APCM

Di Muzio, Maddaloni

February 2023

1 Contesto e finalità

In questa sezione spieghiamo quale è il contesto nel quale immaginiamo di operare. Si noti che le scelte crittografiche che sono state fatte sono fondate sul contesto e sulle finalità di un ipotetica applicazione del progetto.

Le ipotesi che abbiamo fatto sono le seguenti:

- Il server è onesto, ma curioso. Quindi, non possiamo permettere che venga a conoscenza di segreti che si scambiano i clients.
- I clients e il server non hanno accesso a computer quantistici. Quindi, non è stato necessario creare una PKI post-quantum. Inoltre, avrebbe rallentato eccessivamente il processo di public key menagement.
- I clients hanno già effettuato un processo di autenticazione. Quindi, non è stato necessario inserire un processo di controllo di password o altri algoritmi di verifica dell'identità.
- Le informazioni devono avere un valido livello di confidenzialità. Quindi, è stato utilizzato AES perché è stato ritenuto un valido compromesso tra velocità e sicurezza.

Un esempio pratico di possibile applicazione potrebbe essere quello di un'azienda dove i dipendenti si devono scambiare velocemente delle informazioni sicure senza dover fidarsi di applicazioni di messaggistica esterne all'azienda. Utilizzando la nostra chat, le informazioni non usciranno dall'azienda e manterranno un valido livello di confidenzialità anche all'interno della stessa grazie ad AES128. Inoltre, si noti che la seconda ipotesi, risulta essere particolarmente ragionevole anche nel prossimo futuro essendo che, a meno di casi eccezionali, le aziende non avranno libero accesso a tali macchine.

2 Interfaccia grafica

L'applicazione è suddivisa in tre differenti file fxml gestiti dallo stesso controller CLientThreadLaucher.java e da due distinti file css: style.css e readyStyle.css.

I file fxml sono i seguenti: ready.fxml, set.fxml, go.fxml.

"ready.fxml" è l'interfaccia che accoglie il client all'apertura dell'applicazione. Da qui il client potrà decidere se entrare effettivamente nell'applicazione ed iniziare la procedura di login, oppure chiuderla.

"set.fxml" è l'interfaccia che viene mostrata nel caso in cui il client abbia scelto di entrare dal file ready.fxml. Ora il client può decidere l'indirizzo IP a cui connettersi e cioè quello da cui il server sta offrendo il servizio (di default viene visualizzato l'indirizzo della macchina locale). Oltre all'indirizzo, il client deve decidere il nome con cui farsi riconoscere all'interno della chat. Questo nome deve soddisfare due condizioni. Una controllata in locale: il nome non può essere più lungo di 16 caratteri. E la seconda controllata dal server: non ci siano altri clients collegati con lo stesso nome. Inoltre, il client ha sempre a disposizione un bottone che lo fa uscire dall'applicazione, tornando alla schermata precedente. Se invece il client seleziona il bottone di connessione e i parametri inseriti (IP e username) risultano corretti, allora verrà stabilita la connessione con il Server aprendo il Socket e gli Streams per input e output di dati (bytes nel nostro caso); infine verrà aperta la schermata per poter effettivamente iniziare ad inviare e ricevere messaggi.

"go.fxml" è l'interfaccia vera e propria dell'applicazione. In questa interfaccia il client vede in alto a sinistra il nome utente scelto. Può scrivere il messaggio nell'area di testo superiore e può decidere l'utente a cui inviare il messaggio attraverso la tendina (di default il messaggio sarà pubblico, i.e. il destinatario sarà "All"). Quest'ultima viene automaticamente aggiornata ogni qual volta venga eseguito un nuovo accesso o un client abbandoni la chat; nel secondo caso, per ragioni di sicurezza, il destinatario viene modificato impostando il client stesso connesso. Per inviare il messaggio, è sufficiente premere il bottone di invio. I propri messaggi e la cronologia dei messaggi dei clients connessi durante la propria sessione vengono visualizzati nell'area di testo inferiore. Infine, può con l'ultimo pulsante terminare la sessione: Socket e DataStreams venogono chiusi e il client viene reindirizzato alla scermata iniziale in cui potrà effettivamente chiudere l'applicazione.

3 Scambio di dati e scelte crittografiche

La sicurezza della nostra chat, come già specificato, è basata sull'utilizzo del block cipher AES128 in modalità CBC per la parte simmetrica e ECDH per la parte asimmetrica.

In primo luogo abbiamo stabilito quali dati scambiati verranno criptati ed in che modo: ogni client potrà decidere di inviare messaggi all'intero gruppo connesso o ad un utente, per questo vengono generati i parametri chiave-IV sia

per ogni possibile coppia di utenti, in modo da criptare la comunicazione one-to-one, che per la comunicazione one-to-all. Oltre a voler garantire un sistema di crittografia end-to-end abbiamo aggiunto un ulteriore protocollo volto ad evitare tentativi di alterazione dei pacchetti scambiati tra Server e Clients: una volta che il messaggio è stato cifrato dal server, viene inserito in un pacchetto dati da inviare al Server che sarà stato ulteriormente criptato con la chiave simmetrica scambiata precedente con il Server. Infatti, al momento del Login di ogni client verrà effettuato uno scambio di parametri per AES anche con il Server.

Nel caso di scambio di paramentri segreti tra coppie (client-client o client-server) useremo l'algoritmo Diffie-Hellman su Curve Ellittiche, invocando il provider Bouncy Castle; mentre i parametri di gruppo sono generati dal primissimo client connesso mediante la classe SecureRandom dell'API di Java, successivamente ogni nuovo client riceverà tali parametri da uno degli altri clients (colui che in quel momento risulta essere il primo client nel "registro" di utenti del server). I parametri vengono salvati da parte di ogni Client in una Hashtable in cui le chiavi sono Stringhe che rappresentano i nomi utenti e i valori sono oggetti di una nuova classe, Secret, per la quale abbiamo definito funzioni ad hoc che ritornano le Round Keys e l'IV; da parte del server in una LinkedHashMap nella quale al nome utente di un client viene associato come valore un oggetto della nuova classe ChiaviS che consta di un DataOutputStream, un Secret, un array di bytes per la chiave pubblica. La scelta di una LinkedHashMap è dovuta al fatto che essa preserva l'ordine di inserimento degli elementi (il che ci è utile nel momento in cui il server manda la richiesta di inviare i paramentri di gruppo per un nuovo client ad un utente già connesso, il primo), per il resto ha la stessa complessità per l'aggiunta/rimozione/ricerca di elementi che si avrebbe utilizzando una HashTable (costante). Tale procedura di generazione e scambio di chiavi è interamente gestita nella fase di login di ogni nuovo client, a seguire verranno attivati i Threads di ascolto e potranno iniziare ad inviare e ricevere messaggi.

Una volta decise quali informazioni andranno criptate abbiamo delineato la struttura dei vari pacchetti dati nei casi di Login, Logout, scambio parametri segreti di gruppo, scambio messaggi, facendo in modo che gli array di bytes inviati siano sempre divisibili in blocchi da 16 per permettere la crifratura/decifratura con il block cipher selezionato. Tale struttura è riassunta nelle seguenti immagini 3: innanzitutto viene inviata la modality, a seguire i blocchi successivi in modo che i client, a seconda della modality, si preparino a ricevere e dunque decifrare un dato numero di blocchi

A questo punto abbiamo preso in considerazione una potenziale debolezza della modalità CBC scelta: criptare uno stesso messaggio utilizzando con la stessa chiave lo stesso initial vector più di una volta porta ad ottenere lo stesso ciphertext, tale risultato renderebbe la chat vulnerabile contro known-plaintext attacks (ad esempio l'attaccante potrebbe intuire la modality del pacchetto dati inviato ed associare al corrispondente array di bytes di quella modality il ciphertext). Una prima soluzione valutata è stata aggiornare dopo ogni cifratura l'IV, ad esempio mediante uno XOR con l'ultimo blocco del ciphertext, ma ciò avrebbe portato a problemi di sincronizzazione. Dunque abbiamo deciso di

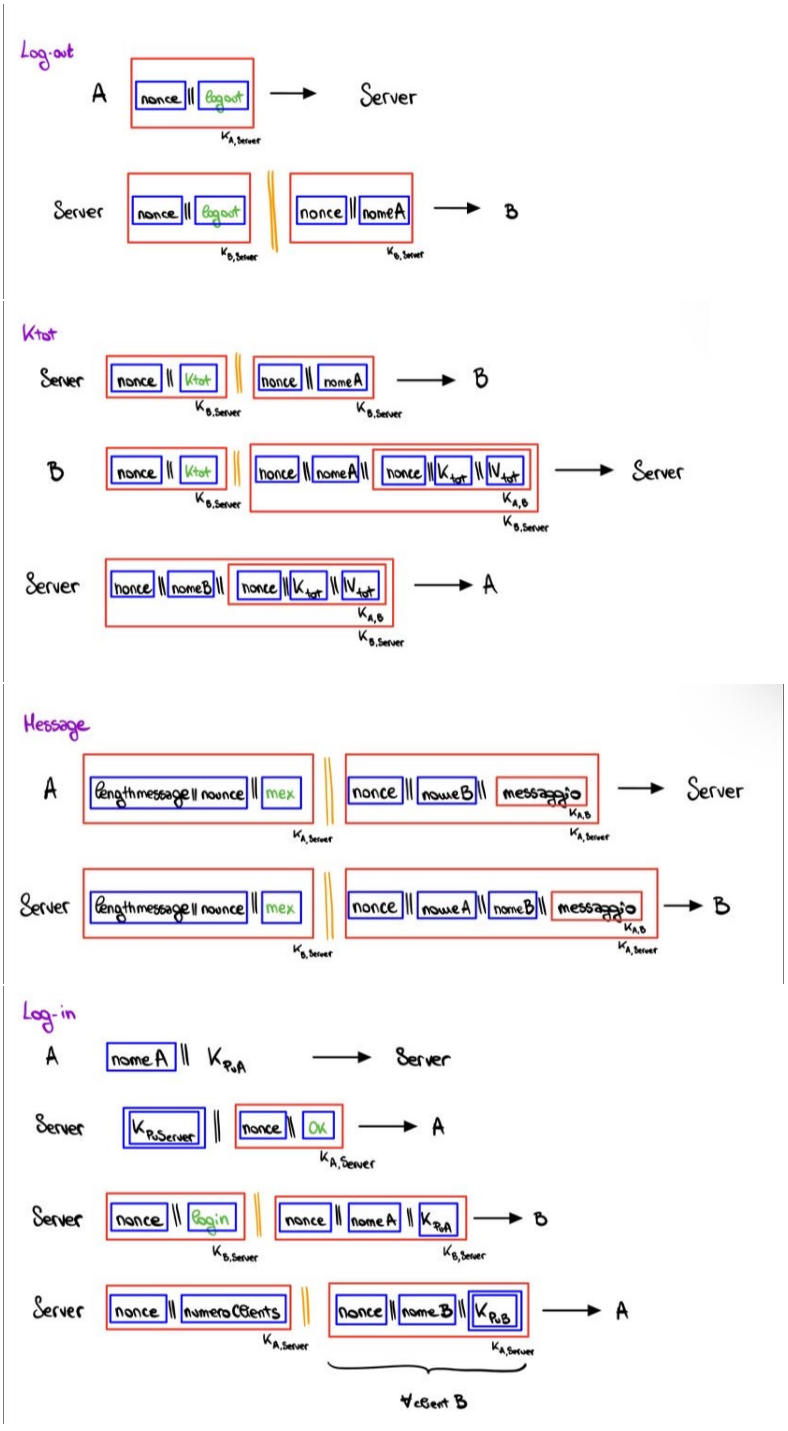

Figura 1: Legenda: Quadrato rosso = cifrati con chiave specificata in basso a destra. Quadrato blu = messaggio lungo 16 byte. Le parole in verde sono simboli noti che indicano la modalità che si deve analizzare.

concatenare un nonce da 16 bytes all'inizio di ogni blocco dati scambiato tra server e client: questa scelta, inoltre, rende inefficiente qualunque tentativo di chosen plaintext/ciphertext attack dal momento che mai saranno mandati due plaintexts identici grazie all'aggiunta di bytes randomici.

Per quanto concerne le funzioni di encryption e decryption, sono state implementate in C nella libreria AESLib.c e vengono chiamate come funzioni native della classe Terabitia che fa appunto da "ponte" con il linguaggio Java mediante la Java Native Interface. In tale classe viene inoltre definita a parte una funzione di setup che si occupa, una volta avvenuto lo scambio di chiave e IV, di eseguire la generazione delle Round Keys: essa verra chiamata dalla classe Secret, infatti una volta che due parti eseguono lo scambio di paramentri segreti con ECDH inizializzano un oggetto Secret mediante un array di 32 byte, il risultato dell'algoritmo, e creano, a partire dalla prima metà dell'input, un array di 160 bytes contenente le 10 chiavi, mediante appunto la funzione Setup, e un array da 16 che è semplicemente la copia della seconda metà dell'input.

4 Conclusioni e possibili aggiornamenti

Come già spiegato nella prima sezione, non è stato implementato un algoritmo post-quantum in quanto la sicurezza aggiunta non valeva la velocità che si perde. Se si fosse voluto utilizzare, come è stato da noi testato, l'algoritmo vincitore della gara effettuata da NIST, lo scambio di chiavi presupponeva uno scambio di 1024 byte contro i 32 dell'algoritmo che viene utilizzato da noi: ECC. Utilizzando le curve ellittiche con chiavi da 256 bits, infatti, il livello di sicurezza crittografica è lo stesso di un cifrario simmetrico a 128 bits, come nel nostro caso. Si noti, che la sicurezza rimane comunque particolarmente elevata se non si suppone l'utilizzo dell'algoritmo di Shor.

Certamente potrebbero essere apportate delle ottimizzazioni nei tipi di oggetti Java utilizzati, in particolare nella gestione di array di bytes scambiati che vengono ripetutamente copiati o sovrascritti, ma per ora si è rivelata la soluzione migliore per rispettare un certo schema nella struttura dei blocchi di dati diversi e paddare messaggi, nomi, simboli per le modalità così da poter lavorare sempre con blocchi da 16 bytes.

Infine si può pensare ad una gestione più completa della PKI aggiungendo anche protocolli di autenticazione e produzione di certificati che leghino la chiave pubblica all'effettivo proprietario.