besar di tempat lain yang lebih efisien. Terakhir, selalu lakukan *monitoring* dan evaluasi berkala terhadap performa dan keamanan sistem yang menggunakan JWT, karena teknologi dan kebutuhan bisnis dapat berubah sehingga penting untuk terus menyesuaikan implementasi JWT dengan praktik terbaik terkini. Dengan mengikuti saran-saran ini, implementasi JWT menggunakan RSA dan/atau HMAC SHA-3 dapat dioptimalkan untuk memenuhi kebutuhan keamanan dan performa aplikasi secara efektif.

VIDEO LINK AT YOUTUBE

https://youtu.be/WfDrPEiBjjw

SOURCE CODE

https://github.com/gibranfsh/JWT-benchmark-using-HMAC-SHA-3-and-RSA

ACKNOWLEDGMENT

Dengan ini, Saya mengucapkan rasa syukur yang luar biasa kepada Tuhan Yang Maha Esa atas karunia-Nya sehingga Saya dapat menyelesaikan makalah berjudul "Evaluasi Kinerja dan Keamanan Penggunaan Algoritma Kriptografi HMAC SHA-3 dan RSA dalam Implementasi JSON Web Token (JWT)". Ucapan terima kasih juga Saya sampaikan kepada teman-teman mahasiswa II4031-Kriptografi dan Koding, Semester II Tahun 2023/2024, yang telah memberikan bantuan selama kegiatan belajar mengajar mata kuliah ini. Secara khusus, Saya ingin menyampaikan terima kasih kepada Bapak Dr. Ir. Rinaldi Munir, M.T., atas bimbingan dan ilmu yang telah beliau berikan selama mengajar mata kuliah ini. Semoga hasil makalah ini dapat menjadi sumber inspirasi bagi pihak lain yang tertarik untuk mengeksplorasi dan mengembangkan topik serupa.

References

[1] R. Munir, "Algoritma RSA" [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi-dan-Koding/2023-2024/11-Algoritma-RSA-2024.pdf. [Accessed: 05-June-2024].

[2] R. Munir, "Fungsi Hash" [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi-dan Koding/2023-2024/14-Fungsi-hash-2024.pdf. [Accessed: 05-June-2024].

[3] Latif, Muhammad & Jacinto, H. & Daoud, Luka & Rafla, Nader. (2018). Optimization of a Quantum-Secure Sponge-Based Hash Message Authentication Protocol. 10.1109/MWSCAS.2018.8623880. [Accessed: 05-June-2024].

[4] R. Munir, "SHA-3 (Keccak)" [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi-dan Koding/2023-2024/16-Fungsi-hash-SHA-3-2024.pdf. [Accessed: 07-June-2024].

[5] A. Anand, "Breaking down : SHA-3 algorithm," Medium, 13-Jan-2020. [Online]. Available: https://infosecwriteups.com/breaking-down-sha-3-algorithm-70fe25e125b6. [Accessed: 07-June-2024].

[6] C.Paar, J. Pelzl SHA-3, and The Hash Function Keccak, Springer, An extension chapter for Understanding Cryptography — A Textbook for Students and Practitioners, (2010). pp. 1–17. [Accessed: 07-June-2024].

[7] Park, BoSun & Song, JinGyo & Seo, Seog. (2020). Efficient Implementation of a Crypto Library Using Web Assembly. Electronics. 9. 1839. 10.3390/electronics9111839. [Accessed: 07-June-2024].

[8] "HMAC (Hash-based Message Authentication Codes) Definition," Okta. [Online]. Available: https://www.okta.com/identity-101/hmac/. [Accessed: 07-June-2024].

[9] "What is token-based authentication?," Okta. [Online]. Available: https://www.okta.com/identity-101/what-is-token-based-authentication/. [Accessed: 07-June-2024].

[10] "What is an authentication token?," Fortinet. [Online]. Available: https://www.fortinet.com/resources/cyberglossary/authentication-token [Accessed: 07-June-2024].

[11] auth0.com, "JSON web tokens introduction" JSON Web Token Introduction. [Online]. Available:https://jwt.io/introduction. [Accessed: 07-June-2024].

[12] R. Gunawan and A. Rahmatulloh, "JSON Web Token (JWT) untuk Authentication pada Interoperabilitas Arsitektur berbasis RESTful Web Service," JEPIN (Jurnal Edukasi dan Penelitian Informatika), vol. 5, no. 1, Apr. 2019. [Accessed: 07-June-2024].

[13] Mertech, "JSON Web Token (JWT) Support Comes to Thriftly," Mertech. [Online]. Available: https://www.mertech.com/blog/json-web-token-jwt-support-comes-to-thriftly. [Accessed: 09-June-2024].

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.

Jakarta, 12 Juni 2024

Gibran Fasha Ghazanfar, 18221069