Enkripsi Ujung ke Ujung untuk File Besar Menggunakan Variasi Signal Protocol

Jesson Gosal Yo - 13519079 (*Penulis*)
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13519079@std.stei.itb.ac.id

***Abstract*—Makalah ini membahas tentang implementasi enkripsi ujung ke ujung untuk file besar menggunakan algoritma Ratchet Ganda. Dalam era pertukaran data yang semakin meningkat, perlindungan privasi dan keamanan data menjadi sangat penting. Metode enkripsi yang ada saat ini masih memiliki beberapa keterbatasan dalam mengamankan file besar yang dikirimkan melalui jaringan. Penulis mengusulkan penggunaan algoritma *double ratchet* untuk memberikan lapisan keamanan tambahan dalam pengiriman pesan berupa file. Prototipe yang dibuat sebatas menguji apakah mungkin untuk meminjam algoritma *double ratchet* yang sekarang banyak dipakai dalam pengiriman pesan ujung ke ujung di dalam *signal protocol* di dalam pengiriman file dan cara mengimplementasinya.**

Kata Kunci—ujung ke ujung, double ratchet, enkripsi, kontribusi

I. Pendahuluan

Dalam era digital yang semakin maju ini, pertukaran data dan informasi melalui jaringan menjadi hal yang umum dan penting dalam berbagai aktivitas sehari-hari. Namun, seiring dengan peningkatan pertukaran data, kekhawatiran tentang keamanan dan privasi juga semakin meningkat. Penting bagi pengguna untuk dapat mengirim dan menerima file besar melalui jaringan dengan tingkat keamanan yang tinggi agar data sensitif tidak jatuh ke tangan yang salah.

Keamanan transmisi file merujuk pada langkah-langkah yang diambil untuk melindungi integritas, kerahasiaan, dan ketersediaan data yang dikirimkan melalui jaringan. Tanpa langkah-langkah keamanan yang memadai, data yang dikirimkan dapat menjadi rentan terhadap ancaman seperti peretasan, pemalsuan, atau pencurian informasi yang dapat berdampak serius pada individu, perusahaan, atau organisasi. Salah satu ancaman yang umum dalam transmisi file adalah pengintersepsi data oleh pihak yang tidak berwenang. Ketika file dikirimkan melalui jaringan, data tersebut dapat dirampas oleh pihak ketiga yang dapat menggunakan informasi tersebut untuk keuntungan pribadi atau bahkan merugikan pihak yang terlibat. Misalnya, file bisnis yang berisi informasi rahasia atau strategi perusahaan dapat direbut oleh pesaing atau individu yang tidak bertanggung jawab.

Oleh karena itu, penting untuk melindungi transmisi file dengan menggunakan metode enkripsi yang kuat dan langkah-langkah keamanan lainnya. Enkripsi melibatkan pengubahan data menjadi bentuk yang tidak dapat dibaca oleh pihak yang tidak berwenang melalui penggunaan kunci enkripsi. Dengan menerapkan enkripsi pada file yang dikirimkan, data sensitif dapat tetap aman dan terlindungi dari ancaman selama proses transmisi.

Selain itu, upaya keamanan transmisi file juga mencakup langkah-langkah untuk memastikan integritas data, seperti penggunaan tanda tangan digital atau metode hash untuk memverifikasi keaslian file. Dengan menggunakan teknologi dan praktik keamanan yang tepat, kita dapat menjaga kerahasiaan, integritas, dan ketersediaan data yang dikirimkan melalui jaringan.

Metode enkripsi yang menjadi solusi umum dalam menjaga kerahasiaan data saat berada dalam perjalanan melalui jaringan seringkali merupakan gabungan enkripsi asimetris dan simetris. Enkripsi asimetris digunakan untuk menentukan sebuah kunci simetris yang nantinya akan digunakan seterusnya dalam enkripsi simetris data. Metode konvensional ini memiliki kelemahan apabila hubungan antara kedua pihak berlangsung lama. Apabila kunci simetris berhasil ditebak oleh penyerang di tengah-tengah waktu komunikasi maka penyerang otomatis dapat melakukan dekripsi seluruh pesan yang telah ditransmisi maupun yang akan ditransmisi. Transmisi file yang berukuran besar

Untuk mengatasi tantangan ini, penulis mengambil ide dari sebuah protokol yang kerap digunakan dalam layanan pengiriman pesan. Protokol ini disebut *signal protocol*. *Signal protocol* di dalamnya berisi satu algoritma yang menarik yaitu algoritma *double ratchet*. Secara sederhana algoritma *double ratchet* merupakan sebuah algoritma enkripsi kunci ganda yang memanfaatkan pertukaran kunci dinamis dengan bantuan X3DH (*extended triple diffie-hellman)*. Selain itu untuk setiap pesan juga dienkripsi menggunakan kunci simetris yang berbeda yang akan dibangkitkan secara independen oleh kedua belah pihak dengan kunci dinamis yang sebelumnya dipertukarkan dengan X3DH berperan sebagai *seed* agar hasil kunci yang dibangkitkan deterministik dan dapat diketahui oleh kedua pihak tanpa perlu komunikasi tambahan. Apabila X3DH tidak dilakukan maka algoritma ini memberikan *forward secrecy* karena apabila penyerang berhasil mendapatkan kunci di tengah-tengah komunikasi maka penyerang tidak dapat mendeduksi kunci-kunci sebelumnya. Akan tetapi penyerang dapat mendeduksi kunci-kunci setelahnya karena pembangkitan kunci ini deterministik relatif terhadap *seed*.

Dalam prototipe yang diusulkan dalam makalah ini, X3DH diusulkan untuk dilakukan secara periodik agar *backward*

secrecy dapat diberikan untuk sebagian kunci yang digunakan dalam enkripsi simetris setelah salah satu kunci terkompromi oleh penyerang. Prototipe yang dibuat merupakan prototipe algoritma *double ratchet* yang sederhana karena dilakukan dengan disimulasikan secara *in-memory* sehingga paket yang ditransmisikan tidak dapat hilang. Selain itu ada beberapa aspek lain yang disederhanakan untuk kemudahan implementasi dan pengujian salah satunya adalah mengganti X3DH dengan pertukaran *diffie-hellman* biasa.

II. LANDASAN TEORI

A. *Signal Protocol*

Signal Protocol adalah sebuah protokol enkripsi *end-to-end* (E2EE) yang digunakan untuk menjaga kerahasiaan dan keamanan pesan di lingkungan komunikasi digital. Protokol ini dikembangkan oleh *Open Whisper Systems* dan telah diimplementasikan dalam berbagai aplikasi populer, termasuk *Signal, WhatsApp, dan Facebook Messenger*.

Signal Protocol dirancang dengan fokus pada privasi dan keamanan komunikasi pengguna. Tujuan utamanya adalah untuk mengamankan pesan dan panggilan suara agar hanya dapat diakses oleh penerima yang dituju, sambil melindungi data dari ancaman seperti peretasan, pemalsuan, dan pemantauan oleh pihak ketiga yang tidak berwenang.

Salah satu fitur kunci dari *Signal Protocol* adalah enkripsi end-to-end. Ini berarti pesan dienkripsi pada perangkat pengirim dan hanya dapat didekripsi oleh perangkat penerima yang sah. Enkripsi *end-to-end* memastikan bahwa data tetap aman selama proses transmisi dan hanya dapat diakses oleh penerima yang memiliki kunci dekripsi yang tepat.

1. Kriptografi Kunci Publik

   Kriptografi kunci publik, juga dikenal sebagai kriptografi asimetris, adalah cabang kriptografi yang melibatkan penggunaan sepasang kunci yang berbeda untuk melakukan enkripsi dan dekripsi. Dalam kriptografi kunci publik, terdapat dua jenis kunci yang saling terkait: kunci publik dan kunci privat. Kunci publik adalah kunci yang bisa diakses semua orang dan kunci ini digunakan untuk mengenkripsi pesan oleh pengirim sebelum dikirim ke penerima. Pesan yang dienkripsi dengan kunci publik hanya dapat didekripsi dengan kunci privat yang sesuai. Kunci privat itu sendiri sifatnya harus tetap rahasia dan hanya diketahui oleh penerima yang dituju. Hal ini menjamin hanya penerima yang memiliki kunci pribadi yang sesuai yang dapat mendekripsi pesan tersebut.

   Manfaat utama dari kriptografi kunci publik adalah kemampuannya untuk menyediakan pertukaran pesan aman tanpa perlu menukar kunci enkripsi secara langsung. Tetapi karena komputasinya yang relatif lebih lambat dibandingkan kriptografi simetris maka kriptografi kunci publik lebih sering digunakan untuk menetapkan sebuah kunci simetris yang seterusnya akan digunakan dalam enkripsi dan dekripsi antara dua belah pihak. Di dalam *signal protocol*, kriptografi kunci publik digunakan untuk menetapkan sebuah *shared secret* mula-mula agar kedua belah pihak dapat langsung berkomunikasi melalui jaringan terbuka yang relatif tidak aman.

2. Protokol Pertukaran Kunci Diffie-Hellman

   Pertukaran Kunci Diffie-Hellman (Diffie-Hellman Key Exchange) adalah protokol kriptografi yang digunakan untuk menghasilkan kunci enkripsi bersama antara dua entitas yang ingin berkomunikasi secara aman melalui saluran komunikasi yang tidak aman. Protokol ini dikembangkan oleh Whitfield Diffie dan Martin Hellman pada tahun 1976 dan telah menjadi dasar dalam banyak sistem keamanan modern. Dasar teori di balik Pertukaran Kunci Diffie-Hellman adalah matematika teori bilangan dan operasi aritmetika modular. Prinsip utama protokol ini adalah bahwa dua entitas yang ingin berkomunikasi dapat menghasilkan kunci enkripsi yang sama tanpa perlu bertukar kunci enkripsi secara langsung.

   Secara umum langkah-langkah dasar dalam pertukaran kunci Diffie-Hellman adalah pengaturan awal, pertukaran nilai pubilk, pertukaran pencampuran nilai publik dengan nilai privat, serta penghasilan kunci bersama. Pengaturan awal berfungsi untuk menyepakati suatu parameter publik dalam protokol yang nilainya diketahui oleh semua entitas dalam komunikasi bahkan dapat diketahui oleh pihak ketiga. Dalam pertukaran nilai publik, kedua pihak membangkitkan sebuah kunci publik yang dihasilkan dari komputasi nilai publik yang ditetapkan sebelumnya dengan kunci privat masing-masing. Langkah berikutnya adalah menukarkan hasil pencampuran ini antara kedua belah pihak yang kemudian akan dicampur lagi dengan kunci privat masing-masing pihak. Hasil akhir pencampuran ini menghasilkan kunci rahasia yang sama yang tidak dapat diketahui oleh pihak lain.

3. Enkripsi *End-to-End*

   Enkripsi *end-to-end* (E2EE) adalah metode enkripsi yang digunakan untuk melindungi kerahasiaan pesan selama transmisi dari pengirim ke penerima. Tujuan utama enkripsi *end-to-end* adalah memastikan bahwa hanya penerima yang dituju yang dapat membaca isi pesan, sementara pihak ketiga, termasuk penyedia layanan atau pihak jahat, tidak dapat mengakses atau memahami pesan tersebut. Selain enkripsi dan dekripsi, enkripsi *end-to-end* juga dapat digunakan untuk memastikan keutuhan dan otentikasi pesan menggunakan *digital signature* misalnya HMAC.

4. *Perfect Forward Secrecy*

   Perfect Forward Secrecy (PFS) adalah konsep dalam kriptografi yang menjamin bahwa jika suatu kunci enkripsi kompromi pada suatu waktu tertentu, pesan yang sudah dienkripsi di masa lalu tetap aman dan tidak dapat dipecahkan. PFS memastikan kerahasiaan

pesan yang telah dikirimkan di masa lalu terlindungi meskipun ada pelanggaran keamanan di masa depan.

Dasar teori PFS melibatkan penggunaan kunci enkripsi simetris yang berbeda untuk setiap sesi komunikasi. Hal ini dicapai dengan melakukan pertukaran kunci yang unik untuk setiap sesi pertukaran pesan. Dalam *signal protocol* hal ini dicapai dengan pertukaran kunci Diffie-Hellman. Kunci simetris ini sifatnya sementara dan akan dihancurkan setelah selesai digunakan. Ini memberikan perlindungan terhadap kompromi kunci karena serangan terhadap salah satu sesi hanya akan bisa dilakukan untuk mengkompromi pesan-pesan yang ada pada sesi tersebut. Pesan yang dienkripsi dalam sesi sebelumnya tetap aman karena kunci enkripsi yang berbeda digunakan dalam setiap sesi komunikasi.

Jika PFS memberikan keamanan untuk sesi sebelumnya maka *perfect backward secrecy* (PBS) memberikan keamanan untuk sesi setelahnya. *Signal protocol* tidak memberikan *perfect backward secrecy* namun memberikan *backward secrecy* yang sifatnya partial. Hal ini dikarenakan kunci simetris baru tidak dijamin selalu dibangkitkan setelah sebuah pesan dikompromi dan didapatkan kuncinya. Kunci hanya akan dibangkitkan ketika sesi baru diciptakan yang momennya bergantung pada implementasi seberapa sering kunci akar baru dibangkitkan dan dipertukarkan menggunakan *Diffie-Hellman*.

5. Verifikasi Identitas

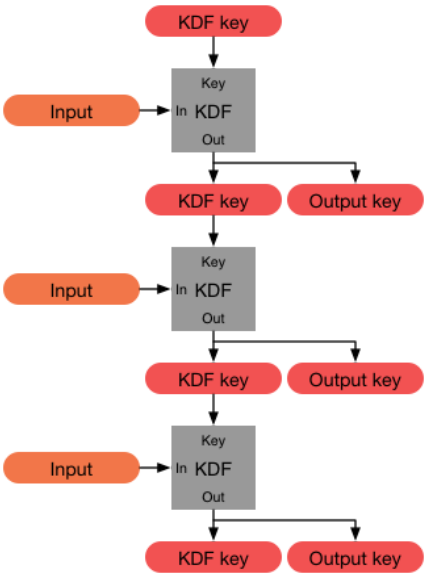
Dalam *signal protocol* hanya pengirim pesan dan pihak yang dituju yang dapat mengetahui isi dari pesan. Ini termasuk layanan yang menyediakan aplikasi pengiriman pesan. Hal ini dicapai dengan melakukan enkripsi dan dekripsi pada sisi *client* misalnya di aplikasi *mobile*. Karena kunci simetris yang digunakan dalam setiap pesan hanya dapat diketahui dan dibangkitkan oleh pengirim dan penerima yang sah maka otomatis hal ini menjamin identitas dari pengirim dan penerima pesan. Walaupun salah satu pesan dapat dikompromi oleh seorang penyerang namun penyerang hanya dapat melakukan dekripsi terhadap sebagian kecil pesan yang dipertukarkan.

B. Algoritma *Double Ratchet*

Algoritma *Double Ratchet* merupakan inti dari *Signal Protocol*. Algoritma ini digunakan oleh dua pihak untuk bertukar pesan terenkripsi yang berbasis pada suatu nilai kunci rahasia yang *shared*. Kedua belah pihak akan senantiasa membangkitkan kunci baru untuk setiap pesan yang ditransmisi menggunakan *double ratchet* sehingga setiap pesan tidak dapat didekripsi menggunakan kunci yang sama apabila terkompromi.

Terminologi *double ratchet* bermaksud untuk merujuk pada dua komponen pembangkit bilangan yang hanya bergerak satu arah seperti roda gerigi (*ratchet)*. Algoritma *double ratchet* dapat dibayangkan seperti dua roda gerigi, roda gerigi yang pertama adalah *root ratchet* dan di dalamnya ada roda gerigi lain yang sering disebut *sending* dan *receiving ratchet*. Kadang *sending* dan *receiving ratchet* digabung sehingga hanya ada dua roda gerigi namun seringkali kedua *ratchet* ini dipisah namun demikian mereka sifatnya independen dan *double ratchet* lebih merujuk ke dua buah tingkatan roda gerigi alih-alih hanya ada tepat dua roda gerigi.

Di dalam setiap *ratchet* terdapat sebuah *key derivation function* (KDF) yaitu sebuah algoritma kriptografi yang dapat membangkitkan suatu nilai kunci rahasia dari sebuah nilai rahasia (*master key/KDF key)* dan sebuah input acak/tetap. Algoritma ini biasanya diimplementasi menggunakan *block cipher* atau fungsi *hash* yang menerima *input* kunci seperti HMAC. Untuk setiap keluaran dari KDF akan menghasilkan dua potongan yaitu kunci keluaran yang diinginkan *(output key)* dan kunci KDF baru (KDF *key*) yang akan mempengaruhi keluaran iterasi berikutnya. Input dapat dibuat semi acak dengan dibangkitkan secara deterministik agar kedua pihak dapat mengetahuinya namun apabila input dibuat konstan pun algoritma masih akan tetap aman apabila KDF diimplementasi menggunakan *secure hash algorithm*. Karena iterasi hanya bergerak satu arah maka bisa dibayangkan seperti sebuah roda gerigi (*ratchet*).



Gambar 1. Rantai fungsi KDF, sumber [1]

Roda gerigi utama yaitu *root ratchet* berfungsi untuk membangkitkan roda gerigi lain di dalamnya yaitu *sending* dan *receiving ratchet*. Pembangkitan pertama dilakukan dari sebuah nilai *shared secret* yang harus diketahui oleh kedua

pihak. Nilai ini awalnya dapat dinegosiasikan dan dipertukarkan antara dua pihak yang berkomunikasi melalui jaringan tidak aman menggunakan enkripsi asimetris. *Shared secret* ini berperan sebagai KDF *key* untuk *root ratchet*. Nilai KDF *key* dari *root ratchet* akan bersifat dinamis untuk memberikan *backward secrecy* terhadap kunci-kunci yang dibangkitkan oleh *sending* maupun *receiving ratchet*. *Root ratchet* diperlukan karena *sending* maupun *receiving ratchet* membangkitkan kunci secara deterministik sehingga hanya memberikan *perfect forward secrecy*. Penyerang yang mendapatkan kunci di salah satu iterasi rantai KDF akan bisa secara mandiri menurunkan kunci berikutnya untuk mendekripsi pesan-pesan berikutnya. Dengan adanya *root ratchet* maka *sending* dan *receiving ratchet* dapat dibangkitkan ulang menggunakan KDF *key* awal yang berbeda sehingga *backward secrecy* juga dapat diberikan. Properti ini juga membuat *Double Ratchet* dikenal sebagai *Axolotl Ratchet* karena kemampuannya dalam melakukan *self-healing*. *Root ratchet* akan membangkitkan *sending* dan *receiving ratchet* yang baru setiap salah satu pengirim atau penerima mengisukan kunci publik *Diffie-Hellman* yang baru. Semakin sering pertukaran kunci publik baru ini dilakukan maka *backward secrecy* akan semakin terjamin.

Receiving ratchet penerima harus tersinkronisasi dengan *sending ratchet* pengirim. Ketika *ratchet* tersinkronisasi maka keluaran dari *receiving ratchet* akan sama dengan *sending ratchet.* Ketika keluaran *sending ratchet* pengirim digunakan untuk mengenkripsi pesan, keluaran *receiving ratchet* penerima digunakan untuk mendekripsi pesan tersebut. Sinkronisasi dilakukan dengan menggerakkan *ratchet* sebesar langkah yang sama untuk setiap pengiriman maupun penerimaan pesan. Sinkronisasi juga terjadi ketika

III. Rancangan dan Implementasi

Implementasi prototipe dilakukan penulis menggunakan bahasa pemrograman Go. Penggunaan algoritma *Double Ratchet* dalam pengiriman *file* adalah dengan membagi *file* menjadi beberapa *chunk* kecil berukuran konstan kemudian mengirimkan setiap *chunk* sebagai pesan. Mirip dengan cara pengiriman pesan biasa di *Signal Protocol* namun yang dikirimkan adalah *chunk* dari *file* dan *response* dari penerima. Alih-Alih mengirimkan berkas melalui jaringan, prototipe yang dibuat melakukan pengiriman secara lokal. Prototipe yang dibuat juga mengasumsikan pengiriman *file* dilakukan menggunakan model *request reply* yang sekuensial yaitu pengirim mengirimkan data kemudian pengirim menunggu konfirmasi *file chunk* telah diterima oleh penerima sebelum mengirimkan *file chunk* berikutnya.

```
for {
    // Read the next chunk
    chunk := make([]byte, chunkSize)
    n, err := reader.Read(chunk)

    if err != nil {
      if err.Error() == "EOF" {
        break
      }
      log.Fatal(err)
    }

    // Alice can now encrypt messages under the Double Ratchet session.
    m, err := alice.RatchetEncrypt(chunk, nil)
    if err != nil {
        log.Fatal(err)
    }

    // Which Bob can decrypt.
    plaintext, err := bob.RatchetDecrypt(m, nil)
    if err != nil {
        log.Fatal(err)
    }
```

Gambar 2. Kode *driver* simulasi pengiriman *file*, sumber dokumentasi pribadi

Langkah pertama yang dilakukan adalah menentukan sebuah nilai *shared secret* yang akan diketahui kedua pihak. Kemudian pihak pertama, misal disebut Bob akan menciptakan pasangan kunci Diffie-Hellman secara acak. Bob kemudian akan menginstantiasi sesi *Double Ratchet* miliknya dengan nilai *shared secret* dan pasangan kunci Diffie-Hellman miliknya. Setelah ini kunci publik dari Bob akan dikirimkan ke pihak kedua Alice. Alice kemudian akan menciptakan sesi *Double Ratchet* berdasarkan *shared secret,* kunci publik Bob, dan terakhir Alice juga akan menciptakan dan menggunakan sebuah pasangan kunci Diffie-Hellman miliknya yang dibangkitkan secara acak. Alasan Alice perlu mengetahui kunci publik dari Bob pada langkah awal ini adalah agar salah satu pihak dapat langsung mengirimkan pesan, dalam kasus ini, pihak yang dapat mengirimkan pesan adalah Alice.

```
// Diffie-Hellman key pair generated by one of the parties during key exchange or
// by any other means. The public key MUST be sent to another party for initialization
// before the communication begins.
keyPair, err := dra_crypto.GenDHKey(rand.Reader)
if err != nil {
    log.Fatal(err)
}

// Bob MUST be created with the shared secret and a DH key pair.
bob, err := dra_session.New([]byte("bob-session-id"), sk, keyPair)
if err != nil {
    log.Fatal(err)
}

// Alice MUST be created with the shared secret and Bob's public key.
alice, err := dra_session.NewWithRemoteKey([]byte("alice-session-id"), sk, keyPair.PublicKey())
if err != nil {
    log.Fatal(err)
}
```

Gambar 3. Kode inisialisasi sesi *Double Ratchet*, sumber dokumentasi pribadi

Setiap pesan akan memiliki nilai *header* yang di dalamnya memiliki beberapa informasi, salah satunya adalah nilai kunci publik pengirim. Ini memungkinkan Bob untuk mendapatkan nilai kunci publik dari Alice saat Alice pertama kali mengirimkan pesan. Mekanisme ini yang memungkinkan

Alice dan Bob untuk melakukan sinkronisasi sesi *Double Ratchet*.

```
type Message struct {
    Header     MessageHeader
    Ciphertext []byte
}

// MessageHeader that is prepended to every message.
type MessageHeader struct {
    // DHr is the sender's current ratchet public key.
    DH dra_crypto.Key `json:"dh"`

    // N is the number of the message in the sending chain.
    N uint32 `json:"n"`

    // PN is the length of the previous sending chain.
    PN uint32 `json:"pn"`
}
```

Gambar 4. Kode *Message struct*, sumber dokumentasi pribadi

Misalnya ketika Alice mengirimkan pesan maka *sending ratchet* miliknya akan maju satu langkah kemudian kunci keluaran dari *sending ratchet* akan digunakan untuk melakukan enkripsi terhadap pesan yang dikirimkan. Saat pesan sampai ke Bob karena *receiving ratchet* Bob identik dengan *sending ratchet* milik Alice sesaat sebelum pengiriman terjadi maka Bob juga hanya perlu menggerakan *receiving ratchet* miliknya sebesar satu langkah. Keluaran *receiving ratchet* dari Bob akan sama dengan keluaran *sending ratchet* milik Alice dan karena pesan dienkripsi menggunakan kriptografi simetris maka kunci keluaran *receiving ratchet* dari Bob juga bisa digunakan untuk mendekripsi pesan Alice. Mekanisme ini yang memberikan properti *perfect forward secrecy* pada *Double Ratchet.*

```
// Symmetric-key ratchet
type SKRatchet struct {
    ChainKey Key
    N uint32
}

// Performs a symmetric-key ratchet step.
// Updates the chain key and the number of message in the ratchet
// and returns the new derived message key.
func (r *SKRatchet) Step() Key {
    var MessageKey Key

    // Derive chain key and message key with HMAC
    r.ChainKey, MessageKey = GenSKRatchetOutput(r.ChainKey)

    r.N++
    return MessageKey
}
```

Gambar 5. Kode *struct* dari *sending/receiving ratchet,* sumber dokumentasi pribadi

Root ratchet hanya akan maju ketika nilai salah satu kunci publik berubah. Setiap sesi *Double Ratchet* akan memiliki sebuah *state* yang di dalamnya berisi banyak informasi, salah satunya adalah informasi mengenai kunci publik *remote* (kunci publik pihak kedua). Misalnya ketika Bob menerima pesan dari Alice, Bob harus mengecek apakah kunci publik yang diberikan pada *message header* berbeda dengan nilai kunci publik *remote* yang disimpan pada *state.* Apabila berbeda maka *root ratchet* akan maju satu langkah berdasarkan kunci publik *remote* yang dicampur dengan kunci privat miliknya sesuai dengan konsep pertukaran kunci Diffie-Hellman. Mekanisme ini otomatis akan seolah-olah melakukan "*reset"* terhadap *step ratchet* dan *receiving ratchet* sehingga memberikan *backward secrecy*.

```
// Diffie-Hellman ratchet
type DHRatchet struct {
    RootChainKey Key
}

// Performs a Diffie-Hellman ratchet step
func (r* DHRatchet) Step(dhOutput Key) (ratchet SKRatchet, headerKey Key) {
    r.RootChainKey, ratchet.ChainKey, headerKey = GenDHRatchetOutput(r.RootChainKey, dhOutput)

    return ratchet, headerKey
}
```

Gambar 6. Kode *struct* dari *root ratchet*, sumber dokumentasi pribadi

Algoritma enkripsi bebas tetapi penulis memilih menggunakan AES karena sudah ada kakas yang tersedia pada pustaka kode *crypto* milik Golang. Varian AES yang digunakan secara spesifik adalah AES-256-CTR. Varian apapun sebenarnya bisa digunakan selama algoritma kriptografi yang dilakukan merupakan kriptografi simetris. Alasan pemilihan AES-256-CTR adalah lebih aman dan masih relatif cepat apabila dibandingkan dengan mode CBC pada AES-256-CBC. Nilai yang dibutuhkan dalam enkripsi seperti kunci enkripsi hingga nilai *initialization vector* diturunkan dari *output key (message key)* yang dikeluarkan oleh *ratchet*. Selain enkripsi pada pesan juga diberikan tanda tangan digital untuk memastikan integritas, otentikasi, dan non-repudiasi pesan. Tanda tangan digital juga bebas namun di sini penulis memilih untuk menggunakan HMAC dengan nilai kunci diturunkan dari *output key*.

```
func (s *sessionState) RatchetEncrypt(plaintext, ad []byte) (Message, error) {
    var (
        h = MessageHeader{
            DH: s.DHs.PublicKey(),
            N: s.SendRatchet.N,
            PN: s.PN,
        }
        mk = s.SendRatchet.Step()
    )
    ct, err := dra_crypto.EncryptAES(mk, plaintext, append(ad, h.Encode()...))
    if err != nil {
        return Message{}, err
    }

    return Message{h, ct}, nil
}
```

Gambar 7. Langkah *ratchet encrypt*, sumber dokumentasi pribadi

```go
// Encrypt uses a slightly different approach than in the algorithm specification:
// it uses AES-256-CTR instead of AES-256-CBC for security, ciphertext length and implementation
// complexity considerations.
func EncryptAES(mk Key, plaintext, ad []byte) ([]byte, error) {
    encKey, authKey, iv := deriveEncKeys(mk)

    ciphertext := make([]byte, aes.BlockSize+len(plaintext))
    copy(ciphertext, iv[:])

    var (
        block, _ = aes.NewCipher(encKey[:]) // No error will occur here as encKey is guaranteed to be 32 bytes.
        stream   = cipher.NewCTR(block, iv[:])
    )
    stream.XORKeyStream(ciphertext[aes.BlockSize:], plaintext)

    return append(ciphertext, computeSignature(authKey[:], ciphertext, ad)...), nil
}
```

Gambar 8. Proses enkripsi pesan dan *signing*, sumber dokumentasi pribadi

Proses dekripsi juga relatif sederhana yaitu sebatas menggerakkan *root ratchet* sebesar satu langkah apabila *public key* yang diberikan di *message header* berbeda. Kemudian tanpa bergantung pada keputusan langkah sebelumnya, *receiving ratchet* akan maju satu langkah untuk menghasilkan *output key* yang dipakai untuk melakukan dekripsi pesan. Setelah dekripsi dilakukan, tanda tangan digital akan diverifikasi terlebih dahulu untuk menghindari penulisan *file* yang tidak diinginkan ke *persistent memory*.

```go
func (s *sessionState) RatchetDecrypt(m Message, ad []byte) ([]byte, error){
    // TODO: Handle skipped messages
    var (
        sc = s.State
    )

    if !bytes.Equal(m.Header.DH, sc.DHr) {
        if err := sc.DHRatchet(m.Header); err != nil {
            return nil, fmt.Errorf("can't perform ratchet step: %s", err)
        }
    }

    mk := sc.RecvRatchet.Step()
    plaintext, err := dra_crypto.DecryptAES(mk, m.Ciphertext, append(ad, m.Header.Encode()...))
    if err != nil {
        return nil, fmt.Errorf("can't decrypt: %s", err)
    }

    s.State = sc

    return plaintext, nil
}
```

Gambar 9. Langkah *ratchet decrypt*, sumber dokumentasi pribadi

```go
func DecryptAES(mk Key, authCiphertext, ad []byte) ([]byte, error){
    var (
        l          = len(authCiphertext)
        ciphertext = authCiphertext[:l-sha256.Size]
        signature  = authCiphertext[l-sha256.Size:]
    )

    // Check the signature.
    encKey, authKey, _ := deriveEncKeys(mk)

    if s := computeSignature(authKey[:], ciphertext, ad); !bytes.Equal(s, signature) {
        return nil, fmt.Errorf("invalid signature")
    }

    // Decrypt.
    var (
        block, _  = aes.NewCipher(encKey[:]) // No error will occur here as encKey is guaranteed to be 32 bytes.
        stream    = cipher.NewCTR(block, ciphertext[:aes.BlockSize])
        plaintext = make([]byte, len(ciphertext[aes.BlockSize:]))
    )
    stream.XORKeyStream(plaintext, ciphertext[aes.BlockSize:])

    return plaintext ,nil
}
```

Gambar 10. Proses dekripsi dan verifikasi *digital signature,* sumber dokumentasi pribadi

Untuk menghindari kompleksitas yang berlebihan maka setiap pihak tidak bisa mengirimkan lebih dari satu pesan sekaligus. Pada kasus dunia nyata hal ini tentu sangat tidak efisien karena pesan dikirimkan melalui jaringan yang merupakan salah satu bentuk I/O sehingga akan jauh lebih efisien apabila dalam satu *batch* seorang pengirim mengirimkan lebih dari satu pesan atau dalam kasus makalah ini, lebih dari satu *file chunk*. Belum lagi mempertimbangkan kasus pesan yang hilang di jalan akibat jaringan yang selalu harus diasumsikan *unreliable*. Akan tetapi semua kasus tersebut kita abaikan karena sekali lagi, prototipe yang dibuat hanya akan mengirimkan pesan secara *in-memory* (lokal).

IV. PENGUJIAN

Pengujian dilakukan hanya untuk membuktikan kebenaran program. Simulasi dilakukan tidak terhadap *file* berukuran besar tetapi menggunakan *file* berukuran kecil dengan ukuran *chunk* yang sangat kecil untuk mensimulasikan *file* berukuran besar. *File* yang digunakan adalah *file* teks berisikan teks *dummy lorem ipsum*. Pada kode program akan di-*hard code* dua buah pihak yaitu Alice dan Bob. Alice akan membuka *file* pada *folder sender* dan mengenkripsi dan mengirimkan *chunk* demi *chunk* berukuran tepat 4 *byte* ke Bob. Bob akan menuliskan hasil dekripsi dari pesan yang dikirimkan Alice ke *folder receiver*. Integritas dari *file* akan diperiksa dengan menghitung hasil *hash* dari *file* yang dikirim dan *file* yang diterima.

```
aphostrophy@DESKTOP-0E6RKI3:~/Double-Ratchet$ sha1sum sender/data.txt
a5f363d8668ef370349c3d2fee6fb56525bdca0a  sender/data.txt
aphostrophy@DESKTOP-0E6RKI3:~/Double-Ratchet$ go run driver.go
aphostrophy@DESKTOP-0E6RKI3:~/Double-Ratchet$ sha1sum receiver/data.txt
34d316c0d04f35641fc2f439d9a2bbabf4f9ff81  receiver/data.txt
```

Gambar 11. Pengujian integritas hasil pengiriman *file,* sumber dokumentasi pribadi

Selain pengujian di atas, penulis juga mencoba untuk mengecek keamanan dari algoritma *Double Ratchet* yang diimplementasi seperti mengubah isi dari teks cipher sehingga verifikasi *digital signature* gagal, membuang *publik key* Diffie-Hellman dari *message header* sehingga *root ratchet* gagal untuk saling mensinkronisasi antara kedua pihak, hingga menggunakan algoritma enkripsi dan dekripsi kriptografi simetris yang berbeda agar pengiriman gagal.

```
aphostrophy@DESKTOP-0E6RKI3:~/Double-Ratchet$ go run driver.go
2023/05/21 22:29:02 can't decrypt: invalid signature
exit status 1
```

Gambar 12. Pengujian kegagalan pengiriman *file*

Perlu dicatat bahwa penambahan *layer* algoritma *Double Ratchet* pasti memberikan *overhead* tambahan pada pengiriman *file*. Pada makalah ini tidak dilakukan pengujian mengenai seberapa besar nilai *overhead* ini namun secara teori seharusnya *overhead* ini jauh melebihi waktu untuk memproses pengiriman *file* secara mentah. Belum lagi ditambah apabila algoritma yang dibuat sudah mempertimbangkan kasus-kasus tertentu, tidak seperti prototipe yang dibuat pada makalah ini yang menggunakan asumsi-asumsi tertentu agar implementasi menjadi lebih mudah.

V. KESIMPULAN

Algoritma *Double Ratchet* dapat digunakan untuk memberikan keamanan tambahan terhadap transmisi *file* berukuran besar. Keamanan yang diberikan adalah penyerang perlu menemukan lebih dari satu kunci untuk melakukan dekripsi terhadap seluruh *file*. Akan tetapi belum divalidasi apakah lapisan keamanan tambahan ini memang esensial untuk diterapkan pada pengiriman *file*. Hal ini dikarenakan lapisan tambahan ini memberikan *overhead* tambahan dalam pengiriman pesan baik dari segi waktu dan sumber daya. Prototipe yang diberikan di dalam makalah ini juga belum efisien karena masih memaksa kedua pihak untuk mengirimkan *request* dan *response* secara sekuensial. Di masa depan pengujian maupun validasi apapun sebaiknya dilakukan terhadap implementasi yang lebih efisien agar hasilnya lebih representatif.

VI. REFERENSI

[1] https://signal.org/docs/specifications/doubleratchet/ , diakses tanggal 18 Mei 2023

[2] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila, “A Formal Security Analysis of the Signal Messaging Protocol.” Cryptology ePrint Archive, Report 2016/1013, 2016. http://eprint.iacr.org/2016/1013

[3] Munir, Rinaldi. (2021). Slide Perkuliahan IF4020 Kriptografi. Di akses melalui https://informatika.stei.itb.ac.id/.