UNIVERSITÀ DEGLI STUDI DI PISA

Facoltà di Ingegneria

Progetto di Sicurezza

Autori

Alessandro Rosetti

Daniele Lazzarini

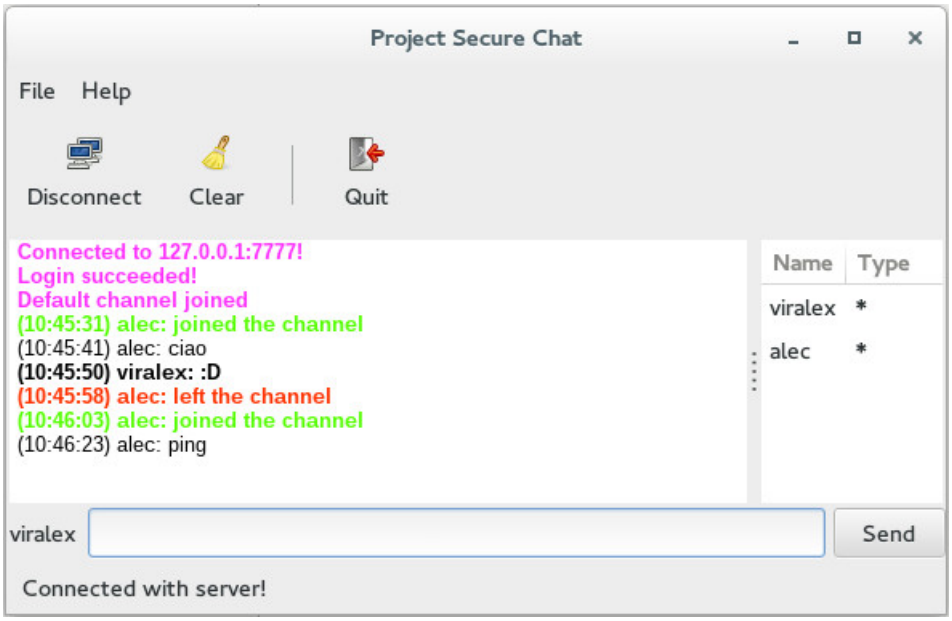
Anno accademico 2011-2012

Documento sviluppato con LaTeX
16 febbraio 2017

Indice

1 Introduzione

Il progetto illustrato in questo documento si chiama **pschat** e rappresenta un sistema client-server di chat ispirato al modello **IRC** (Internet Relay Chat) che integra caratteristiche di sicurezza.



1.1 Componenti e Tool

Il software sviluppato utilizza le seguenti librerie:

- **OpenSSL**: per AES, RSA (firma,verifica,cifratura,decifratura), SHA256.
- **LibConfuse**: per la gestione dei file di configurazione.
- **Sqlite3**: per la gestione del database utente.
- **Xmlite2**: per la gestione dei messaggi della chat.
- **GTK+3**: per l'interfaccia grafica dei client.

Il software è stato sviluppato utilizzando i seguenti tool:

- **Cmake**: per la gestione del meccanismo di build.
- **Valgrind**: per il controllo di eventuali memory-leak.
- **Merurial e TortoiseHG**: per il controlo di versione, coordinamento del lavoro.

1.2 Formato del pacchetto

Pacchetto **RSA PKCS#1 OEAP** viene usato solo nei primi messaggi di autenticazione:

```
[LEN]  RSA-ENCRYPTED{ [OPCODE][LEN][TIME][PAYLOAD] }
  2                     2      2     4   0-248/506
```

$Size = 2 + 256 = 258$ Bytes per chiavi 2048bit.
$Size = 2 + 512 = 514$ Bytes per chiavi 4096bit.

Pacchetto **AES-128/256 CBC** viene usato per cifrare la comunicazione tra client e server nella funzionalità di chat:

```
[LEN]   AES-ENCRYPTED { [IV] }{ [OPCODE][LEN][TIME][SEQ][PAYLOAD] }
  2                      16        2      2    4     4    0-65000
```

Il testo cifrato è multiplo della dimensione di un blocco AES di 16 Byte.
$Size_{min} = 2 + 16 + 16 = 34$ Bytes.
$Size_{max} = Size_{min} + 65000 + (65000\%16) \cong 64k$Bytes.

Il campo **TIME** evita attacchi a lungo termine scartando pacchetti con una data in unix-time maggiore di due giorni.
Il campo **SEQ** evita attacchi a corto termine come reply e injection.

2 Protocollo

L'autenticazione usa pacchetti cifrati con **RSA** in modalità **OEAP**.

2.1 Server

Il server ha una propria chiave privata e una pubblica (la pubblica è nota a priori per ogni client), è dotato di un database che contiene i nomi degli utenti e altre informazioni.
Il server conosce tutte le chiavi pubbliche degli utenti registrati al sistema.

- server.pem
- server.pub
- client_*.pub
- database's table user(name, user_type, reg_date);

2.2 Client

Il client ha una chiave privata e pubblica e conosce la chiave pubblica del server.

- client_${username}.pem
- client_${username}.pub
- server.pub

La chiave privata di ogni utente è cifrata in **AES256**, quando effettua il login inserisce la password che permette di utilizzarla.

2.3 Protocollo Base

Il seguente paragrafo mostra la procedura di autenticazione effettivamente implementata, i messaggi utilizzano il modello dei pacchetti descritto precedentemente.
La procedura di autenticazione permette di stabilire una chiave di sessione tra client e server. Tutti i numeri random sono generati con le funzioni della libreria OpenSSL.
Gli attori e le operazioni effettuate dal protocollo sono i seguenti:

- **S** : Server
- **C** : Client
- N_s : Server's nonce
- N_c : Client's nonce
- $E_s()$: Encrypt with server's public key.
- $E_c()$: Encrypt with client's public key.
- $E_{s^{-1}}(h(*))$: Sign with server's private key.
- $E_{c^{-1}}(h(*))$: Sign with client's private key.
- K_c : Client's public asymmethric key.
- K_s : Server's public asymmethric key.
- $K_{c^{-1}}$: Client's private asymmethric key.
- $K_{s^{-1}}$: Server's private asymmethric key.
- K_{k1} : Client's temp symmethric key.
- K_{k2} : Server's temp symmethric key.
- X_c : Client's partial session key.
- X_s : Server's partial session key.
- K_{sc} : Combined session key: $K_{sc} = (X_c \oplus X_s)$.
- R : Random number.

Il protocollo di autenticazione effettivamente implementato usa la seguente sequenza di messaggi.

M1 : $S \rightarrow C$
N_s

Il *server* genera e invia un nonce al *client* che si è appena connesso.

M2 : $S \leftarrow C$
E_s (E_{k1}), E_{k1} (N_c, N_s, User, X_c, $E_{c^{-1}}$ (h(N_c, N_s, User, X_c)))

Il **client** genera il suo nonce e una chiave simmetrica crittograficamente sicura X_c. Il *client* genera una chiave simmetrica crittograficamente sicura K_{k1} e la cifra con la chiave pubblica del server K_s. Successivamente usa la chiave generata per cifrare simmetricamente in CBC i dati necessari all'autenticazione:
il **nonce ricevuto** dal server in M1, un **proprio nonce**, il **nome utente**, la sua parte della **chiave di sessione** X_c e la **firma digitale** di tutti i precedenti campi eseguita utilizzando la propria chiave privata. (la sintassi della firma digitale è stata messa a puro scopo indicativo, non rappresenta l'effettiva implementazione di openssl che potrebbe includere eventuali funzioni di ridondanza)

M3 : $S \rightarrow C$
E_c (E_{k2}), E_{k2} (N_c, Response, X_s, $E_{s^{-1}}$ (h(N_c, Response, X_s))), $E_{K_{sc}}(N-1)$

Il nome utente permette di localizzare la corretta chiave pubblica del client con cui si sta comunicando. Se il server verifica la firma del precedente messaggio, l'utente non è già connesso, ha un nome valido ed è presente nel database può procedere con l'autenticazione.
Il **server** genera una chiave simmetrica crittograficamente sicura X_s.
Il **server** genera una chiave simmetrica crittograficamente sicura K_{k2} e la cifra con la chiave pubblica del *client* K_c. Successivamente usa la chiave generata per cifrare simmetricamente in CBC i dati necessari all'autenticazione:
l'altra parte della **chiave di sessione** X_s, il **nonce del client**, la **risposta**, e la **firma digitale** di tutti i precedenti campi eseguita utilizzando la propria chiave privata. *Client* e *server* possono calcolare indipendentemente la chiave di sessione $K_{sc} = (X_c \oplus X_s)$, per verificarne il funzionamento il server invia un **dato cifrato** con tale chiave decrementato di uno ricavato dai primi 4 byte del nonce del *client*.
Il dato cifrato con la chiave simmetrica servirà a testare il funzionamento della chiave appena calcolata.

M4 : $S \leftarrow C$
$E_{K_{sc}}$ (N - 2)

Se il *client* ha precedentemente verificato la firma digitale del *server*, e il dato cifrato con la chiave di sessione corrisponde alla parte estratta dal suo nonce decrementato di uno, risponde decrementando nuovamente il dato ricevuto, rimandandolo indietro sempre criptato con la chiave di sessione, dimostrando anche al server il funzionamento del canale.

2.4 Protocollo Reale

Il protocollo descritto precedentemente viene riassunto nei seguenti messaggi.

$$
\begin{aligned}
&M_1 : S \rightarrow C \quad N_s \\
&M_2 : S \leftarrow C \quad \{K_{k1}\}_{K_s}, \left\{N_s, N_c, X_c, \{N_s, N_c, X_c\}_{K_{c^{-1}}}\right\}_{K_{k1}} \\
&M_3 : S \rightarrow C \quad \{K_{k2}\}_{K_c}, \left\{N_c, X_s, \{N_c, X_s\}_{K_{s^{-1}}}\right\}_{K_{k2}}, \{N-1\}_{K_{sc}} \\
&M_4 : S \leftarrow C \quad \{N-2\}_{K_{sc}}
\end{aligned}
$$

L'inserimento delle chiavi K_{k1} e K_{k2} è stato inserito solo a scopo pratico perché le informazioni da cifrare in RSA avrebbero superato la dimensione del blocco, nella successiva idealizzazione si considererà solo la chiave RSA.

2.5 Protocollo Idealizzato

M1 non è cifrato e contiene solamente il nonce del server quindi non ha rilevanza ai fini dell'analisi BAN.

$$
\begin{aligned}
&M_2 : S \leftarrow C \quad \left\{N_s, N_c, X_c, \{N_s, N_c, X_c\}_{K_{c^{-1}}}\right\}_{K_s} \\
&M_3 : S \rightarrow C \quad \left\{N_c, X_s, \{N_c, X_s\}_{K_{s^{-1}}}\right\}_{K_c}, \{N_c\}_{K_{sc}} \\
&M_4 : S \leftarrow C \quad \{N_c\}_{K_{sc}}
\end{aligned}
$$

2.6 Analisi BAN

2.6.1 Obiettivi

Gli obiettivi sono:

- **O1**: $\boxed{S \mid\equiv S \overset{k_{sc}}{\leftrightarrow} C}$ $\boxed{C \mid\equiv S \overset{k_{sc}}{\leftrightarrow} C}$ (*key-authentication*)
- **O2**: $\boxed{C \mid\equiv \#(S \overset{k_{sc}}{\leftrightarrow} C)}$ $\boxed{S \mid\equiv \#(S \overset{k_{sc}}{\leftrightarrow} C)}$ (*key-freshness*)
- **O3**: $\boxed{S \mid\equiv C \mid\equiv S \overset{k_{sc}}{\leftrightarrow} C}$ $\boxed{C \mid\equiv S \mid\equiv S \overset{k_{sc}}{\leftrightarrow} C}$ (*key-confirmation*)

2.6.2 Ipotesi

Le ipotesi iniziali sono:

$$
\begin{aligned}
H_1 &: \quad C \models \overset{Ks}{\mapsto} S \\
H_2 &: \quad S \models \overset{Kc}{\mapsto} C \\
H_3 &: \quad S \models \overset{k_s}{\mapsto} S \\
H_4 &: \quad C \models \overset{k_c}{\mapsto} C \\
H_5 &: \quad C \models \#(N_c) \\
H_6 &: \quad S \models \#(N_s) \\
H_7 &: \quad C \models \#(X_c) \\
H_8 &: \quad S \models \#(X_s) \\
H_9 &: \quad S \models C \Rightarrow X_c \\
H_{10} &: \quad C \models S \Rightarrow X_s
\end{aligned}
$$

2.6.3 Analisi

M2-1 :

$$\frac{S \models \overset{k_s}{\mapsto} S, S \triangleleft \left\{ N_s, N_c, X_c, \{N_s, N_c, X_c\}_{K_{c^{-1}}} \right\}_{k_s}}{S \triangleleft (N_s, N_c, X_c, \{N_s, N_c, X_c\}_{K_{c^{-1}}})}$$

Per H3 e M2.

M2-2 :

$$\frac{S \models \overset{k_c}{\mapsto} C, S \triangleleft \{N_s, N_c, X_c\}_{K_{c^{-1}}}}{S \models C \mid\sim (N_s, N_c, X_c)}$$

Per H2 e punto M2-1 viene verificata la *meaning rule* per il server.

M2-3 :

$$\frac{S \models \#(N_s)}{S \models \#(N_s, N_c, X_c)}$$

M2-4 :

$$\frac{S \models \#(N_s, N_c, X_c), S \models C \mid\sim (N_s, N_c, X_c)}{S \models C \models (N_s, N_c, X_c)}$$

Per H6, M2-2, M2-3 viene verificata la *nonce verification rule* per il server.

M3-1 :

$$\frac{C \models \overset{k_c}{\mapsto} C, C \triangleleft \left\{ N_c, X_s, \{N_c, X_s\}_{K_{s^{-1}}} \right\}_{k_c}}{C \triangleleft (N_c, X_s, \{N_c, X_s\}_{K_{s^{-1}}})}$$

Per H4 e M3.

M3-2 :

$$\frac{C \models \overset{k_s}{\mapsto} S, C \triangleleft \{N_c, X_s\}_{K_{s^{-1}}}}{C \models S \mid\sim (N_c, X_s)}$$

Per H1 e punto M3-1 viene verificata la *meaning rule* per il client.

M3-3 :

$$\frac{C \models \#(N_c)}{C \models \#(N_c, X_s)}$$

M3-4 :

$$\frac{C \models \#(N_c, X_s), C \models S \mid\sim (N_c, X_s)}{C \models S \models (N_c, X_s)}$$

Per H5, M3-2, M3-3 viene verificata la *nonce verification rule* per il client.

M3-5 :

$$\frac{C \models S \models (X_s), C \models S \Rightarrow X_s}{C \models X_s}$$

Per M3-4 e H10.
Il client calcola quindi $K_{sc} = (X_c \oplus X_s)$ e $C \mid\equiv S \overset{k_{sc}}{\leftrightarrow} C$

M3-6 :

$$\frac{S \models C \models (X_c), S \models C \Rightarrow X_c}{S \models X_c}$$

Per M2-4 e H9.
Il server calcola quindi $K_{sc} = (X_c \oplus X_s)$ e $S \mid\equiv S \overset{k_{sc}}{\leftrightarrow} C$

M3-7 :

$$S \mid\equiv S \overset{k_{sc}}{\leftrightarrow} C, C \mid\equiv S \overset{k_{sc}}{\leftrightarrow} C$$

Per M3-5, M3-6.
E' stato verificato l'obiettivo **O1** di *key authentication.*

M3-8 :

$$\frac{C \models \#(X_s, X_c)}{C \models \#(S \overset{k_{sc}}{\leftrightarrow} C)}$$

Per M2-3, M3-3 e perchè K_{sc} deriva da combinazione di X_c e X_s in M3-5.

M3-9 :

$$\frac{S \models \#(X_s, X_c)}{S \models \#(S \overset{k_{sc}}{\leftrightarrow} C)}$$

Per M2-3, M3-3 e perchè K_{sc} deriva da combinazione di X_c e X_s in M3-6.

M3-10 :

$$S \mid\equiv \#(S \overset{k_{sc}}{\leftrightarrow} C)$$
$$C \mid\equiv \#(S \overset{k_{sc}}{\leftrightarrow} C)$$

Per M3-9 e M3-10. E' stato verificato l'obiettivo **O2** di *key freshness*.

M3-11 :

$$\frac{C \models S \overset{k_{sc}}{\leftrightarrow} C, C \models \#(S \overset{k_{sc}}{\leftrightarrow} C), C \models S \models X_s}{C \models S \models S \overset{k_{sc}}{\leftrightarrow} C}$$

Per M3-5, M3-8, M3-4.
Perchè K_{sc} deriva da combinazione di X_c e X_s in M3-5.

M3-12 :

$$\frac{S \models S \overset{k_{sc}}{\leftrightarrow} C, S \models \#(S \overset{k_{sc}}{\leftrightarrow} C), S \models C \models X_c}{S \models C \models S \overset{k_{sc}}{\leftrightarrow} C}$$

Per M3-6, M3-9, M2-4.
Perchè K_{sc} deriva da combinazione di X_c e X_s in M3-6.

2.7 Rekeying: aggiornamento chiave di sessione

Come già detto la sessione è cifrata in **AES128/256-CBC**, per evitare che un attaccante riesca a colleizionare un eccessivo numero di pacchetti cifrati con la stessa chiave il server aggiorna la chiave con una frequenza determinata da un valore che per default è 15 minuti, con una minima frequenza di aggiornamento di 60 secondi. Un possibile miglioramento potrebbe essere effettuare *rekeying* in base al tempo e al traffico dati che sono stati inviati/ricevuti.
Il *rekeying* viene fatto sfruttando il canale cifrato già stabilito, ovvero la chiave precedente:

$$E_{k_{sc}}(\ E_{k'_{sc}}).$$

3 Implementazione

Sono stati implementati un client e un server che usano il protocollo descritto sopra per scambiarsi i messaggi cifrati con esso. Il codice è stato scritto in C++ e usa la libreria OpenSSL per cifrare, decifrare e generare numeri random. Inoltre è usata la libreria libconfuse per la gestione del file di configurazione e le le librerie GTK per la grafica del client.
Il codice è implementato in numerosi file, alcuni dei quali sono condivisi tra client e server contenuti nella cartella shared.

Nella cartella **Shared** sono presenti i sorgenti a comune tra *server* e *client*:

- **base64.cpp base64.h** Contengono le due funzioni per la conversione del testo in base64 e viceversa per evitare che caratteri speciali possano disturbare il parsing dell'XML.
- **crypto.cpp crypto.h** Contengono le funzioni per la criptazione e decriptazione in AES e RSA oltre che a funzioni d'appoggio.
- **socket-base.cpp socket-base.h** Implementazione base di un socket, con le funzioni si send e recv. Questa classe viene derivata nel client e nel server nel SocketClient e SocketServer.
- **packet.cpp packet.h opcode.h** Implementazione del pacchetto dati usato all'interno del client e del server descritto da un opcode che ne identifica la funzione che gestirà quel pacchetto. I vari opcode del protocollo si trovano all'interno di opcode.h.
- **sessionbase.cpp sessionbase.h** Implementazione della sessione dell'utente che include al suo interno il Socket che gestirà, con le dovute cifrature, la trasmissione dei dati. La SessionBase viene derivata in Session all'interno del client e del server.
- **cartella threading** Contiene le classi per una gestione semplificata dei threads.
- **cartella utility** Classi implementate per semplificare la gestione di code, singleton, logger, timer, buffer di byte ecc...

In **Client** sono presenti i file:

- **client-config.cpp client-config.h** Contiene le funzioni che permettono di gestire i file di configurazione
- **client-core.cpp client-core.h** Contiene le funzioni che permettono di collegare la gui alla parte della gestione della sessione.
- **main.cpp** Contiene il main, inizializza il client.
- **gui.cpp gui.h** Contiene il codice per la grafica GTK+3 del client.
- **cartella command** Contiene la classe per la gestione dei comandi della chat.
- **cartella networking** Contiene gli opcode dei messaggi e la classe che permette di connettersi al server utilizzando i socket.
- **cartella session** Contiene la classe che permette di gestire la sessione attiva con il server, gestire le funzioni di login e di chat.

In **Server** sono presenti i file:

- **server-config.cpp server-config.h** Contiene le funzioni che permettono di gestire i file di configurazione
- **server-core.cpp server-core.h** Contiene le funzioni che inizializzano tutti i thread di esecuzione delle funzioni del server.
- **main.cpp** Contiene il main, inizializza il server.
- **cartella channel** Contiene le classi per la gestione dei canali della chat.
- **cartella command** Contiene la classe per la gestione dei comandi della chat.
- **cartella database** Contiene la classe che permette di accedere al database.
- **cartella networking** Contiene gli opcode dei messaggi e la classe che permette di gestire i pacchetti, il socket e i thread delle connessioni sui socket.
- **cartella session** Contiene la classe che permette di gestire le sessioni attive con i client, gestire le funzioni di login e di chat.
- **cartella threading** Contiene le classi che permettono di eseguire e schedulare i messaggi come thread.