# Sicurezza Informatica

Relazione Progetto 3



Gruppo 2

Avitabile Gennaro

Di Benedetto Daniele

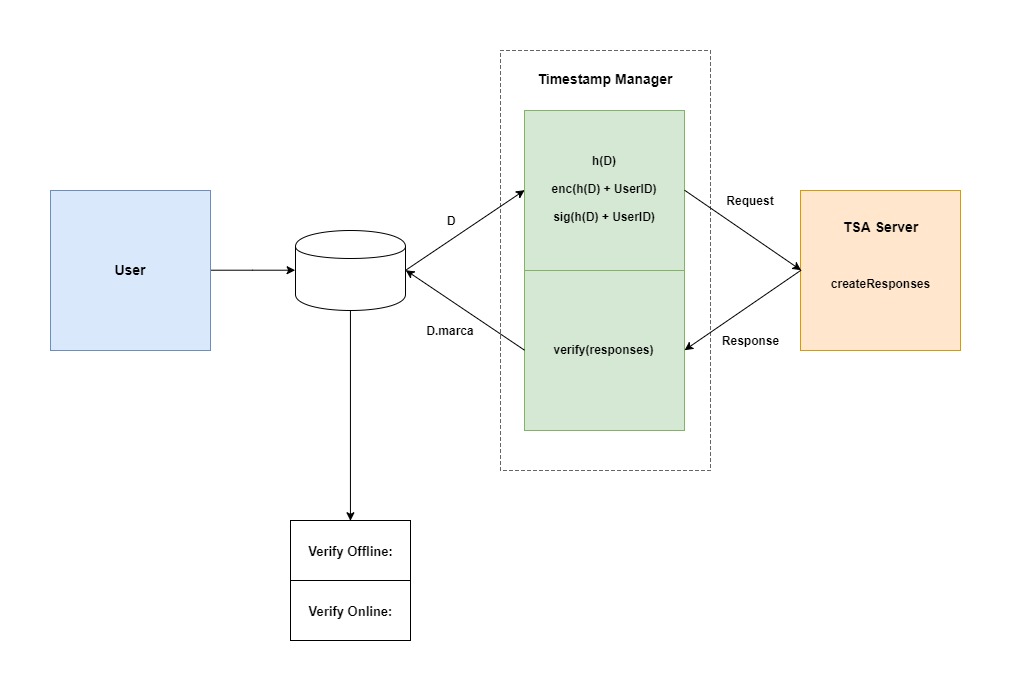
Pironti Christopher

Sicurezza Informatica 2017/2018

Corso di Laurea Magistrale in Ingegneria Informatica

Il task principale del progetto 3 è quello di implementare un servizio di timestamping. L’applicazione prevede un TimeStampingAuthority Server che riceve delle richieste da diversi utenti e genera le corrispondenti marche temporali. La seguente immagine illustra uno schema generale dell’applicazione:

## Schema Generale



## Timestamp Manager

La classe TimestampManager svolge il ruolo di interfaccia tra l’user e il TSA Server. Essa ha come attributi principali un ArrayList di richieste da inviare alla TSA e un ArrayList di risposte da ricevere dalla TSA e da processare (i.e. salvare su file, verificare offline e online).

1. Il metodo *generateRequest* si occupa principalmente della creazione della richiesta e della chiamata al metodo che permette di inviare automaticamente le richieste al TSAServer una volta che il numero di richieste ha raggiunto il limite massimo del timeframe (8). Il metodo prende in input l’ID del mittente, il suo file KeyRing, la sua password e il documento da inviare.   
   Nel metodo inizialmente vengono recuperate:
   1. La chiave privata DSA del mittente dal suo KeyRing.
   2. La chiave pubblica RSA della TSA dal file delle chiavi pubbliche.

Viene calcolato il digest del documento il quale viene inserito in un JSONObject insieme all’ID dell’user.

Viene istanziato l’oggetto TSAMessage passandogli in input le chiavi recuperate e il JSONObject costruito. A questo punto il TSAMessage viene inserito nell’ArrayList di richieste e nel caso in cui il numero di richieste è arrivato a 8 viene chiamato il metodo *processRequests* responsabile di inviare l’insieme di richieste al TSAServer, di ricevere le corrispondenti risposte e chiamare il metodo che permette di iniziare le successive operazioni su quest’ultime.

1. Il metodo *processResponses* si occupa della verifica delle firme digitali delle risposte ricevute dal TSA Server e del salvataggio delle marche su file. Inizialmente viene recuperata la chiave pubblica DSA della TSA dal file delle chiavi pubbliche dopodichè viene effettuato un for che cicla sull’ArrayList delle risposte. Per ogni risposta viene verificata la firma digitale tramite il metodo verifyText (che prende in input gli attributi *info* e *sign* della risposta e la chiave pubblica DSA del TSA Server). Nel caso in cui la firma è verificata la marca viene salvata su file (nella cartella /marche aggiungendo il suffisso .marca al file di partenza) altrimenti viene generato un errore con conseguente negazione del salvataggio della marca.
2. *verifyOffline*
3. *verifyOnline*

## TSA Message

La classe TSAMessage incorpora sia il ruolo di richiesta da parte dell’utente che quello di risposta da parte del server TSA.

1. Nel caso di richiesta, il costruttore prende in input il Json contenente ID e digest del documento, la chiave privata DSA dell’user e la chiave pubblica RSA della TSA. Tramite il metodo *byteFromJson* vengono recuperati i byte del JSONObject e tali byte vengono firmati (la firma va nell’attributo *sign*) e cifrati (il messaggio cifrato va nell’attributo *info*) (Tali operazioni vengono effettuate con le chiavi sopra citate).
2. Nel caso di risposta il costruttore prende in input il Json (che questa volta contiene le informazioni calcolate dal TSAServer) e la chiave privata DSA della TSA. Tramite il metodo *byteFromJson* vengono recuperati i byte del JSONObject e tali byte vengono salvati nell’attributo *info* e firmati (la firma va nell’attributo *sign*).

## TSA

Tale classe incapsula tutte le elaborazioni svolte da un server TSA.

La classe mette a disposizione due costruttori: entrambi prendono in input il file su cui poi andare a pubblicare gli HashValue e SuperHashValue, ma viene data la possibilità, attraverso uno dei due, di poter riprendere lo stato di una computazione precedente caricando le informazioni di cui sopra da file. Il timeframe sarà dato dalla lunghezza del JSONArray meno l’elemento 0, e il serial number da timeframe\*8, essendo questo il numero di richieste (vere o fittizie) processate in un dato istante temporale.

Si riporta in basso il metodo responsabile di dare il via a tutte le operazioni svolte dal server TSA, chiamato dall’istanza di TimeStampManager:

public ArrayList<TSAMessage> **generateTimestamp**(ArrayList<TSAMessage> requests) {

this.mt = new **MerkleTree**();

this.timeframe += 1;

ArrayList<JSONObject> partialResponses = **createResponses**(requests);

**computeHashValues**();

ArrayList<String> merkleInfo = mt.**buildInfo**();

**saveHashValues**();

return **finalizeResponses**(partialResponses, merkleInfo);

}

Il metodo:

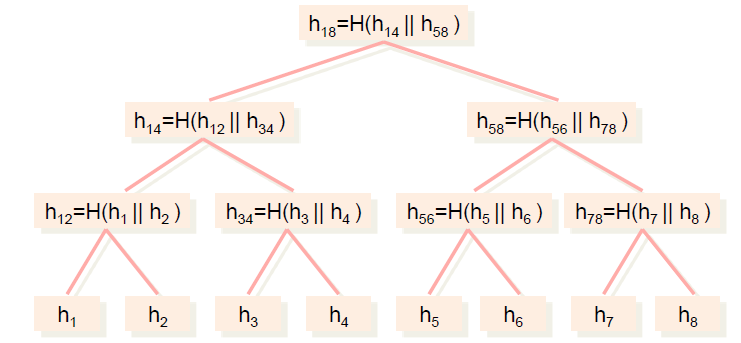
1. Riceve le richieste come un ArrayList di TSAMessage, e viene inizializzato un nuovo timeframe: questo prevede di istanziare un nuovo MerkelTree, sovrascrivendo il vecchio (non è utile tenerli memorizzati in memoria, giacchè alla fine l’unica informazione di interesse è il rootHashValue), e incrementare un contatore interno;
2. Il metodo *createResponses:*
   1. Per ogni richiesta, il TSA la decifra con la propria chiave RSA privata e verifica la firma con la chiave pubblica DSA del mittente. Il server TSA dispone di un proprio KeyRing privato, mentre per le chiavi pubbliche si ricorre al PublicKeyManager. Qualora dovessero accadere errori in fase di verifica della firma o decifratura, viene lanciato un messaggio a schermo e la richiesta viene semplicemente ignorata;
   2. Per ogni richiesta valida, viene costruito il JSONObject ricorrendo al metodo *makeResponseInfo*. Tale JSONObject andrà a costituire il campo informativo del TSAMessage di risposta e contiene:
      1. Il Timestamp, generato ricorrendo alla classe TimeStamp di java.sql: il Timestamp viene generato riferendosi al tempo attuale di sistema (espresso in millisecondi), incrementato di 10000\*serialNumber, per assicurarsi che le richieste siano sufficientemente distanziate in ordine temporale;
      2. L’ ID dell’utente che ha generato la richiesta;
      3. Il numero seriale della richiesta, incrementato dal TSA a ogni richiesta pervenuta, anche per le richieste fittizie da esso aggiunte qualora ci fossero errori o pervenissero richieste in numero inferiore rispetto al massimo consentito;
      4. Il digest calcolato dall’utente sul proprio documento;
      5. Il digest calcolato dal TSA, ottenuto concatenando il digest dell’utente con il Timestamp e applicandovi sopra la stessa funzione di hash;
      6. Il timeframe, che sarà usato dall’utente in fase di verifica.
   3. Vengono effettuati gli inserimenti all’interno del MerkelTree. Qualora le richieste risultanti fossero in numero inferiore al massimo consentito, vengono inseriti all’interno dell’albero degli hash casuali, in particolare vengono generati casualmente dei byte, concatenati con un Timestamp comunque generato, e applicandovi la funzione di hash. La randomness assicura che non vi sia alcuna forma di staticità o dipendenza utilizzabile per improntare un attacco, mentre l’uso dei timestamp lega i nodi fittizi all’arco temporale in cui essi sono stati generati, uniformandosi a quanto fatto per i nodi validi;
3. Il metodo *computeHashValues* provvede alla costruzione del MerkelTree, invocandone il metodo opportuno, al calcolo e al salvataggio del e del , ottenuto come: . I due valori vengono salvati in un JSONArray attributo della classe: la posizione i-esima del JSONArray corrisponde al timeframe i-esimo, e il valore corrispondente è un JSONObject con i campi *HashValue* e *SuperHashValue* per quel timeframe. Nota: Il metodo viene invocato anche per calcolare l’ all’atto della prima istanziazione di TSA;
4. Vengono calcolate le informazioni necessarie, per ogni nodo foglia del MerkleTree, per ricostruire l’HashValue;
5. Viene salvato su file il JSONArray contenente gli HV e SHV fino a quel momento: questo equivale a renderli pubblici;
6. Le risposte formate al punto 2 non sono complete: manca infatti l’HashValue per quel timeframe e le informazioni per poterlo ricostruire e verificare poi che coincidano. Vengono inserite nel JSONObject di risposta tali informazioni, e viene invocato il costruttore di TSAMessage che provvede a firmare l’oggetto JSON con la chiave DSA privata del TSA.

Il TSA è stato implementato in maniera tale da processare le richieste come una coda. Le richieste arrivano in ordine, vengono elaborate e le risposte generate nello stesso ordine.

Le eccezioni vengono gestite visualizzando a schermo lo stack trace e chiudendo l’applicazione: ove necessario, prima di chiudere l’applicazione, vengono salvati gli HV e SHV su file in maniera tale da poter ripristinare lo stato della computazione poi.

## Merkle Tree

Il numero massimo di richieste processabili dal TSA in un dato timeframe è pari a 8. Si è pertanto realizzato il Merkle Tree ricorrendo ad un array di byte bidimensionale di dimensione fissata a 15. In particolare, a fronte della struttura ad albero sottostante:



L’array realizzato si configura come segue:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **h1** | **h2** | **h3** | **h4** | **h5** | **h6** | **h7** | **h8** | **h12** | **h34** | **h56** | **h78** | **h14** | **h58** | **h18** |

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Ove gli elementi corrispondenti ai nodi foglia dell’albero sono:

Passiamo in rassegna le operazioni che la classe MerkleTree consente di effettuare:

* *public void* ***insert****(byte[] elem, byte[] timestamp):* Il metodo consente di generare i nodi foglia dell’albero e riceve come parametro l’hash del documento dell’utente e la marca temporale generata dal server TSA come array di byte. I due array vengono concatenati mediante il metodo *arrayConcat* di **ByteUtils**, all’array risultante viene applicata nuovamente la funzione di hash, e viene inserito automaticamente nella prima foglia libera all’interno dell’albero;
* *public byte[]* ***buildMerkleTree****():* Una volta completato l’inserimento dei nodi foglia si procede alla costruzione dell’interno albero invocando questo metodo. Partendo dalla posizione 0 dell’array, gli hash vengono concatenati a due due, ad essi viene di nuovo applicata la funzione di hash, e inseriti nelle posizioni da 8 in poi. L’ultima operazione produrrà la radice dell’albero che viene restituita dalla funzione come Root Hash Value;
* *public ArrayList<String>* ***buildInfo****():* Il metodo costruisce, per ciascun nodo foglia, le informazioni necessarie a poter calcolare il Root Hash Value. In particolare, noto l’hash del nodo foglia i-esimo, sono necessari:
  + L’hash del fratello;
  + L’hash del fratello del padre;
  + L’hash del fratello del nonno.

E’ facile notare che, per come è stato costruito l’array, il fratello di un nodo corrisponde alla posizione successiva all’interno dell’array, se il nodo occupa una posizione pari, altrimenti corrisponde alla posizione precedente; il padre di un nodo invece è dato da se il nodo si trova in posizione pari, altrimenti è pari al padre del fratello, se si trova in posizione dispari.

Ancora, l’ordine di concatenazione dei vari hash, segue il seguente pattern:

|  |  |
| --- | --- |
| h1 | dx,dx,dx |
| h2 | sx,dx,dx |
| h3 | dx,sx,dx |
| h4 | sx,sx,dx |
| h5 | dx,dx,sx |
| h6 | sx,dx,sx |
| h7 | dx,sx,sx |
| h8 | sx,sx,sx |

In conclusione, per ogni nodo foglia, viene costruita una stringa con il seguente formato:

*“hash1,pos1,hash2,pos2,hash3,pos3”*

Dove gli hash e le posizioni di concatenazione vengono ottenute ricorrendo ai metodi *sibling, parent, evalpos1, evalpos2, evalpos3* che ricevono in ingresso la posizione del nodo foglia attuale che si sta processando. La stringa così costruita per ogni nodo foglia viene inserita in un ArrayList di String e restituita dal metodo al chiamante.

## PublicKeyManager

Durante l’esecuzione della simulazione si presenta diverse volte la necessità, da parte di un utente, di ottenere una chiave pubblica associata ad un altro utente; pertanto è stata realizzata la classe ***PublicKeyManager***, con lo scopo di simulare un provider in grado di fornire a un qualsiasi utente le chiavi pubbliche di altri utenti.

La classe ***PublicKeyManager***, modellata come un singleton ed inizializzata con un file contente tutte le mappature necessarie tra utenti e chiavi, fornisce, tramite il metodo ***getPublicKey****(String user, String keyId),* la possibilità di recuperare una specifica chiave pubblica associata ad un certo utente.

Il file di inizializzazione utilizzato all’atto della creazione del singleton rappresenta, ai fini della simulazione, una sorte di database contenente le chiavi pubbliche degli utenti.

Esso è formattato secondo il formato JSON e contiene un JSONObject per ogni utente all’interno del quale sono presenti tanti campi quante sono le chiavi associate a quell’utente.

Gli utenti sono identificati tramite un identificativo univoco costituito da una stringa, i campi che rappresentano le chiavi sono invece identificate da una stringa che segue la seguente formattazione:

Key/*TipoChiave*/*DimensioneInBit*/*TipoDiServizio*

Dove:

* *TipoChiave* indica il tipo di chiave a cui ci si riferisce. Es. RSA, DSA etc
* *DimensioneInBit* indica la dimensione della chiave in bit
* *TipoDiServizio* indica il servizio per cui viene utilizzata quella chiave (es. dirigenza, insegnamento, etc.), di default è “Main”.

## Keyring

Per quanto concerne la memorizzazione delle chiavi private e delle password dell’utente è messo a disposizione, attraverso le classi ***KeychainUtils*** e **Keychain**, un meccanismo che consenta di memorizzarle su disco in maniera cifrata e di recuperare le password e le chiavi in un secondo momento, a partire da una password. Sono consentite inoltre la memorizzazione di chiavi e password di lunghezza eterogenea e la presenza di più chiavi o password dello stesso tipo/servizio ma per scopi/account differenti.

Anche per il Keyring è stato utilizzato il formato JSON per la rappresentazione delle informazioni in particolare esso è costituito da un JSONObject i cui campi rappresentano

## Note sul test

Nel main sono state realizzate due funzioni che consentono di alterare il contenuto delle marche ricevute dagli utenti o del file contenente gli HV e SHV pubblicato dal server TSA. In particolare:

*public static void* ***alterHashValuesFile****(String fieldToAlter, String regex, String replace, String hashFile, int timeframe)*

*public static void* ***alterUserTimestamp****(String fieldToAlter, String regex, String replace, String marcaFile)*

Tali funzioni sono da intendersi simulative del fatto che il server TSA ha collocato nella marca, o nella catena di HV e SHV, delle informazioni false, che l’utente può immediatamente riscontrare.

Dall’alterazione effettiva della marca, o perché intercettata e modificata sul percorso di rete tra utente e server TSA, o perché corrotta per qualche motivo localmente sul pc dell’utente, ci si protegge verificando che la firma apposta dalla TSA sia valida (si suppone per semplicità che la chiave di firma DSA del server TSA non cambi mai).

Pertanto, il metodo ***verifyTSASign*** di *TimestampManager*, va chiamato a rigore nei metodi ***verifyOnline***e***verifyOffline***: ovviamente, questo non avrebbe consentito di simulare lo scenario in cui la TSA falsifica le informazioni, per cui tale metodo è stato reso pubblico e chiamato dal main di test.

Infine, nel metodo ***generateRequest*** di *TimeStampManager* è presente una sezione di codice commentata: decommentandola si simula il caso in cui la richiesta ricevuta dal server TSA è corrotta (viene fatto per semplicità generando una chiave RSA a caso per la TSA). Correttamente la richiesta viene ignorata e la TSA procede nella sua computazione. Ovviamente si noti che alla prima esecuzione del main con tale sezione di codice non commentata, verranno generate delle eccezioni perché le marche afferenti a quelle richieste non saranno presenti (come dovrebbe essere); viceversa, se si decide di decommentare in una esecuzione successiva alla prima, il test andrà a lavorare con le marche vecchie, che non vengono da esso cancellate.