Crittografia in Java

di Oreste Delitala

Progetto di Computer Security 2013-2014

Introduzione

La crittografia è un particolare processo grazie al quale, per mezzo di sofisticati algoritmi, è possibile trasformare una sequenza di byte con senso logico (messaggio) in un altra del tutto incomprensibile.
Scopo della crittografia è consentire la trasmissione di un messaggio in forma non leggibile ad altri che non sia il destinatario inteso, che deve essere il solo a poterne capire il significato, la trasformazione avviene grazie ad una chiave : solo chi possiede la chiave per aprire e chiudere il messaggio potrà criptare e decriptare il messaggio.

La Crittografia è usata per garantire la confidenzialità di un messaggio tra mittente e ricevente, oppure per autenticare al ricevente un messaggio con il legittimo mittente.
Le tecniche usate per garantire confidenzialità sono la cifratura Simmetrica e la cifratura Asimmetrica, dove la prima utilizza una sola chiave condivisa tra i due interlocutori, invece la seconda usa due chiavi distinte per entrambi gli interlocutori una chiave per cifrare ed una chiave per decifrare, la chiave per cifrare resa pubblica in modo che chiunque possa mandare messaggi nascosti e la chiave per decifrare privata in modo da poter interpretare soltanto il ricevente il messaggio inviato e garantendo confidenzialità.
Le tecniche usate invece per garantire l'autenticazione del messaggio invece sono il Mac e la Firma Digitale. Il Mac viene utilizzato per autenticare messaggi usando una chiave Simmetrica che deve essere condivisa tra entrambi gli interlocutori, invece la Firma Digitale usa due chiavi come per la crittografia asimmetrica, con la differenza che firmo con la chiave privata e verifico che la firma sia corretta con la chiave pubblica.

Il progetto consiste nel creare una demo che implementi queste tecniche di crittografia nel linguaggio di programmazione Java, utilizzando le librerie disponibili sul web.
Le librerie scaricate sono:

- javax-crypto.jar
- javax-security.jar
- sun.misc.BASE64Decoder.jar

JCA e JCE

Java Cryptography Architecture (JCA)

La Java Cryptography Architecture (JCA) è un framework Java, incluso nel JDK, che permette di eseguire operazioni crittografiche, basato su principi di:

- Indipendenza dall'implementazione
- Interoperabilità delle implementazioni
- Indipendenza e estendibilità degli algoritmi

Indipendenza dall'implementazione perché le applicazioni non hanno bisogno di implementare funzioni crittografiche, possono richiedere i servizi crittografici alla piattaforma Java, dove i servizi per le funzioni di sicurezza saranno implementati da diversi Provider.
I Provider stessi dovranno conformarsi a un'interfaccia comune, che permette alla piattaforma di fornire i servizi richiesti.

Interoperabilità perché i servizi forniti dalle diverse implementazioni devono essere interoperabili, è possibile ad esempio creare una chiave di firma con un'implementazione, usare un'implementazione diversa per firmare i documenti e una terza implementazione per verificare le firme. L'interoperabilità delle implementazioni è complementare all'indipendenza dalle stesse, si possono usare i servizi crittografici senza preoccuparsi di chi li implementa.

Indipendenza dagli algoritmi si ottiene attraverso la definizione di Servizi che forniscono le funzionalità di base necessarie alla crittografia come:

- Signature per la firma
- MessageDigest per le impronte
- KeyPairGenerator per le coppie di chiavi
- KeyFactory per la chiavi simmetriche
- Cipher per la cifratura
- MAC
- …

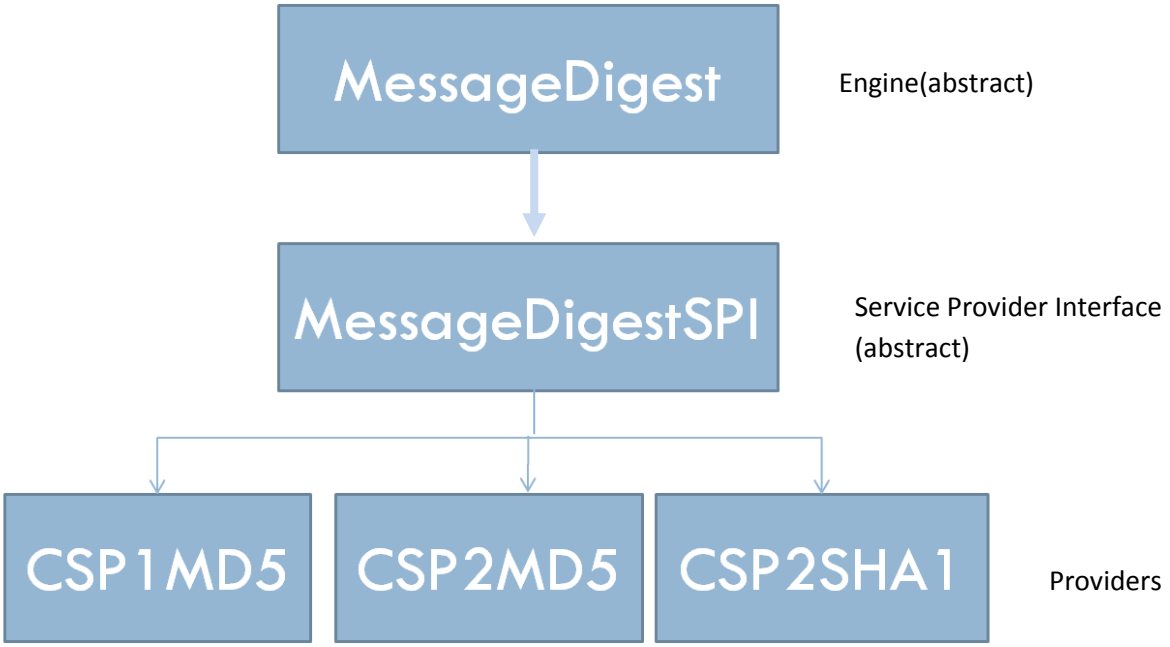
L'indipendenza dall'implementazione è ottenuta aggiungendo alla piattaforma dei plugin chiamati CSP (Cryptographic Service Provider) o semplicemente Provider. Ogni Provider fornisce uno o più Servici per determinati algoritmi, l'interoperabilità degli algoritmi garantisce che i Servizi forniti da Provider diversi ma relativi allo stesso algoritmo possano cooperare.

Estendibilià degli algoritmi è il principio di estendibilità su cui è basata la piattaforma JCA, implica che nuovi algoritmi possono essere aggiunti facilmente, l'unico requisito è che siano uniformabili a uno dei Servizi previsti: Cipher, MessageDigest etc.

Architettura JCA

L'architettura JCA è composta dalle seguenti componenti:

- **Factory pattern** che definisce un interfaccia per la creazione di un oggetto ma lascia che le sottoclassi decidano quale classe istanziare, e l'istanza si ottiene con **getInstance( ).**
- **Strategy pattern** che astrae una famiglia di algoritmi, li incapsula e li rende interscambiabili, permettendo all'utente di scambiare gli algoritmi provenienti da provider diversi



Provider

Un Provider è un insieme di implementazioni di vari algoritmi.

Il provider SUN (sun.java.security.sun) incorpora i seguenti algoritmi:

- MD5, SHA-1
- DSA: firma, verifica e generazione delle chiavi, parametri
- Creazione di certificati X.509
- Generazione di numeri random proprietaria
- Keystore proprietario

Altri provider RSAJCA (com.sun.rsajca.Provider):

- Gestione chiavi RSA
- Firma digitale RSA con SHA-1 o MD5

Java Cryptography Extension (JCE)

La Java Cryptography Extension (JCE) implementa in modo completo le funzionalità di cifratura e decifrazione dichiarate dalla JCA, offrendo il supporto all'utilizzo di cifrari simmetrici a blocco e a flusso, cifrari asimmetrici e cifrari con password tutti applicabili a Dati, Serializable Objects e I/O stream.

Le sue classi principali di JCE sono:

- Cipher
  - CipherInputStream / CipherOutputStream
- KeyGenerator
- SecretKeyFactory
- SealedObject
- MAC

Principali Provider per JCE sono:

- Provider "SunJCE"
  - Algoritmi di cifratura (AES, DES, TripleDes, PBE, …)
  - Modalità di cifratura (ECB, CBC, OFB, CFB, CTR, PCBC)
  - Padding (NoPadding, PKCS5PADDING)
  - Facilità per la conversione delle chiavi da oggetti Java in array di byte e viceversa
  - KeyGenerator
  - Algoritmi MAC

- Provider "BouncyCastle" (non integrato nel JDK "va installato")
  - Implementa ed estende le tecniche crittografiche definite dalla JCA e dalla JCE
    - Implementazione alternativa degli algoritmi di cifratura (AES, DES, TripleDES, PBE, …)
    - Algoritmi di cifratura asimmetrica (RSA, ElGamal)
    - Algoritmi di firma digitale
    - Padding (None, PKCS1PADDING)
    - Facilità per diversi protocolli di KeyAgreement
    - Facilità per utilizzare cifrari simmetrici a flusso e a blocchi

Le **API** utilizzate dalle 2 architetture JCA e JCE sono:

- Package java.security:
  - API per configurare JCA
  - Configurare e interrogare i Provider
  - Definizione generica di Service
  - Classi statiche con metodi statici di configurazione
- Package javax.crypto
  - API crittografiche (JCE)
  - Definizione dei diversi Service (non implementazione degli algoritmi)
  - Pattern Factory

Cifratura Simmetrica

Introduzione

Un algoritmo di cifratura si dice a **chiave simmetrica** quando la stessa chiave K viene usata sia per la cifratura, sia per la successiva decifratura.

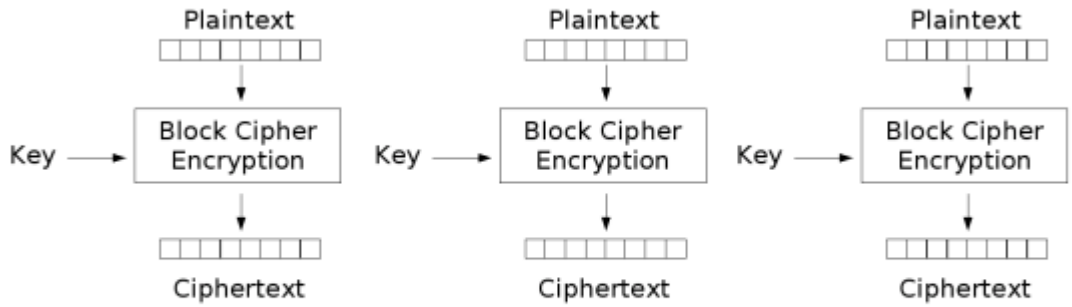
Le implementazioni svolte nella Demo fanno riferimento ai seguenti algoritmi di Cifratura:

- DES
- DESede (TripleDes)
- AES

Il primo usa una chiave a 64 bit, il secondo una chiave a 192 bit ed il terzo una chiave a 128 bit.

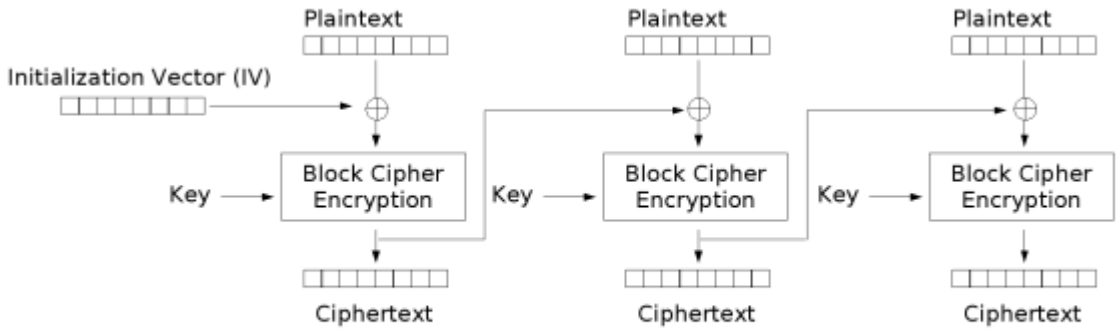
Vengono usate due modalità di cifratura:

- **ECB**: dove ogni blocco da cifrare viene cifrato direttamente con la chiave simmetrica K.



Electronic Codebook (ECB) mode encryption

- **CBC**: dove il primo blocco viene XORato con un IV generato Random e crittografato con la chiave K, e la risultate codifica generata viene XORata con il blocco successivo del testo in chiaro e codificato con la chiave K, e così via fino all'ultimo blocco del testo in chiaro.



Cipher Block Chaining (CBC) mode encryption

E vengono usati i seguenti Padding:

- **NoPadding**: non viene inserita nessuna sequenza di byte nell'ultimo blocco del testo in chiaro, quindi si suppone che il testo in chiaro sia un multiplo del blocco di cifratura.

- **PKCS5Padding**: viene aggiunta all'ultimo blocco del testo in chiaro una sequenza di byte in modo da renderlo della stessa grandezza del blocco per la cifratura.

Implementazione

Le principali classi ed interfacce utilizzate sono state:

- javax.crypto.Cipher
  - getInstance, init , update, doFinal
- java.security.Key (interfaccia)
  - Un oggetto Key viene creato con un factory:
    - javax.crypto.KeyGenerator (se generiamo la chiave senza specifiche iniziali)
    - Javax.security.KeyFactory (se vogliamo che la chiave sia generata seguendo delle specifiche iniziali)
- javax.crypto.KeyGenerator
  - getInstance, init, generateKey

Vediamo nello specifico la classe principale utilizzata per eseguire Cifratura Simmetrica (la classe **Cipher**)
La classe Cipher è un engine class, una classe astratta che definisce le funzionalità di un dato tipo di algoritmo crittografico, senza però fornire alcuna implementazione, questa sarà definita con il metodo getInstance dove noi andremo a inserire il Provider e le informazioni sull'algoritmo che utilizzeremo.

- getInstance ("Algoritmo/Modalità/Padding", "Provider");
  - Algoritmo:
    - DES
    - DESede
    - AES
  - Modalità
    - ECB
    - CBC
  - Padding
    - NoPadding
    - PKCS5Padding
  - Provider
    - SUN
    - Può essere anche omesso usa quello standard SunJCE

Altri metodi della classe Cipher utilizzati sono:

- init(mode, key)
  - mode:
    - Cipher.ENCRYPT_MODE (modalità di cifratura)
    - Cipher.DECRYPT_MODE (modalità di decifratura)
- update(bytes) (vengono inseriti i bytes da elaborare)
- doFinal(bytes) (esegue la Codifica o Decodifica sui bytes passati in input)

I passi della Crittografia Simmetrica

I passi per effettuare la Crittografia Simmetrica in ambiente Java sono i seguenti
(ora un esempio di Crittografia Des in modalità ECB):

- Creazione di una chiave
    1. KeyGenerator keyGenerator = KeyGenerator.getIstance(“DES”);
    2. keyGenerator.init(64);
    3. SecretKey key = keyGenerator.generateKey();
- Creazione ed inizializzazione di un cifrario
    1. Chiper cipher = Cipher.getInstance(“DES/ECB/PKCS5Padding”);
- Cifratura
    1. cipher.init(Cipher.ENCRYPT_MODE, key);
    2. byte[] cipherText = cipher.doFinal(stringToEncrypt.getBytes());
- Decifratura
    1. cipher.init(Cipher.DECRYPT_MODE, key);
    2. byte[] plainText = cipher.doFinal(cipherText);

La classe KeyGenerator si occupa di generare una chiave simmetrica DES di lunghezza 64 bit. Con *getInstance(“DES”)* si definisce l’algoritmo per cui sarà usata la chiave, con *init(64)* si definisce la lunghezza della chiave e con *generateKey()* viene creata la chiave.

Adesso guardiamo la differenza con la modalità CBC (un esempio di Cifratura Des in modalità CBC):

- Creazione di una chiave
    1. KeyGenerator keyGenerator = KeyGenerator.getInstance(“DES”);
    2. SecretKey key = keyGenerator.generateKey();
- Creazione ed inizializzazione di un cifrario
    3. byte[] randomBytes = new byte[8]; //iv size = block size
    4. SecureRandom random = new SecureRandom();
    5. random.nextBytes(randomBytes);
    6. IvParameterSpec ivparams = new IvParameterSpec(randomBytes);
    7. Cipher cipher = Cipher.getInstance(“DES/CBC/PKCS5Padding”);
    8. Cipher.init(Cipher.ENCRYPT_MODE, key, ivparams); // cifratura
    9. Cipher.init(Cipher.DECRYPT_MODE, key, ivparams); // decifratura

La differenza sta nell’utilizzo di una variabile IV generata Random.

C’è anche un modo alternativo per la generazione della chiave, invece di lasciarla generare al KeyGenerator( ) la si fa generare a partire da una stringa.

```
1. String password = “password”;
2. byte[] desKeyData = password.getBytes();
3. DESKeySpec desKeySpec = new DESKeySpec(desKeyData);
4. SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DES");
5. SecretKey key = keyFactory.generateSecret(desKeySpec);
```

Sicurezza Cifratura Simmetrica

Per quanto riguarda la sicurezza per la Crittografia Simmetrica si hanno vari scenari. Per quanto riguarda la Modalità di cifratura la modalità ECB è vulnerabile ad attacchi di crittoanalisi (valutare la frequenza dei blocchi), invece nella modalità CBC questo tipo di attacco non dà nessun risultato. Per quanto riguarda invece l’algoritmo utilizzato il Des è vulnerabile in quanto usa una chiave a 64 bit e si potrebbe effettuare l’attacco del compleanno con un costo computazione di 2^{32} passi. Per risolvere questo problema è stato implementato il TripleDes (DESede) con una chiave a 192 bit, sufficiente a rendere inpraticabile l’attacco del compleanno, però la grandezza del blocco di 64 bit è sempre un limite per questo algoritmo. L’algoritmo più sicuro e più efficiente è AES con la lunghezza del blocco e della chiave a 128 bit.

Cifratura Asimmetrica

Introduzione

Un algoritmo di cifratura si dice a **chiave Asimmetrica** quando si usane sue chiavi, una viene usata sia per la cifratura ed è pubblica,una si usa per la successiva decifratura ed è privata.

Le implementazioni svolte nella Demo fa riferimento al seguente algoritmo di Cifratura:

- RSA

RSA è un algoritmo che permette di usare chiave di lunghezza 1024 o 2048 bit.
Un limite della Crittografia Asimmetrica è la complessità dell'algoritmo elevata, in quanto permette di cifrare solo piccole quantità di dati, per grandi quantità di dati viene usata un altro tipo di Crittografia Asimmetrica detta ***a chiave di sessione*** che consiste nei seguenti passi:

- **(e, d)** un public-private key pair
- **P** un "grande" payload
- **K** *newSimmetricKey()*
- **M = [E_k(P), E_e(K)]**

Un ibrido tra Crittografia Simmetrica e Crittografia Asimmetrica, viene generata una chiave simmetrica e viene cifrata con la chiave pubblica della controparte, e con la chiave simmetrica viene crittografato il Payload.

Un alternativa al normale pudding per RSA è l'algoritmo OAEP, dove prima di eseguire la cifratura RSA effettua due funzioni hash G e H (SHA-1 o SHA-256).

$$
\begin{aligned}
&\textbf{Enc}(N, m) \quad // \ m \in \{0,1\}^n \\
&\quad r \leftarrow_R \{0,1\}^{k_0}; \\
&\quad s \leftarrow G(r) \oplus (m || 0^{k_1}) \quad // \ s \in \{0,1\}^{n+k_1} \\
&\quad t \leftarrow H(s) \oplus r; \quad // \ t \in \{0,1\}^{k_0}; \\
&\quad w \leftarrow s||t; \quad // \ w \in \{0,1\}^k \\
&\quad y \leftarrow \mathtt{RSA}(w); \quad // \ y \in \{0,1\}^k; \\
&\quad \text{Return } y
\end{aligned}
$$

Implementazione

Principali classi ed interfacce utilizzate nella demo sono state:

- java.security.KeyPair
    1. getPublic(), getPrivate()
- java.security.PublicKey(interfaccia)
- javax.crypto.PrivateKey(interfaccia)
- Java.security.KeyPairGenerator
    1. genKeyPair()

e la classe Cipher come per la Crittografia Simmetrica:

- getInstance(“algorithm/mode/padding”, provider)
    1. Algorithm
        - RSA
    2. Mode
        - ECB
    3. Padding
        - NoPadding
        - PKCS1Padding
        - OAEPWithSHA-1AndMGF1Padding
        - OAEPWithSHA-256AndMGF1Padding
- init(mode, key)
    1. Cipher.ENCRYPT_MODE
        - publicKey
    2. Cipher.DECRYPT_MODE
        - privateKey
- update(bytes)
- doFinal(bytes)

I passi della Crittografia Asimmetrica

I passi per effettuare la Crittografia Asimmetrica in ambiente Java sono i seguenti
(ora un esempio di Crittografia RSA con padding PKCS1Padding):

- Creazione di una chiave
  1. KeyPairGenerator generator = KeyPairGenerator.getIstance(“RSA”);
  2. keyGenerator.init(1024);
  3. KeyPair pair = generator.generateKeyPair();
  4. Key pubKey = pair.getPublic();
  5. Key privKey = pair.getPrivate();
- Creazione ed inizializzazione di un cifrario
  6. Chiper cipher = Cipher.getInstance(“RSA/ECB/PKCS1Padding”);
- Cifratura
  7. cipher.init(Cipher.ENCRYPT_MODE, pubKey);
  8. byte[] cipherText = cipher.doFinal(stringToEncrypt.getBytes());
- Decifratura
  9. cipher.init(Cipher.DECRYPT_MODE, privKey);
  10. byte[] plainText = cipher.doFinal(cipherText);

La coppia di chiavi (pubblica, privata) vengono codificate con le seguenti codifiche, chiave pubblica con codifica X.509 e chiave privata con codifica PKCS#8, quindi se vogliamo mantenere le chiavi su dei file bisogna usare la giusta codifica e decodifica dei byte della chiave:

Decodifica

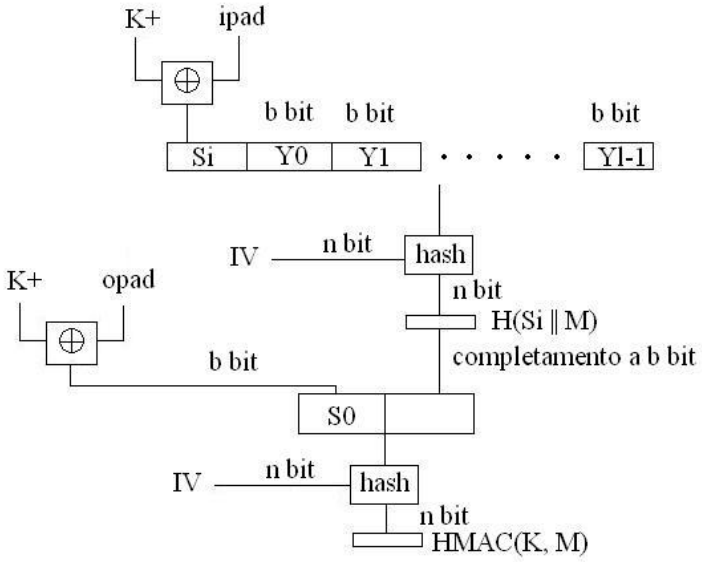
- Public key
  1. KeyFactory keyFactory = KeyFactory.getInstance("RSA");
  2. X509EncodedKeySpec publicKeySpec = new X509EncodedKeySpec(keyBytes);
  3. PublicKey publicKey = keyFactory.generatePublic(publicKeySpec);
- Private key
  4. KeyFactory keyFactory = KeyFactory.getInstance("RSA");
  5. PKCS8EncodedKeySpec privateKeySpec = new PKCS8EncodedKeySpec(keyBytes);
  6. PrivateKey privateKey = keyFactory.generatePrivate(privateKeySpec)

Sicurezza Cifratura Asimmetrica

La sicurezza riguardo la cifratura Asimmetrica sta nel problema della fattorizzazione, che è un problema computazionalmente intrattabile.

HMAC

L'Hmac è una tipologia di codice per l'autenticazione di messaggi basata su una funzione hash, utilizzata in diverse applicazioni legate alla sicurezza informatica. Tramite l'Hmac è infatti possibile garantire sia l'integrità che l'autenticità di un messaggio. Il messaggio originale viene codificato tramite una chiave simmetria condivisa tra i due agenti generando un codice che sarà inviato insieme al testo in chiaro.



Implementazione

Principali classi ed interfacce utilizzate nella Demo per l'Hmac sono state:

- java.crypto.Mac
    1. getInstance("algoritmo", "provider")
        - algoritmo
            - HmacSHA1
            - HmacSha256
    2. init(key)
    3. update(bytes)
    4. doFinal()

I passi della codifica

Adesso vedremo un esempio di codifica Hmac usando come funzione hash Sha-1.

- Generazione della chiave
    1. KeyGenerator keygen = KeyGenerator(“HmacSha1”)
    2. Key maKey = keygen.generateKey();
- Creazione di un oggetto MAC
    1. Mac mac = Mac.getInstance(“HmacShai1”);
- Inizializzazione del MAC
    1. mac.init(macKey);
- Restituzione del MAC
    1. mac.update(text.getByte());
    2. byte[] result = mac.doFinal();
- Verifica del MAC
    1. Si prende in input (messaggio, mac)
    2. Si calcola il mac del messaggio con la chiave Simmetrica
    3. Si confronta il mac calcolato con il mac ricevuto, se sono uguali torna esito positivo se sono diversi torna esito negativo

Sicurezza Hmac

L’Hmac è soggetto ad attacchi di tipo replay in quanto un messaggio con la stessa chiave torna sempre lo stesso codice mac. Una soluzione a questo tipo di attacco è cifrare con cifratura simmetrica in modalità CBC il codice mac e spedirlo. Oppure usare crittografia asimmetria con il vantaggio di non dover avere il problema della confidenzialità della chiave. Oppure mantenere condiviso, e segreto tra i due agenti, un valore SAB da concatenare al messaggio su cui calcolare l’Hmac:

HMAC(SAB||M)

Un altro problema è che non si ha la sicurezza di chi ha generato l’Hmac tra i due agenti quindi non garantisce autenticazione su chi ha generato il codice.

Firma Digitale

La Firma Digitale come l’Hmac è una tipologia di codice per l’autenticazione di messaggi utilizzata in diverse applicazioni legate alla sicurezza informatica. Tramite la Firma Digitale sempre come l’Hmac è infatti possibile garantire sia l’integrità che l’autenticità di un messaggio. La differenza sta sia nell’algoritmo di implementazione ma la differenza sostanziale sta nell’utilizzo di due chiavi asimmetriche (pubblica, privata), la chiave privata usata per firmare e la chiave pubblica per verificare la firma ricevuta.

Implementazione

Principali classi ed interfacce utilizzate nella Demo per l'Hmac sono state:

- java.crypto.Signature
    1. getInstance("algoritmo", "provider")
        - algoritmo
            - SHA1whithDSA
    2. initSign(privKey) (per inizializzare la firma)
    3. initVerify(pubKey) (per inizializzare la verifica)
    4. update(bytes)
    5. sign() (genera la firma)
    6. verify(sign) (verifica che la firma sia corretta)

I passi della Firma

Adesso vedremo un esempio di Firma Digitale implementato nella Demo.

- Generazione di una coppia di chiavi
    1. KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA", "SUN");
    2. SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
    3. keyGen.inizialize(1024, random);
    4. KeyPair keyPair = keyGen.generateKey();
- Generazione di un Signature engine
    1. Signature sig = Signature.getInstance("SHA1withDSA", "SUN");
- Generazione della Firma Digitale
    1. sig.initSign(privKey);
    2. sig.update(data);
    3. Byte[] signByte = sig.sign();
- Verifica della Firma Digitale
    1. sig.initVerify(pubKey);
    2. sig.update(data);
    3. try{
    4. verified = sig.verify(signByte);
    5. }catch(SignatureException e){
    6. verified = false;
    7. }

Sicurezza Firma Digitale

La **Firma Digitale** rispetto all'Hmac riesce a risolvere il problema di autenticare chi generare il codice, in quanto si usa la chiave privata per crittografare. Quindi riesce a garantire a livello di sicurezza la non ripudiabilità.