SECURITY IN NETWORKED COMPUTING SYSTEMS

David Costa
Daniele Battista

A.Y. 2013/2014

Indice

Capitolo 1

Specifiche

Si consideri un sistema distribuito di tipo cliente-servitore in cui ciascun cliente A condivide una chiave segreta K_{ab} con il servitore B. Supponendo di essere in una situazione di mutual-trust, si specifichi, si analizzi, si progetti ed, infine, si implementi un protocollo crittografico che soddisfa i seguenti requisiti:

- al termine dell'esecuzione del protocollo, viene stabilita una chiave di sessione, K'_{ab} , tra A e B;
- al termine dell'esecuzione del protocollo, il cliente A ritiene che il server B ha la chiave di sessione K'_{ab};
- al termine dell'esecuzione del protocollo, il server B ritiene che il cliente A ha la chiave di sessione K'_{ab};
- la chiave di sessione K'_{ab} viene generata dal server B.

La specifica del protocollo deve mettere chiaramente in evidenza le ipotesi sotto le quali il protocollo funziona correttamente.

L'implementazione deve comprendere la realizzazione di un prototipo in cui il server ed il cliente si scambiano del materiale (testo o binario) cifrato con la chiave di sessione K'_{ab}.

Capitolo 2

Implementazione del cloud

Il progetto consiste nello sviluppo di un software di cloud storage, che offre un servizio di file hosting. Gli utenti sono identificati dal server tramite *username* ed ogni utente registrato ha a disposizione, sul server, una cartella (all'interno di *usr*) con il proprio spazio di hosting, dove posizionare e gestire i file. Il servizio consente di effettuare upload e download dei dati, ricevere una lista con i propri file sul cloud e di poterli eliminare dal server.

Un utente, per manipolare i propri file, deve posizionarli all'interno della propria cartella *file*, dove verranno poi anche scaricati in fase di download.

Per semplicità supponiamo che gli utenti presenti siano già registrati, senza il bisogno di dover effettuare la registrazione manualmente, le loro cartelle siano già presenti (così come le loro chiavi) e che un utente non registrato non possa usufruire del servizio.

Il protocollo di trasferimento è simile a FTP (File Transfer Protocol): trasmissione di dati tra host basata su TCP e dati trasferiti in maniera affidabile ed efficiente. A differenza di FTP, che utilizza due connessioni separate per gestire comandi e dati, qui abbiamo una sola connessione per entrambi.

2.1 Lato client

Il client deve essere avviato con la seguente sintassi:

`./client <host remoto> <porta> <username>`

dove:

- `<host remoto>` è l'indirizzo dell'host su cui è in esecuzione il server;
- `<porta>` è la porta su cui il server è in ascolto;
- `<username>` è il nome utente con il quale il client viene riconosciuto dal server.

I comandi disponibili per l'utente sono:

- `1 nome_file`: invia il file `nome_file` al server.
- `2`: riceve la lista dei propri file presenti sul server.
- `3 nome_file`: scarica il file `nome_file` dal server.
- `4 nome_file`: elimina il file `nome_file` dal server.
- `5`: disconnette l'utente dal server.

Gli errori che si possono verificare sono:

- errori a livello protocollare.
- `nome_file` errato.
- `nome_file` da scaricare o eliminare non presente sul server.

Più in dettaglio il client, in fase di upload, invierà al server nome e dimensione del file (in modo da poterglielo far ricevere correttamente), seguiti dal trasferimento dei dati da caricare.

Viceversa per il download, dove i dati saranno trasmessi da server a client, e dove il primo effettuerà un controllo, una volta ricevuto il nome del file da trasferire, per verificare se quel dato è effettivamente presente sul server.

Similmente accade per la rimozione dei dati, dove anche qui l'utente comunica il nome del file da cancellare che, se presente, verrà eliminato dal proprio spazio di hosting su server.

Infine, per gli utenti è possibile ricevere una lista di tutti i propri file presenti sul server.

2.2 Lato server

Il programma `server` si occupa di gestire le richieste provenienti dai client. Il server tramite l'uso della select, accetterà nuove connessioni TCP, registrerà nuovi utenti e gestirà le richieste dei vari client per il trasferimento file.

La sintassi del comando è la seguente:

```
./server <porta>
```

dove:

- `<porta>` è la porta su cui il server è in ascolto.

L'indirizzo IP è già definito all'interno del programma .

I dati degli utenti vengono gestiti lato server tramite delle subdirectory contenute nella cartella `usr`. All'interno della cartella sono presenti tante directory quanti gli utenti registrati; ogni directory è nominata con il nome del client `nome_client` (che l'utente segnala al momento della connessione) ed al suo interno sono presenti i dati del client: rappresenta dunque lo spazio di hosting dell'utente.

Per semplicità supponiamo che gli utenti che si collegheranno al server saranno già registrati ed avranno un proprio spazio (quindi la connessione di un utente non registrato causerà un errore). Inizialmente sono registrati gli utenti `daniele` e `david`.

2.3 Utilizzo della sicurezza

I messaggi scambiati tra client e server saranno di due tipi:

- **comando**, per comunicare al server l'azione da compiere;
- **dati**, contenenti i file scambiati o la lista dei file.

I dati sono personali (così come la lista file) e verranno cifrati prima dell'invio, in modo da mantenere riservatezza ed autenticità, per poi essere decifrati quando ricevuti dall'altro lato.

I comandi indicano il servizio richiesto dall'utente ed anche loro, prima dell'invio, verranno cifrati.

Capitolo 3

Protocollo

Il protocollo che abbiamo utilizzato è ispirato a quello di Needham-Schroeder, con la semplificazione di avere la comunicazione limitata a client-server e tre messaggi anziché cinque.

Il protocollo reale è definito come segue:

$M1 : A \rightarrow B$	A, B, N_a
$M2 : B \rightarrow A$	$\{B, N_a, K'_{ab}\}_{K_{ab}}, \{N_b\}_{K'_{ab}}$
$M3 : A \rightarrow B$	$\{N_b - 1\}_{K'_{ab}}$

3.1 Analisi BAN

3.1.1 Obiettivi

$$K_{ab} = \text{chiave a lungo termine}$$

$$K'_{ab} = \text{chiave di sessione}$$

Key authentication:

$$A| \equiv A \overset{K'}{\leftrightarrow} B$$

$$B| \equiv A \overset{K'}{\leftrightarrow} B$$

Key confirmation:

$$A| \equiv B| \equiv A \overset{K'}{\leftrightarrow} B$$

$$B| \equiv A| \equiv A \overset{K'}{\leftrightarrow} B$$

3.1.2 Assunzioni

Ipotesi vecchia chiave:

$$A| \equiv A \overset{K}{\leftrightarrow} B$$

$$B| \equiv A \overset{K}{\leftrightarrow} B$$

$$A| \equiv B| \equiv A \overset{K}{\leftrightarrow} B$$

$$B| \equiv A| \equiv A \overset{K}{\leftrightarrow} B$$

Ipotesi freschezza nonce:

$$A| \equiv \#(N_a)$$

$$B| \equiv \#(N_b)$$

Ipotesi freschezza nuova chiave:

$$B| \equiv \#(A \overset{K'}{\leftrightarrow} B)$$

Ipotesi abilità di B:

$$A| \equiv B \Rightarrow A \overset{K'}{\leftrightarrow} B$$

$$A| \equiv B \Rightarrow \#(A \overset{K'}{\leftrightarrow} B)$$

3.1.3 Protocollo idealizzato

$M2 : B \rightarrow A$	$\left\{N_a, A \overset{K'}{\leftrightarrow} B, \#(A \overset{K'}{\leftrightarrow} B)\right\}_{K_{ab}}, \left\{N_b, A \overset{K'}{\leftrightarrow} B\right\}_{K'ab}$
$M3 : A \rightarrow B$	$\left\{N_b, A \overset{K'}{\leftrightarrow} B\right\}_{K'_{ab}}$

3.1.4 Significato dei messaggi

M2: dopo aver ricevuto N_a, B dice che K'_{ab} è adatta per parlare con lui e che è generata nell'attuale esecuzione del protocollo. Dopodiché, B dice che sta utilizzando la nuova chiave di sessione K'_{ab}.
M3: dopo aver ricevuto N_b, A dice che K'_{ab} è adatta per parlare con B e che da ora utilizzerà questa.

3.1.5 Postulati

Message meaning rule:

$$\frac{P| \equiv Q \overset{K'}{\leftrightarrow} P, P \triangleleft \{X\}_K}{P| \equiv Q| \sim X}$$

Nonce verification rule:

$$\frac{P| \equiv \#(X), P| \equiv Q| \sim X}{P| \equiv Q| \equiv X}$$

Jurisdiction rule:

$$\frac{P| \equiv Q| \equiv X, P| \equiv Q \Rightarrow X}{P| \equiv X}$$

3.1.6 Analisi dei messaggi

A riceve M2:

Dato che $A| \equiv A \overset{K}{\leftrightarrow} B$ e $A \triangleleft \{Messaggio\}_{K_{ab}}$, dal postulato *message meaning rule* si ricava che $A| \equiv B| \sim Messaggio$. $Messaggio$ contiene N_a e K'_{ab}, così, poiché $A| \equiv \#(N_a)$, dal postulato *nonce verification rule* si deriva che $A| \equiv B| \equiv Messaggio$. Si deduce che $A| \equiv B| \equiv A \overset{K'}{\leftrightarrow} B$ e dato che, per ipotesi, $A| \equiv B| \Rightarrow A \overset{K'}{\leftrightarrow} B$, per il terzo postulato, *jurisdiction rule*, $A| \equiv A \overset{K'}{\leftrightarrow} B$.

Inoltre, poiché $A| \equiv B| \equiv \#(A \overset{K'}{\leftrightarrow} B)$ e per ipotesi $A| \equiv B \Rightarrow \#(A \overset{K'}{\leftrightarrow} B)$, allora, ancora per il terzo postulato, si ricava che $A| \equiv \#(A \overset{K'}{\leftrightarrow} B)$.

B riceve M3:

Dato che $B| \equiv A \overset{K'}{\leftrightarrow} B$, essendo appena generata, e $B \triangleleft \{Messaggio\}_{K'_{ab}}$, allora B deduce che $B| \equiv A| \sim Messaggio$ e, dalla freschezza del $Messaggio$, che $B| \equiv A| \equiv Messaggio$. Ciò significa che $B| \equiv A| \equiv A \overset{K'}{\leftrightarrow} B$.

Capitolo 4

Implementazione della sicurezza

Gli utenti sono identificati sul server attraverso i loro username. Sia nel lato server che nel lato client è presente una cartella `key` che contiene i file, nominati con l'username dell'utente, contenenti la chiave a lungo termine K_{ab} del suddetto utente. Nella cartella `key` presente sul server, saranno contenuti tanti file quanti gli utenti registrati; in quella lato client sarà presente, ovviamente, un solo file per la propria chiave a lungo termine.

Si suppone che i client siano già registrati e, quindi, che i file contenenti le chiavi a lungo termine siano già presenti.

4.1 Generazione chiave a lungo termine

La chiave a lungo termine viene generata attraverso un programma scritto appositamente a questo scopo. La lunghezza della chiave, poiché utilizziamo come cifrario DES basato su ECB, viene fissata tramite un assegnamento a `EVP_DES_ECB`.

Il programma utilizzato per generare queste chiavi viene eseguito a riga di comando con la seguente sintassi:

```
./keygen nome_utente
```

Viene così generata la chiave per l'utente `nome_utente`. Per generare la chiave per tutti gli utenti che vogliono interagire con il server è necessario eseguire il programma per ognuno di loro (differenziando i nomi degli utenti).

4.2 Lato server

Il server gestisce gli utenti connessi tramite una lista di strutture `client_list` composte dai seguenti campi:

```
struct client_list{
  struct client_info client;
  struct client_list* next;
};
```

dove `next` è il puntatore al prossimo elemento e `client` è una struttura di tipo `client_info` contenente le informazioni dell'utente strutturata come segue:

```
struct client_info{
  int socket_fd;
  char* Name;
  char* usr_dir;
  unsigned char* session_key;
  unsigned char* Nb;
  unsigned char* Na;
  EVP_CIPHER_CTX* session_ctx;
  EVP_CIPHER_CTX* ctx;
  int statusHandshake;
  int M1, M2, M3;
};
```

I campi `ctx` e `session_ctx` sono, rispettivamente, il contesto per la cifratura con la chiave a lungo termine e quello per la cifratura con la chiave di sessione; `session_key` è la chiave di sessione (quella a lungo termine è salvata su file); `Na` e `Nb` sono i nonce utilizzati nei messaggi del protocollo; `M1`, `M2` e `M3` sono dei flag utilizzati al momento della connessione, durante il protocollo, per stabilire in quale fase del protocollo ci si trova e quale messaggio si attende o si deve inviare.

Durante la sua esecuzione, il server sfrutta le funzioni riportate di seguito:

```
int getKeyFromFile(unsigned char* long_term_key,
                   client_info* client);
```

Prende dal file nominato con l'username dell'utente `client` la chiave a lungo termine da utilizzare nella comunicazione con tale client e la salva all'interno di `long_term_key`; ritorna -1 in caso di errore, 0 altrimenti.

```
int doHandshakeProtocol(client_info* client, int phase);
```

Esegue il protocollo per stabilire con il client la chiave di sessione.

```
int manageOperation(int listening, fd_set* readSet,
                    fd_set* writeSet, fd_set* readSetTmp,
                    fd_set* writeSetTmp, int *maxFD);
```

Connette il server con il client se se ne collega uno nuovo, controlla se sono in arrivo delle richieste e gestisce parte del protocollo per la chiave di sessione. Ritorna -1 in caso di errore, 0 altrimenti.

```
int manage_request(client_info* client);
```

Riceve il comando dal client, lo decifra e discrimina quale funzione eseguire per fornire il servizio richiesto. Ritorna -1 in caso di errore, 0 altrimenti.

```
int saveRecivingFile(client_info* client);
```

Riceve, decifra e salva nell'apposita cartella il file dell'utente `client`. Ritorna -1 in caso di errore, 0 altrimenti.

```
int sendFile(client_info* client);
```

Controlla se il file richiesto è presente nella cartella dell'utente `client` e, in caso positivo, lo cifra e lo invia al client. Ritorna -1 in caso di errore, 0 altrimenti.

```
int send_list(client_info* client);
```

Genera una lista con i file dell'utente `client` presenti nella sua cartella, la cifra e la invia all'utente. Ritorna -1 in caso di errore, 0 altrimenti.

```
int removeFile(client_info* client);
```

Controlla se il file da rimuovere è presente nella cartella dell'utente `client` e, in caso positivo, lo cancella. Ritorna -1 in caso di errore, 0 altrimenti.

4.3 Lato client

Il client implementa la sicurezza utilizzando le seguenti funzioni:

```
int readKey(unsigned char * key, int maxlength,
            const char * fk);
```

Legge la chiave a lungo termine di dimensione `maxlength` dal file nominato `fk` e la inserisce all'interno di `key`; ritorna la dimensione della stringa letta su file.

```
int start_protocol(int sk, unsigned char** session_key,
                   int * session_key_len, char*argv[]);
```

Avvia il protocollo e stabilisce la chiave di sessione con il server. Ritorna -1 in caso di errore, 0 altrimenti.

```
int send_message(unsigned char* mex, int sk,
                 unsigned char* session_key,
                 int session_key_len);
```

Cifra ed invia al server il comando per specificare il servizio richiesto. Ritorna -1 in caso di errore, 0 altrimenti.

```
int send_file_crypt(const char* file_name, int sk,
                    unsigned char* session_key,
                    int session_key_len );
```

Cifra il file nominato `file_name` con la chiave di sessione `session_key` e lo invia al server. Ritorna -1 in caso di errore, 0 altrimenti.

```
int ask_for_the_list(int sk, unsigned char* session_key,
                     int session_key_len);
```

Richiede al server la lista dei propri file. Ritorna -1 in caso di errore, 0 altrimenti.

```
int recv_file_crypt( const char* file_name, int sk,
                     unsigned char* session_key,
                     int session_key_len );
```

Specifica al server il nome del file richiesto e, se presente, lo riceve e lo decifra. Ritorna -1 in caso di errore, 0 altrimenti.

```
int rmv_file(const char* file_name, int sk,
             unsigned char* session_key, int session_key_len);
```

Specifica al server il nome del file da eliminare dallo spazio di hosting dell'utente. Ritorna -1 in caso di errore, 0 altrimenti.