Implementasi SHA-3 (Keccak) dan RSA untuk Enkripsi dan Keamanan Key-Value pada Apache Kafka

Abraham Megantoro Samudra - 18221123
Program Studi Sistem dan Teknologi Informasi
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): abrahamsamudra6@gmail.com

***Abstract*—Di era digital saat ini, pengelolaan dan analisis data dalam jumlah besar menjadi sangat penting bagi banyak organisasi. Apache Kafka, sebuah platform distribusi streaming, memungkinkan pemrosesan data real-time dengan volume besar. Dikembangkan oleh LinkedIn dan kini menjadi proyek *open-source* di bawah Apache Software Foundation, Kafka telah menjadi solusi utama untuk event streaming, digunakan oleh berbagai industri. Namun, popularitas Kafka juga membawa tantangan keamanan. Kerentanan pada platform ini dapat mengancam integritas, kerahasiaan, dan ketersediaan data. Berdasarkan data IT Governance, pada tahun 2024 terjadi 35,9 miliar pelanggaran data. Mengingat risiko ini, keamanan data menjadi sangat penting. Penelitian ini membandingkan keamanan Apache Kafka dengan dan tanpa kriptografi. Jurnal ini mencakup tinjauan pustaka, konfigurasi environment, analisis hasil pengiriman data dengan dan tanpa kriptografi, serta rekomendasi peningkatan keamanan. Kesimpulan dan panduan praktis untuk implementasi keamanan di lingkungan produksi Kafka juga disertakan. Tujuan penelitian ini adalah memberikan pemahaman komprehensif tentang penggunaan kriptografi dalam pengiriman data menggunakan Apache Kafka, sehingga profesional TI dan pengembang dapat menggunakan platform ini dengan lebih aman dan efektif. (*Abstract*)**

Keywords—Apache Kafka; Keccak; RSA; kriptografi; event-streaming; enkripsi; WireShark (key words)

I. Latar Belakang

Di era digital yang semakin maju, pengelolaan dan analisis data dalam jumlah besar menjadi sangat penting bagi banyak organisasi. Data yang dikumpulkan secara terus-menerus dari berbagai sumber memerlukan pemrosesan yang efisien dan real-time untuk mendukung pengambilan keputusan yang cepat dan akurat. Salah satu teknologi kunci yang memungkinkan hal ini adalah Apache Kafka, sebuah platform distribusi streaming yang dirancang untuk menangani aliran data dengan volume besar dalam waktu nyata.

Apache Kafka awalnya dikembangkan oleh LinkedIn dan kemudian menjadi proyek open source di bawah naungan Apache Software Foundation [1]. Kafka telah berkembang menjadi salah satu solusi terkemuka untuk event streaming, digunakan oleh berbagai perusahaan di berbagai industri untuk menangani data real-time, dari pengumpulan hingga pemrosesan dan distribusi.

Namun, dengan popularitas dan penggunaan yang luas, Apache Kafka juga menghadapi berbagai tantangan, terutama dalam hal keamanan. Kerentanan keamanan pada platform ini dapat berpotensi mengancam integritas, kerahasiaan, dan ketersediaan data yang diproses. Menurut data yang diterbitkan oleh IT Governance, pada tahun 2024 sudah terdapat 35.900.145.035 *records* yang mengalami pelanggaran data dengan rata-rata setiap bulan sebanyak 8.975.036.235 *records* [2]. Hal ini tentunya menyebabkan kerugian materiil dan imateriil yang sangat besar baik untuk perusahaan maupun individu. Dengan taruhan yang setinggi ini, sangat penting bahwa pengamanan data ditempatkan di garis depan penelitian. Oleh karena itu, memahami dan mengatasi masalah keamanan dalam Apache Kafka menjadi sangat penting untuk memastikan operasional yang aman dan andal.

Dalam jurnal ini, penulis akan membandingkan keamanan yang ada pada Apache Kafka jika dengan dan tanpa kriptografi. Jurnal ini dibagi menjadi beberapa bagian utama, dimulai dengan tinjauan pustaka yang mencakup pengenalan Apache Kafka beserta arsitekturnya dan konsep event streaming hingga Messaging Protocol. Bagian berikutnya akan melakukan konfigurasi *environment* untuk melakukan analisis. Kemudian, akan dibahas hasil analisis dan perbedaan pengiriman data dengan Apache kafka dengan dan tanpa menggunakan kriptografi, diikuti dengan rekomendasi untuk meningkatkan keamanan. Bagian akhir akan memberikan Kesimpulan serta panduan praktis untuk implementasi keamanan dalam lingkungan Kafka yang digunakan di produksi.

Tujuan dari makalah ini adalah untuk memberikan pemahaman yang komprehensif mengenai penggunaan kriptografi pada pengiriman data *key-value* menggunakan Apache Kafka. Dengan demikian, diharapkan bahwa para profesional TI dan pengembang yang menggunakan Kafka dapat memanfaatkan platform ini dengan lebih aman dan efektif, meminimalkan risiko yang terkait dengan serangan siber dan menjaga integritas data mereka.

II. TINJAUAN PUSTAKA

A. Algoritma RSA

RSA (Rivest-Shamir—Adleman) merupakan algoritma kunci publik yang paling dikenal saat ini. Algoritma RSA dibuat oleh tiga peneliti dari MIT (Massachusetts Institute of Technology), yaitu Ronald Rivest, Adi Shamir, dan Leonard Adleman pada tahun 1976. RSA memiliki beberapa properti dalam pembuatan algoritmanya.

1) *p dan q bilangan prima (RAHASIA)*
2) *n = p x 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 = gcd*
5) *d → kunci dekripsi (RAHASIA) dihitung dari d ≡ e -1 mod ($\phi(n)$)*
6) *m → plainteks (RAHASIA)*
7) *c → cipherteks (TIDAK RAHASIA)*

Dari properti – properti tersebut, akan dilakukan prosedur pembangkitan sepasang kunci dimana kunci publik adalah pasangan (e, n) dan kunci privat adalah pasangan (d, n). Fungsi enkripsi dan dekripsi dari Algoritma RSA dapat dilihat pada Eq. (1). dan Eq. (2).

$$c = m^e \bmod n \quad (1)$$

$$m = c^d \bmod n \quad (2)$$

B. Algoritma SHA-3 (Keccak)

Nama Keccak berasal dari tari Kecak, tarian asal Bali, Indonesia. Keccak merupakan pemenang kompetisi pembuatan fungsi *hash* baru (SHA-3) yang diselenggarakan oleh National Institute of Standards and Technology (NIST). Keccak memanfaatkan *sponge construction* untuk menyerap dan kemudian memeras *message digest.*

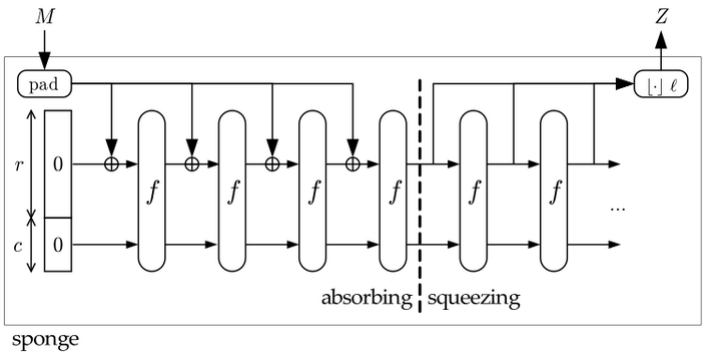

Fig. 1. Algoritma SHA-3 Keccak

Misalkan panjang *digest* yang diinginkan adalah d bit, panjang setiap blok adalah r bit. Pertama, pesan M ditambah dengan *padding* menjadi string P sehingga habis dibagi dengan r. Selanjutnya, P dipotong menjadi blok-blok Pi berukuran r-bit. Setelah itu, b bit dari peubah status (state) S diinisialisasi menjadi nol dan konstruksi spons berlangsung dalam dua fase, yaitu penyerapan dan pemerasan.

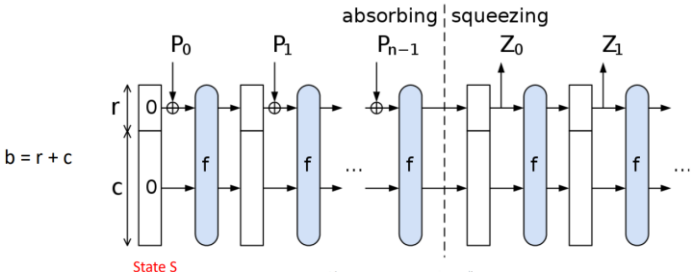

Fig. 2. Algoritma SHA-3 Keccak

1) *Fase Penyerapan*

Setiap blok Pi berukuran r bit akan di-XOR dengan r bit pertama dari state S, lalu hasilnya akan dimasukkan ke dalam fungsi permutasi f untuk menghasilkan state S yang baru. Setelah semua blok selesai diproses, konstruksi spons lanjut ke fase pemerasan.

2) *Fase Pemerasan*

Message digest akan disimpan di dalam Z. Setelah itu, Z diinisialisasi dengan *string* kosong. Selama panjang Z belum sama dengan d, r bit pertama dari state S disambungkan ke Z. Lalu jika panjang Z masih belum sama dengan d, masukkan ke dalam fungsi permutasi f untuk menghasilkan state S yang baru.

C. Apa itu Apache Kafka?

Apache Kafka adalah platform distribusi *event streaming* bersifat *open-source* yang dirancang untuk menangani aliran data dengan volume besar dalam waktu nyata. Apache Kafka dikembangkan oleh Apache Software Foundation menggunakan bahasa pemrograman Scala. Dalam beberapa tahun terakhir, penggunaan Kafka telah meningkat secara signifikan di berbagai industri, termasuk perbankan, telekomunikasi, dan teknologi informasi. Menurut data dari 6Sense, saat ini Apache Kafka menempati peringkat 1 sebagai platform Queueing, Messaging, and Background Processing dengan *market share* sebesar 39.84%, mengalahkan kompetitornya yaitu RabbitMQ, IBM MQ, dan Apache ActiveMQ. [3]

D. Event Streaming

1) *Apa itu Event Streaming?*

Event streaming adalah paradigma pengolahan data di mana data diperlakukan sebagai aliran peristiwa (*event*, bisa disebut juga dengan *message*) yang terus-menerus dan *real-time*. Setiap *event* mewakili perubahan atau aksi yang terjadi dalam sistem, seperti transaksi keuangan, pembaruan sensor IoT, klik pengguna di situs web, atau perubahan status pada aplikasi [4]. Secara konsep, satu *event* memiliki *topic* (nama topic yang akan menyimpan message), *partition* (nomor partisi untuk menyimpan *message), key*, *value*, dan *timestamp*, serta *metadata headers* atau informasi tambahan untuk *message* yang bersifat opsional. Berikut merupakan contoh dari sebuah event:

- Event topic : “Payment”

- Event partition : 1
- Event key : “Alice”
- Event value : “Mentransfer uang Rp20.000 ke Bob”
- Event timestamp : “Jun. 01, 2024 at 03:14 p.m.”

Dalam konteks ini, *event streaming* memungkinkan organisasi untuk mengumpulkan data secara terus-menerus, memproses dan menganalisis data secara *real-time*, dan mendistribusikan hasil analisis ke berbagai aplikasi atau sistem lainnya.

2) Mengapa Event Streaming Penting?

Event Streaming sudah menjadi komponen penting dalam arsitektur pengolahan data *modern* karena beberapa alasan berikut :

- ***Real-Time Data Processing*** **:** Kemampuan untuk memproses dan bertindak setelah data diterima, memungkinkan keputusan bisnis yang lebih cepat dan responsive
- ***Scalability*** **:** Sistem *event streaming* dirancang untuk menangani volume data yang besar dengan latensi yang rendah, membuat Apache Kafka menjadi keputusan yang ideal untuk digunakan perusahaan skala besar.
- ***Reliability*** **:** Dengan adanya mekanisme replikasi data dan toleransi kesalahan, *event streaming* memastikan data tetap tersedia dan konsisten bahkan dalam kondisi kegagalan komponen sistem.
- ***Integration*** **:** *Event streaming* memastikan integrasi yang mulus antara berbagai sistem dan aplikasi, memfasilitasi aliran data yang kontinu di seluruh ekosistem teknologi.

3) Event Streaming Use Cases

Saat ini *event streaming* sudah digunakan di berbagai jenis industri dan organisasi. Beberapa contoh penerapan *event streaming* saat ini adalah sebagai berikut :

- Memproses pembayaran dan transaksi secara *real-time*, seperti pembelian saham, bank, dan asuransi.
- Melakukan *tracking* kendaraan untuk pengiriman barang.
- Secara terus-menerus menangkap dan menganalisis sensor data dari perangkat Internet of Things (IoT) atau perangkat lainnya, seperti perangkat pada pabrik.
- Menyimpan dan memberikan notifikasi atau reaksi lainnya kepada interaksi dari *customers* dan pesanan, seperti saat memesan hotel atau travel.
- Memantau pasien di rumah sakit serta memprediksi perubahan kondisi pasien untuk memastikan perawatan diberikan dalam waktu yang tepat.
- Menghubungkan, menyimpan, dan menyediakan data yang dihasilkan dari berbagai divisi dalam sebuah *company.*
- Sebagai fondasi untuk interaksi pada arsitektur *microservices* dan arsitektur yang menggunakan konsep *event-driven*

4) Event Streaming Pada Apache Kafka

Kafka mengombinasikan tiga kemampuan utama sehingga pengguna dapat meingimplementasikan *use-case* untuk *event streaming* secara menyeluruh dalam satu solusi yang sudah teruji :

- Untuk ***publish*** *(write)* dan ***subscribe*** *(read)* ke kumpulan *events*, termasuk *import/export* data dari sistem lain secara terus menerus.
- Untuk **menyimpan** kumpulan *events* secara tahan lama dan andal selama apapun yang pengguna inginkan.
- Untuk **memproses** kumpulan *events* sesaat setelah muncul atau secara retrospektif (berdasarkan *events* yang sudah terjadi).

Fungsionalitas ini disediakan Apache Kafka secara terdistribusi, skalabilitas yang tinggi, *elastic, fault-tolerant*, dan aman. Kafka dapat di-*deploy* di perangkat keras, *virtual machines*, kontainer, *on-premises*, begitu juga di *cloud.* Pengguna dapat memilih untuk mengelola Kafka di lingkungan sendiri maupun menggunakan layanan dari berbagai *vendors* secara menyeluruh.

E. Publish Subscribe

Saat membuat sebuah aplikasi, pasti akan melakukan komunikasi dengan aplikasi lainnya. Contoh paling sederhana adalah aplikasi yang berkomunikasi dengan aplikasi *database.* Komunikasi ini disebut RPC (Remote Procedure Call). Keuntungan menggunakan RPC adalah komunikasi bisa dilakukan secara *synchronous* dan *real-time.* [5]

Contoh kasus adalah saat membuat platform *e-commerce* dimana *platform* tersebut memiliki banyak aplikasi seperti aplikasi *product*, promo, *shopping cart*, *order*, *logistic*, dan *payment.* Untuk mendapatkan informasi produk dan promo yang sedang berjalan, *shopping cart* harus berkomunikasi dengan aplikasi *product* dan promo. Setelah barang di *shopping cart* akan dibeli, maka *shopping cart* mengirim semua data *product* ke aplikasi *order* yang nantinya akan diproses untuk ke aplikasi *logistic* untuk menghitung biaya pengiriman serta aplikasi *payment* untuk diproses pembayarannya. Komunikasi seperti ini tentu membutuhkan struktur yang kompleks dan tidak memiliki skalabilitas yang baik, terlebih lagi semisal *owner* ingin menerapkan aplikasi *fraud detection*, maka mekanisme pengiriman data ke aplikasi *fraud detection* harus diintegrasikan dengan seluruh aplikasi lainnya yang ada kembali.

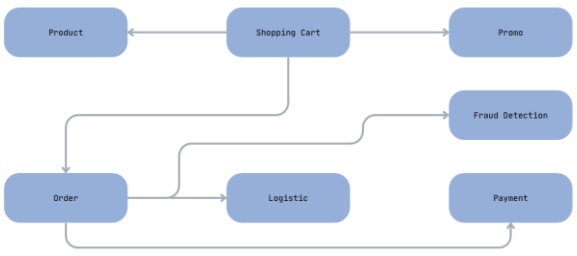

Fig. 3. Diagram RPC (Remote Procedure Call)

Selain RPC, terdapat mekanisme komunikasi Bernama Messaging atau lebih dikenal dengan Publish/Subscribe.

Mekanisme ini berbeda dengan RPC, Dimana pengirim tidak menentukan siapa yang menerima data, melainkan pengirim akan mengirim data ke perantara yang disebut dengan Message Broker. Seluruh penerima data akan mengambil data langsung dari Message Broker.

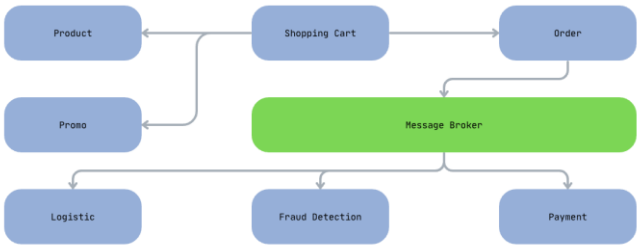

Fig. 4. Diagram Messaging Protocol

F. *Cara Kerja Apache Kafka*

Dalam Apache Kafka, pengirim data disebut dengan *producers* dan penerima data disebut dengan *consumers.* Sementara itu, kegiatan mengirim data atau *event (write)* disebut *publish,* dan penerimaan data (*read and process)* disebut *subscribe.* Producers dan consumers sepenuhnya terpisah dan tidak terhubung satu sama lain Dimana konsep ini yang memungkinkan Kafka memiliki skalabilitas yang tinggi.

Events dikelola dan disimpan dengan aman di tempat penyimpanan yang disebut *topics.* Secara sederhana, topic mirip dengan sebuah *folder* dalam sebuah *filesystem* dan *events* adalah *files* dalam *folder* tersebut. Topics pada Kafka dapat memiliki nol, satu, atau lebih *producers* maupun *consumers. Events* di dalam sebuah topic dapat di-read sebanyak apapun yang diinginkan karena *events* tidak langsung dihapus setelah digunakan oleh *consumers.* Akan tetapi, *developers* menentukan seberapa lama Kafka dapat mempertahankan *events* dengan mengonfigurasikan masing-masing topic, setelah itu event yang lama akan dibuang. Performa dari Kafka pada dasarnya adalah konstan sehubungan dengan *data size*, sehingga menyimpan data dalam jangka Waktu yang lama bukanlah masalah.

Topics dipartisi sehingga dalam sebuah topik tersebar beberapa "*buckets"* yang terletak di *brokers* yang berbeda. Distribusi ini penting untuk skalabilitas Kafka karena dengan adanya distribusi ini, aplikasi dapat *publish* maupun *subscribe* ke atau dari berbagai *brokers* dalam waktu yang sama. Saat sebuah *event* dikirim ke *topic, event* ditambahkan ke salah satu partisi dalam topik. *Events* dengan *event key* yang sama ditulis ke dalam partisi yang sama, dan *consumers* dari sebuah paritsi topic selalu membaca *events* sesuai dengan urutan *event* tersebut di-*write.*

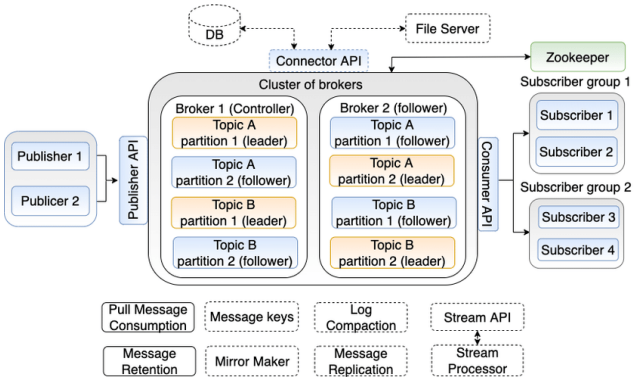

Fig. 5. Arsitektur Apache Kafka [6]

Saat *consumer* membaca data dari *topic*, maka *consumer* dapat menentukan *consumer group (subscriber group)* yang digunakan. *Consumer group* memungkinkan data untuk tidak dikirim berkali-kali ke semua *consumer*, melainkan hanya sekali *consumer group.* Untuk membuat data menjadi *fault-tolerant* dan *highly available*, setiap topic dapat direplika sehingga ada beberapa *brokers* yang mempunyai *copy* dari data. Replikasi ini dijalankan di partisi-partisi dalam *topic*.

G. *Tantangan Keamanan pada Apache Kafka*

Dengan semakin banyaknya data yang mengalir melalui sistem event streaming, keamanan menjadi aspek kritis yang harus diperhatikan. Beberapa tantangan utama dalam keamanan event streaming meliputi:

- **Autentikasi dan Otorisasi :** Memastikan hanya pengguna dan sistem yang berwenang yang dapat mengakses dan mengirim data.
- **Enkripsi :** Melindungi data dari intersepsi selama transmisi antara berbagai komponen dalam arsitektur event streaming.
- **Integritas Data :** Memastikan data tidak diubah tanpa otorisasi selama proses pengiriman dan penyimpanan.
- **Manajemen Keamanan :** Mengelola konfigurasi dan kebijakan keamanan secara efektif untuk mencegah dan mendeteksi serangan.

III. METODE PENELITIAN

Penelitian ini bertujuan untuk menganalisis kerentanan keamanan pada Apache Kafka, khususnya terkait dengan mekanisme enkripsi data yang dikirimkan oleh *producer* untuk diterima oleh *consumer*. Metode penelitian yang digunakan melibatkan beberapa langkah utama, yaitu konfigurasi sistem, penulisan kode untuk produser dan consumer dengan dan tanpa menggunakan kriptografi, serta penggunaan Wireshark untuk menangkap dan menganalisis lalu lintas jaringan. Berikut adalah rincian langkah-langkah yang dilakukan dalam penelitian ini.

A. *Konfigurasi Apache Kafka*

1. Instalasi Apache Kafka melalui *website* resmi Apache Kafka (https://kafka.apache.org/downloads)
2. Konfigurasi server.properties di dalam folder /config/kraft dengan menggunakan PLAINTEXT.

```
listeners=PLAINTEXT://:9092, CONTROLLER://:9093

inter.broker.listener.name=PLAINTEXT

advertised.listeners=PLAINTEXT://localhost:9092
```

3. Jalankan Apache Kafka server sesuai dengan *Operating System* dengan perintah berikut

```
# Windows OS
./bin/windows/kafka-server-start.bat
config/kraft/server.properties

# Linux or Mac OS
./bin/kafka-server-start.sh
config/kraft/server.properties
```

B. Konfigurasi Apache Kafka menggunakan Java Spring Boot

1. Buka *website* https://start.spring.io/
2. Masukkan konfigurasi seperti berikut, tambahkan *dependency* Spring for Apache Kafka

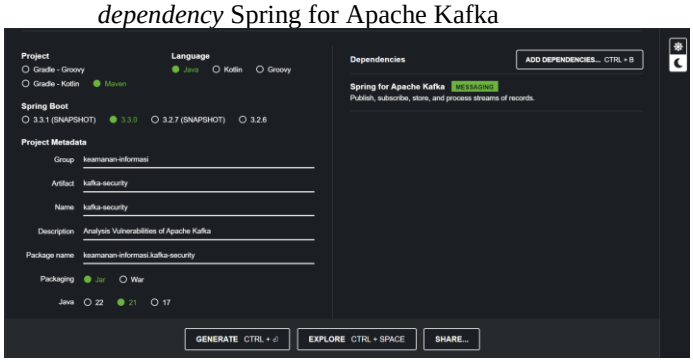

Fig. 6. Konfigurasi Spring Boot

3. Klik Generate atau tekan CTRL + Enter
4. Simpan di tempat yang diinginkan lalu buka menggunakan Visuall Studio Code

C. Penulisan Kode untuk Producers dan Consumers

Untuk mengirim dan menerima pesan, masing-masing akan dibuat kode Java untuk produser dan consumer tanpa penerapan kriptografi dan dengan penerapan kriptografi. Detail kode dapat dilihat lebih detail pada *repository* GitHub.

1. Tanpa Kriptografi
    a. Kode *Producers* (KafkaPlainProducer.java)

```java
package keamanan_informasi.kafka_security;

import org.apache.kafka.clients.producer.*;

import java.util.Properties;

public class KafkaPlainProducer {
    public static void main(String[] args) {
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringSerializer");
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringSerializer");

        Producer<String, String> producer = new KafkaProducer<>(props);
        String topic = "your-topic";

        try {
            // Mengirim pesan dengan value "test"
            String key = "key-test";
            String value = "test";
            producer.send(new ProducerRecord<>(topic, key, value), new Callback() {
                public void onCompletion(RecordMetadata metadata, Exception exception) {
                    if (exception != null) {
                        exception.printStackTrace();
                    } else {
                        System.out.printf("Produced record to topic %s partition [%d] @ offset %d%n",
                                metadata.topic(), metadata.partition(), metadata.offset());
                    }
                }
            });
        } finally {
            producer.close();
        }
    }
}
```

Fig. 7. File KafkaPlainProducer.java

b. Kode *Consumers* (KafkaPlainConsumer.java)

```java
package keamanan_informasi.kafka_security;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.ConsumerRecord;

import java.util.Collections;
import java.util.Properties;
import java.time.Duration;

public class KafkaPlainConsumer {
    public static void main(String[] args) {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put("group.id", "test-group");
        props.put("auto.offset.reset", "earliest");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringDeserializer");
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringDeserializer");

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
        String topic = "your-topic";

        try {
            consumer.subscribe(Collections.singletonList(topic));

            while (true) {
                ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
                for (ConsumerRecord<String, String> record : records) {
                    System.out.printf("Consumed record from topic %s partition [%d] @ offset %d with key
%s and value %s%n",
                            record.topic(), record.partition(), record.offset(), record.key(),
record.value());
                }
            }
        } finally {
            consumer.close();
        }
    }
}
```

Fig. 8. File KafkaPlainConsumer.java

2. Dengan Kriptografi (SHA-3 dan RSA)
    a. Kode *Producers* (KafkaCryptoProducers.java)

```java
package keamanan_informasi.kafka_security;

import org.apache.kafka.clients.producer.*;
import java.math.BigInteger;
import java.nio.charset.StandardCharsets;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Base64;
import java.util.Properties;

public class KafkaCryptoProducer {

    private static final BigInteger N = new BigInteger("187");
    private static final BigInteger E = new BigInteger("3");

    public static void main(String[] args) {
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringSerializer");
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringSerializer");

        Producer<String, String> producer = new KafkaProducer<>(props);
        String topic = "your-topic";

        try {
            String value = "test";
            String encryptedValue = encryptRSA(value);
            String key = calculateSHA3(value);

            producer.send(new ProducerRecord<>(topic, key, encryptedValue), new Callback() {
                public void onCompletion(RecordMetadata metadata, Exception exception) {
                    if (exception != null) {
                        exception.printStackTrace();
                    } else {
                        System.out.println("Encrypted Value: " + encryptedValue);
                        System.out.printf("Produced record to topic %s partition [%d] @ offset %d%n",
                                metadata.topic(), metadata.partition(), metadata.offset());
                    }
                }
            });
        } finally {
            producer.close();
        }
    }

    private static String calculateSHA3(String value) {
        try {
            MessageDigest digest = MessageDigest.getInstance("SHA3-256");
            byte[] hash = digest.digest(value.getBytes(StandardCharsets.UTF_8));
            return Base64.getEncoder().encodeToString(hash);
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
    }

    private static String encryptRSA(String data) {
        StringBuilder encryptedData = new StringBuilder();
        for (char ch : data.toCharArray()) {
            BigInteger ascii = BigInteger.valueOf((int) ch);
            BigInteger encryptedAscii = ascii.modPow(E, N);
            encryptedData.append(encryptedAscii.toString()).append(" ");
        }
        return encryptedData.toString().trim();
    }
}
```

Fig. 9. File KafkaCryptoProducer.java

b. Kode *Consumers* (KafkaCryptoConsumers.java)

```java
package keamanan_informasi.kafka_security;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.bouncycastle.jce.provider.BouncyCastleProvider;

import java.math.BigInteger;
import java.nio.charset.StandardCharsets;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.Security;
import java.time.Duration;
import java.util.Base64;
import java.util.Collections;
import java.util.Properties;

public class KafkaCryptoConsumer {

    private static final BigInteger N = new BigInteger("187");
    private static final BigInteger D = new BigInteger("267");

    public static void main(String[] args) throws NoSuchAlgorithmException {
        Security.addProvider(new BouncyCastleProvider());

        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put("group.id", "test-group");
        props.put("auto.offset.reset", "earliest");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
                "org.apache.kafka.common.serialization.StringDeserializer");
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
                "org.apache.kafka.common.serialization.StringDeserializer");

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
        String topic = "your-topic";

        try {
            consumer.subscribe(Collections.singletonList(topic));

            while (true) {
                ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
                for (ConsumerRecord<String, String> record : records) {
                    String receivedHash = record.key();
                    String encryptedValue = record.value();

                    String decryptedValue = decryptRSA(encryptedValue);
                    String calculatedHash = calculateSHA3("test");

                    if (calculatedHash.equals(receivedHash)) {
                        System.out.println("Encrypted Value: " + encryptedValue);
                        System.out.println("Message is authentic: " + decryptedValue);
                    } else {
                        System.out.println("Message is corrupted or tampered with.");
                    }
                }
            }
        } finally {
            consumer.close();
        }
    }

    private static String calculateSHA3(String data) {
        try {
            MessageDigest digest = MessageDigest.getInstance("SHA3-256");
            byte[] hashBytes = digest.digest(data.getBytes(StandardCharsets.UTF_8));
            return Base64.getEncoder().encodeToString(hashBytes);
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
    }

    private static String decryptRSA(String encryptedText) {
        String[] encryptedValues = encryptedText.split(" ");
        StringBuilder decryptedMessage = new StringBuilder();

        for (String value : encryptedValues) {
            BigInteger encryptedAscii = new BigInteger(value);
            BigInteger decryptedAscii = encryptedAscii.modPow(D, N);
            decryptedMessage.append((char) decryptedAscii.intValue());
        }

        return decryptedMessage.toString();
    }
}
```

Fig. 10. File KafkaCryptoConsumer.java

D. Instalasi dan Konfigurasi Wireshark

1. Install Wireshark melalui *website* resminya (https://www.wireshark.org/download.html)
2. Jalankan Wireshark
3. Pilih Adapter for Loopback Traffic Capture sebagai *interface* jaringan

E. Konfigurasi Private Key dan Public Key RSA

Untuk konfigurasi *private key* dan *public key* RSA, penulis menggunakan p = 11 dan q = 17 dimana akan menghasilkan pasangan kunci publik (267, 187) dan kunci privat (3, 187).

IV. PEMBAHASAN

A. *Tanpa Kriptografi*

Setelah melakukan konfigurasi, server Kafka akan dijalankan dengan perintah yang sudah ditetapkan pada bagian konfigurasi. Selain itu, *consumers* (KafkaPlainConsumer.java) dan *producers* (KafkaPlainProducer.java) juga dijalankan menggunakan *Compiler* dari Visual Studio Code. Hasil *compile* dari kedua file adalah seperti pada Fig. 11 dan Fig. 12. Gambar tersebut membuktikan bahwa saat producers dijalankan, consumers menerima data yang dikirim oleh producers.

```
Consumed record from topic your-topic partition [0] @ offset 11 with key key-test and value test
```

Fig. 11. Hasil compile file KafkaPlainConsumer.java

```
Produced record to topic your-topic partition [0] @ offset 11
```

Fig. 12. Hasil compile file KafkaPlainProducer.java

Selama kedua file dijalankan, dilakukan juga packet capture menggunakan Wireshark. Setelah dilakukan analisis *packet-packet* yang dikirim antara klien dan broker, dapat dilihat bahwa data yang dikirim adalah data dalam bentuk teks biasa dan tidak terenkripsi seperti Pada Fig. 13.

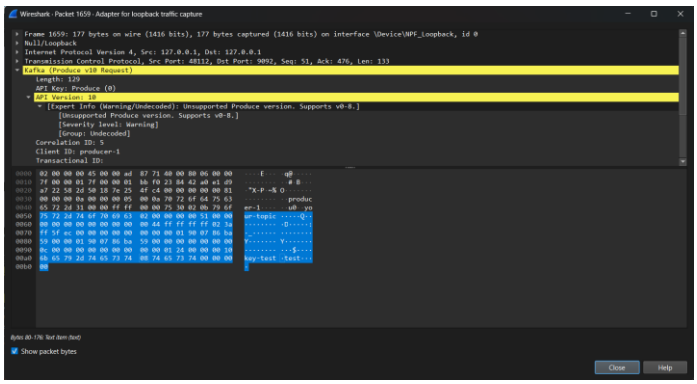

Fig. 13. Packet Analysis Apache Kafka tanpa Kriptografi

Dapat dilihat pada bagian *packet bytes*, bahwa data yang dikirimkan memiliki *key* dengan nama "*key-test"* dan *value* dengan nama "*test"* tanpa disertai adanya kriptografi. Hal ini meningkatkan kemungkinan akan adanya *eavesdropping* karena teks disajikan secara plain dan dapat dikenali oleh pihak luar.

B. *Dengan Kriptografi*

Melakukan pengecekan data yang dikirimkan dengan kriptografi menggunakan langkah-langkah yang sama dengan pengecekan tanpa kriptografi. Fig. 14. dan Fig. 15. menunjukkan hasil *compile* file KafkaCryptoConsumer.java serta file KafkaCryptoProducer.java.

```
Encrypted Value: 7 118 4 7
Message is authentic: test
```

Fig. 14. Hasil compile file KafkaCryptoConsumer.java

```
Encrypted Value: 7 118 4 7
Produced record to topic your-topic partition [0] @ offset 33
```

Fig. 15. Hasil compile file KafkaCryptoProducer.java

Setelah dilakukan analisis *packet-packet* yang dikirim antara klien dan broker kembali, dapat dilihat bahwa data yang dikirimkan adalah data yang sudah dilakukan *hash* dan sudah dienkripsi seperti pada Fig. 16.

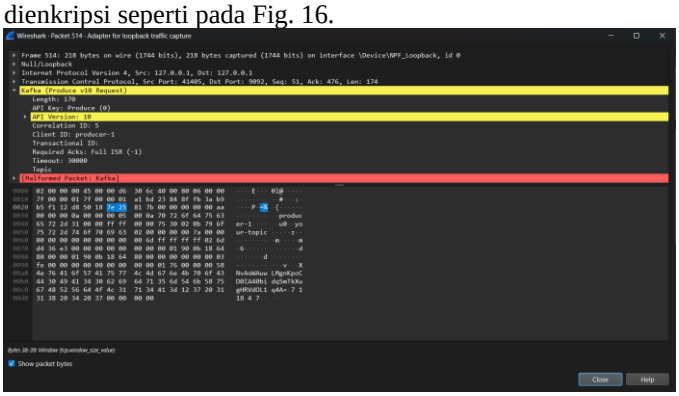

Fig. 16. Packet Analysis Apache Kafka dengan Kriptografi

Dapat dilihat pada bagian packet bytes, meskipun pada kode data yang dikirimkan memiliki key dengan nama "key-test" dan value dengan nama "test", tetapi dengan menggunakan kriptografi, data tersebut sudah menjadi lebih aman dan tidak ditampilkan secara *plain.* Hal ini tentu saja menurunkan kemungkinan adanya eavesdropping karena teks disajikan secara terenrkipsi dan tidak dapat dikenali oleh pihak luar.

V. KESIMPULAN DAN SARAN

Di era digital yang terus berkembang, pengelolaan dan analisis data dalam jumlah besar menjadi sangat penting bagi banyak organisasi. Apache Kafka telah muncul sebagai salah satu solusi terkemuka untuk menangani aliran data real-time dalam berbagai industri. Namun, dengan popularitasnya, muncul tantangan signifikan terkait dengan keamanan data yang diproses melalui platform ini. Berdasarkan penelitian dan analisis yang dilakukan, ditemukan bahwa penggunaan kriptografi, khususnya algoritma RSA dan SHA-3, dapat secara signifikan meningkatkan keamanan dalam transmisi data menggunakan Apache Kafka.

Selain melakukan enkripsi dan *hashing* pada data, keamanan data pada Apache Kafka juga dapat ditingkatkan dengan melakukan enkripsi pada protokol komunikasi dengan mengganti PLAINTEXT menjadi SSL/TLS.

```
listeners=SSL://localhost:9092

security.inter.broker.protocol=SSL

ssl.keystore.location=/path/to/keystore

ssl.keystore.password=keystore_password

ssl.key.password=key_password
```

```
ssl.truststore.location=/path/to/truststore

ssl.truststore.password=truststore_password
```

Komunikasi dari producers ke brokers hingga ke consumers jika tidak dienkripsi, dapat membocorkan data yang sensitif kepada orang yang tidak bertanggung jawab. Kesalahan yang sering kali dilakukan adalah tidak menggunakan enkripsi SSL/TLS untuk komunikasi ketiga aspek penting tersebut. Untuk mengatasi hal tersebut, tentunya perlu enkripsi SSL/TLS pada komunikasi internal dan eksternal dengan melakukan konfigurasi pada file server.properties [7].

Di samping itu, Untuk mengatasi tantangan keamanan yang diidentifikasi, disarankan langkah-langkah berikut:

- **Penggunaan Enkripsi :** Menerapkan enkripsi SSL/TLS untuk semua komunikasi antara producers, brokers, dan consumers.
- **Implementasi ACL :** Menggunakan Access Control Lists untuk membatasi akses hanya kepada entitas yang berwenang.
- **Autentikasi yang Kuat :** Menggunakan mekanisme autentikasi yang lebih aman seperti SCRAM-SHA-256.
- **Konfigurasi ZooKeeper yang Aman :** Mengimplementasikan autentikasi dan enkripsi pada ZooKeeper untuk memastikan keamanan koordinasi dan sinkronisasi dalam Kafka.
- **Monitoring dan Logging :** Menggunakan alat monitoring dan logging untuk mendeteksi aktivitas yang mencurigakan dan mengidentifikasi potensi ancaman.

Selain itu, penting untuk dicatat bahwa Apache Kafka selalu mempertimbangkan keamanan sebagai hal yang penting dalam implementasinya. Hal ini dapat dilihat dari upaya Kafka yang terus-menerus memperbarui dan memperbaiki potensi keamanannya. Daftar CVE (Common Vulnerabilities and Exposures) di situs web resmi Kafka (https://kafka.apache.org/cve-list) menunjukkan bahwa Kafka terus memantau dan menangani kerentanan keamanan secara proaktif.

Dengan menerapkan langkah-langkah ini, organisasi dapat meminimalkan risiko keamanan dan memanfaatkan Apache Kafka dengan lebih aman dan efektif. Jurnal ini diharapkan dapat memberikan panduan praktis bagi para profesional TI dan pengembang dalam mengimplementasikan Kafka di lingkungan produksi yang aman dan andal.

VIDEO LINK AT YOUTUBE *(Heading 5)*

https://youtu.be/CyXMPOUp2uE

IMPORTANT LINKS

1) Link Wireshark Packet Capture dengan kriptografi : https://drive.google.com/file/d/1a89qamBbsXEBGCH7ucUdE4T8Nzx9JqCZ/view?usp=sharing

2) Link Wireshark Packet Capture tanpa kriptografi : https://drive.google.com/file/d/1PFtew1fMET4pFVCWW_pYAKTF_PBGPlT0/view?usp=sharing

3) Link Kode Github : https://github.com/abrahammegantoro/apache-kafka-pentesting

REFERENCES

[1] Losinski, T. (2021, September). Attacking the messenger exploring the security of Big Data ... ATTACKING THE MESSENGER: EXPLORING THE SECURITY OF BIG DATA MESSENGER APACHE KAFKA. https://library.ndsu.edu/ir/bitstream/handle/10365/32736/Attacking%20the%20Messenger%20Exploring%20the%20Security%20of%20Big%20Data%20Messenger%20Apache%20Kafka.pdf?sequence=1&isAllowed=y

[2] Apache Kafka - market share, competitor insights in queueing, messaging and background processing. 6sense. (n.d.). https://6sense.com/tech/queueing-messaging-and-background-processing/apache-kafka-market-share

[3] Apache kafka. Apache Kafka. (n.d.). https://kafka.apache.org/

[4] Kurniawan, E. (n.d.). Apache Kafka Dasar. Google Slides. https://docs.google.com/presentation/d/130iELJYouuSEmHFUfEmAXi00FuvtFI9633B5Tmj7bwk/edit#slide=id.g26412031b50_0_934

[5] Kurniawan, E. (n.d.). Apache Kafka Dasar. Google Slides. https://docs.google.com/presentation/d/130iELJYouuSEmHFUfEmAXi00FuvtFI9633B5Tmj7bwk/edit#slide=id.g26412031b50_0_934

[6] Lazidis, A., Tsakos, K., & Petrakis, E. G. M. (2022). Publish–subscribe approaches for the IOT and the cloud: Functional and performance evaluation of open-source systems. Internet of Things, 19, 100538. https://doi.org/10.1016/j.iot.2022.100538

[7] Syn Cubes Community. (2023, April 24). Pentesting Apache Kafka 101 : Top 5 security misconfigurations. Pentesting Apache Kafka 101 : Top 5 Security Misconfigurations. https://www.syncubes.com/pentesting-apache-kafka-101

[8] Nandi, D. (2023, August 26). Security series: How to harden apache kafka. Medium. https://medium.com/@java.dev/security-series-how-to-harden-apache-kafka-eb57c9787cf3

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.

Bandung, 12 Juni 2024

Abraham Megantoro Samudra, 18221123