UNIVERSITÀ DI PISA

Corso di Laurea Magistrale in Cybersecurity

Relazione progetto
Bulletin Board System

Studente:
Giuseppe Conte

Febbraio 2025

Indice

Introduzione

Il progetto consiste nello sviluppo di un **Bulletin Board System (BBS)**, un sistema distribuito[1] che consente agli utenti di leggere messaggi e aggiungerne di propri. Ogni utente è identificato attraverso un nickname univoco, scelto al momento della registrazione insieme a una password.

I messaggi all'interno del BBS sono rappresentati come tuple composte dai seguenti campi: *identificativo univoco*, *titolo*, *autore* (che corrisponde al nickname dell'utente che ha aggiunto il messaggio) e *testo*. Il BBS offre diverse funzionalità agli utenti registrati:

- `List(n)`: mostra gli ultimi *n* messaggi disponibili nel sistema.
- `Get(mid)`: consente di scaricare un messaggio specifico identificato dall'ID `mid`.
- `Add(title, author, body)`: permette di aggiungere un nuovo messaggio al sistema specificandone il titolo, l'autore e il testo.

Per utilizzare queste funzionalità, gli utenti devono essere registrati e aver effettuato correttamente l'accesso. Ogni operazione è eseguita attraverso un canale sicuro, garantendo così la riservatezza delle comunicazioni. Gli utenti non possono interagire con il sistema fino a quando non effettuano nuovamente il login dopo essersi disconnessi.

Il BBS è implementato in modo centralizzato attraverso un **server BBS**, accessibile tramite una coppia ben nota di *indirizzo IP e porta*. Il server dispone di una coppia di chiavi crittografiche pubblica-privata: la componente pubblica (`pubKbbs`) è condivisa con gli utenti per garantire la sicurezza delle comunicazioni.

Gli utenti interagiscono con il sistema attraverso due fasi principali:

- **Registrazione**: l'utente si connette in modo sicuro al server BBS e fornisce un indirizzo email, un nickname e una password. Il server genera una sfida, la stampa sul terminale e attende che il client inserisca il valore corretto. Se la sfida viene superata, il server salva le informazioni e completa la registrazione.[2]
- **Login**: un utente registrato si connette al server BBS in modo sicuro e fornisce il proprio nickname e password. Il server verifica le credenziali e consente l'accesso. Una volta effettuato il login, viene stabilita una sessione sicura che resta attiva fino alla disconnessione.

[1] Il progetto realizzato è più vicino a un sistema centralizzato che a uno distribuito.

[2] Per semplicità, la sfida non viene inviata via email come richiesto dalla traccia, ma viene stampata sul terminale del server per essere inserita dal client.

Compilazione ed esecuzione su terminale

Per compilare ed eseguire il progetto, sono necessari alcuni passaggi preliminari per l'installazione delle dipendenze e per configurare correttamente l'ambiente di sviluppo. In particolare, il progetto utilizza la libreria OpenSSL, che deve essere installata sul proprio sistema.

Installazione di OpenSSL

Prima di compilare il progetto, è necessario assicurarsi che la libreria OpenSSL sia installata sul proprio sistema. È possibile installarla utilizzando il gestore di pacchetti del proprio sistema operativo. Ad esempio, su un sistema basato su Ubuntu, puoi utilizzare il comando:

```
sudo apt install openssl libssl-dev
```

Per altri sistemi operativi, consulta la documentazione ufficiale di OpenSSL per le istruzioni di installazione.

Struttura delle cartelle

Il progetto è organizzato in tre cartelle principali:

- `client_folder/`: Contiene il codice sorgente relativo al client.
- `server_folder/`: Contiene il codice sorgente relativo al server, alcune librerie specifiche per il server e i file con i dati degli utenti, dei post e la chiave privata.
- `shared_folder/`: Contiene codice e librerie condivise tra il client e il server e la chiave pubblica.

Compilazione del progetto

Per compilare il progetto, è necessario aprire il terminale e navigare nella cartella in cui si trova il `Makefile` fornito, che include una serie di comandi utili per la gestione della compilazione, la creazione degli eseguibili e la gestione dei dati. Di seguito vengono descritti i principali comandi disponibili:

- `make all`: Compila tutti i file necessari per il progetto, generando gli eseguibili `client`, `server` e `genera_rsa_keys`. Questo comando esegue tutte le operazioni necessarie per la creazione degli eseguibili.
- `make client`: Compila il file `client.c` all'interno della cartella `client_folder` e genera l'eseguibile `client`.
- `make server`: Compila il file `server.c` all'interno della cartella `server_folder` e genera l'eseguibile `server`.
- `make keys`: Compila il file `Genera_RSA_Keys.c` nella cartella `server_folder` e genera l'eseguibile `genera_rsa_keys`, utile per la generazione delle chiavi RSA.
- `make clean`: Rimuove gli eseguibili generati (`client`, `server`, `genera_rsa_keys`). Questo comando è utile per ripulire la directory dai file temporanei e dagli eseguibili.
- `make reset`: Cancella tutti i dati relativi al progetto, inclusi i file di chiavi (`public_key.pem`, `private_key.pem`) e i file di dati (`counter.txt`, `posts.txt`, `utenti.txt`). È utile per resettare completamente l'ambiente di esecuzione e rimuovere eventuali dati di test.
- `make debug`: Compila il progetto includendo i flag di debug. Questo comando attiva il blocco `#ifdef DEBUG` nel codice, abilitando la stampa di messaggi di debug. Vengono compilati sia `client` che `server`, con l'opzione di debug attivata.

Questi comandi forniscono un flusso di lavoro efficiente per compilare ed eseguire il progetto, con opzioni specifiche per la pulizia, la generazione di chiavi e l'attivazione di modalità di debug

Esecuzione del progetto

Dopo aver compilato correttamente il progetto, sarà possibile eseguire il server e il client su differenti terminali. Per avviare il server, eseguire il comando:

```
./server
```

Per avviare il client, eseguire su uno o più terminali il comando:

```
./client
```

Assicurarsi che il server sia in esecuzione prima di avviare il client, in quanto il client deve connettersi al server per poter interagire con il sistema.

Generazione delle Chiavi RSA

Per generare le chiavi RSA, è possibile eseguire il comando:

```
./genera_rsa_keys
```

Tuttavia, non è necessario eseguire manualmente questo comando. Ogni volta che si avvia il server, l'operazione di generazione delle chiavi RSA viene eseguito automaticamente in un processo separato. In questo modo, vengono generate nuove chiavi RSA ogni volta che il server viene avviato.

Errori comuni

Se durante la compilazione si verificano errori, verificare che la libreria OpenSSL sia correttamente installata e che il sistema sia configurato correttamente per il compilatore `gcc`. Eventuali errori di compilazione potrebbero anche derivare da modifiche al codice sorgente o dalla mancata presenza di alcune librerie condivise.

Panoramica del progetto

Il progetto è stato sviluppato in linguaggio C, progettato per essere eseguito su un sistema operativo Linux. La struttura del sistema segue un'architettura **client-server**, dove il **server** è il componente centrale che gestisce le richieste provenienti da più client simultaneamente.

Il server è progettato per **gestire molteplici connessioni contemporaneamente** utilizzando **thread**. Ogni volta che un client si connette al server, viene creato un nuovo **thread** dedicato per quella connessione, in modo da garantire che il server possa gestire più client simultaneamente senza bloccare altre operazioni e garantendo isolamento e fluidità nella gestione delle richieste. Questo approccio consente al server di rimanere reattivo, rispondendo in parallelo a più client senza la necessità di gestire ogni richiesta in modo sequenziale.

Un elemento fondamentale per la sicurezza del sistema è l'uso della **crittografia**. Ogni client che si connette al server genera una chiave **AES** univoca per quella sessione. Questa chiave viene quindi **criptata** con la **chiave pubblica** del server utilizzando l'algoritmo **RSA**, inviata attraverso la connessione, e infine **decriptata** dal server utilizzando la sua **chiave privata**. Una volta che la chiave AES è stata decriptata dal server, tutti gli scambi di messaggi successivi tra client e server avvengono utilizzando la crittografia **AES**, garantendo la riservatezza e la sicurezza delle informazioni scambiate.

Inoltre, il sistema include una gestione sicura delle credenziali degli utenti. Durante la **registrazione** di un nuovo utente, la **password** viene **criptata** utilizzando una funzione di **hashing** sicura. La password, una volta hashata, non viene mai memorizzata in chiaro nel sistema, ma viene salvato solo l'hash della password. Questo approccio aumenta la sicurezza e previene potenziali vulnerabilità in caso di violazione dei dati.

Il server è progettato per essere facilmente scalabile e versatile, permettendo l'aggiunta di nuove funzionalità e miglioramenti in base alle esigenze specifiche del contesto. Per garantire la sicurezza delle comunicazioni, il sistema fa uso di librerie esterne come **OpenSSL**, che fornisce implementazioni sicure per la gestione delle chiavi pubbliche e private, e **pthread**, per la gestione dei thread.

Il progetto si distingue per l'adozione di una solida architettura multithread e per le misure di sicurezza avanzate, come la gestione delle chiavi di crittografia e l'hashing sicuro delle password, offrendo così un sistema robusto e sicuro per la comunicazione tra client e server.

Formattazione dei messaggi

Nel progetto non è stata prevista un'interfaccia grafica. Pertanto, tutte le interazioni con il sistema devono essere eseguite tramite il terminale. L'utente può inviare comandi direttamente nella finestra

del terminale per eseguire le operazioni desiderate, come la registrazione, l'aggiunta di post, o la gestione della sessione.

Il protocollo di comunicazione tra client e server si basa su comandi predefiniti e messaggi strutturati e utilizza il delimitatore `::` per separare i diversi campi dei messaggi. Ogni comando viene inviato dal client al server insieme ai parametri necessari, e il server restituisce un messaggio di risposta, strutturato o semplice, a seconda dell'operazione richiesta. I messaggi scambiati tra client e server seguono un formato ben definito:

- Dal client al server: `comando::parametro1::parametro2::...`, dove ogni comando è seguito dai relativi parametri, separati dal delimitatore `::`.
- Dal server al client: il formato dei messaggi varia in base al contesto. Le risposte di errore e successo sono semplici e non contengono il delimitatore `::`, mentre le risposte più complesse, come la lista dei post, seguono un formato strutturato con campi separati.

I principali comandi supportati sono:

- `list` e `get`: per richiedere la lista degli ultimi post o un post specifico tramite il suo ID.
- `add`: per aggiungere un nuovo post.
- `register`: per la registrazione di un nuovo utente.
- `login`: per l'autenticazione di un utente.
- `logout`: per effettuare il logout.
- `close`: per chiudere il programma.

Ad esempio, un messaggio inviato per aggiungere un post potrebbe essere strutturato come segue:

`add::autore::titolo::testo`

dove `add` è il comando predefinito, mentre gli altri parametri sono in formato testuale.
Una risposta dal server potrebbe essere:

```
Il post è stato aggiunto al BBS
```

oppure

```
Errore durante il salvataggio del post
```

che verrà semplicemente stampata sul terminale del Client. I dettagli di ciascun comando, inclusa la loro sintassi e il loro funzionamento, saranno descritti nella sezione dedicata al *Client*.

Server

Funzionamento del Server

Il server del progetto si avvia tramite il comando:

```
./server
```

All'avvio, il server esegue due operazioni principali. La prima consiste nella generazione di una coppia di chiavi pubblica e privata, necessaria esclusivamente per crittografare la chiave simmetrica AES trasmessa dal client, garantendo la sicurezza nella comunicazione. Per fare ciò, il server avvia un processo separato che crea queste chiavi. Una volta che la coppia di chiavi è stata generata, il processo termina, mentre il server prosegue con le altre operazioni.

La seconda parte del server si occupa della gestione delle connessioni client e di tutte le operazioni successive. Prima di tutto, il server maschera i segnali di sistema come `SIGINT`, `SIGTERM`, `SIGQUIT` e `SIGHUP`, per evitare che questi segnali possano interrompere il funzionamento del server in momenti inopportuni. Successivamente, viene creato un thread speciale, chiamato **sigHandler**, che rimane in attesa di uno di questi segnali. Se, ad esempio, l'utente invia il segnale `SIGINT` (premendo `Ctrl+C` nel terminale del server), il server si fermerà in modo controllato, eseguendo le operazioni di chiusura.

Il server si prepara quindi a gestire le connessioni in arrivo dai client, aprendo una socket in ascolto su un indirizzo noto. In attesa di una connessione, il server rimane pronto a ricevere richieste. Ogni volta che un client si connette, viene avviato un nuovo thread dedicato alla gestione della sua richiesta. In questo modo, il server è in grado di gestire più client contemporaneamente senza bloccare l'esecuzione di altre operazioni, poiché ogni client ha il proprio thread separato. Una volta che il thread del client ha completato la sua operazione, il server torna a stare in attesa di una nuova connessione.

Nel caso in cui il server riceva un segnale dal **sigHandler**, che indica l'intenzione di fermare il server, vengono intraprese le seguenti azioni: prima di tutto, il server smette di accettare nuove connessioni. Poi, il server termina tutte le connessioni attive, fa la `join` su tutti i thread attivi (compresi i thread che gestiscono i client e quello del signal handler), chiude la socket e infine termina l'esecuzione. Questo processo garantisce che il server si spenga in modo ordinato, senza lasciare operazioni incomplete o risorse non rilasciate.

Per quanto riguarda la gestione della concorrenza, ogni client ha un proprio buffer dedicato. Questo significa che i messaggi tra diversi client non si mescolano, e ogni client può comunicare

in modo sicuro senza interferire con gli altri. Questo approccio consente di gestire la concorrenza senza conflitti, mantenendo separati i dati di ogni client.

In sintesi, il server offre una gestione robusta delle connessioni client, si ferma in modo sicuro tramite l'ascolto di segnali di sistema e gestisce efficacemente la concorrenza tramite thread separati e buffer indipendenti per ogni client.

Salvataggio dei dati

Il server si occupa della gestione e del salvataggio di tutti i dati, che vengono archiviati in file. La chiave pubblica, memorizzara nel file `public_key.pem`, viene salvata nella cartella `shared_folder`, mentre tutti gli altri dati vengono conservati nella cartella `server_folder`. Tra questi file vi sono la chiave privata, memorizzata nel file `private_key.pem`, il file `posts.txt` contenente i post, il file `utenti.txt` con le credenziali degli utenti e il file `counter.txt` che memorizza l'ultimo ID assegnato a un post.

I **post** sono memorizzati nel seguente formato:

```
id::autore::titolo_del_post::testo_del_post
```

Ogni post include un ID univoco, il nome dell'autore, il titolo e il contenuto del post. I post vengono salvati nel file in ordine cronologico, dal meno recente al più recente. Di conseguenza, per leggere i post più recenti, è necessario scorrere l'intero file.
L'**ID** è una stringa alfanumerica di 4 caratteri, composta da lettere maiuscole, minuscole e numeri. Gli ID vengono generati in modo sequenziale, a partire da `AAAA`, `AAAB`, e così via. Per velocizzare la creazione dell'ID successivo, l'ultimo ID assegnato viene memorizzato nel file `counter.txt`.
È possibile generare fino a 62^4 ID diversi.

Le credenziali degli **utenti** sono archiviate nel seguente formato:

```
email::nickname::salt::hash_password
```

Il file memorizza l'email e il nickname di ogni utente, insieme all'hash della password e al relativo salt. L'uso del salt garantisce che utenti con la stessa password abbiano hash differenti, migliorando così la sicurezza del sistema.

Client

Funzionamento del Client

Il client del progetto si avvia tramite il comando:

```
./client
```

All'avvio, il client esegue una serie di operazioni iniziali: crea la socket e si connette al server, genera la chiave AES e l'IV, quindi avvia la funzione `handle_communication`, che si occupa della gestione della comunicazione con il server.
All'interno di `handle_communication`, vengono creati due thread: uno per l'invio dei messaggi e uno per la ricezione. La funzione attende la loro terminazione con una *join*, dopodiché il controllo torna al `main`, che chiude la socket e termina definitivamente il programma.
Il thread per l'invio dei messaggi gestisce gli input che l'utente digita e monitora un timer che si resetta ogni volta che l'utente invia un messaggio al server.
L'esecuzione del client termina in uno dei seguenti casi:

- L'utente digita il comando `close`.
- Il timer scade, indicando che l'utente non ha inviato alcun messaggio entro il tempo limite.
- Il server si chiude.

Gestione dei Thread nel Client

Nel progetto, il client è progettato per operare in modo asincrono, senza attendere una risposta dal server dopo ogni comando inviato. Per ottenere questo comportamento, il client utilizza due thread indipendenti che gestiscono rispettivamente l'invio e la ricezione dei messaggi.

- **Thread di invio**: si occupa esclusivamente dell'invio dei messaggi al server. Ogni comando è inviato in modo indipendente, senza che il client si fermi ad aspettare una risposta. In questo modo, il client può continuare a inviare comandi senza interruzioni.
- **Thread di ricezione**: gestisce l'ascolto dei messaggi inviati dal server. Ogni volta che il server invia un messaggio, questo viene ricevuto e stampato a schermo dal client. Il thread di ricezione è in esecuzione continua, pronto a stampare qualsiasi messaggio che arrivi dal server.

Questa architettura multithread consente al client di essere sempre reattivo: questo consente di stampare immediatamente i messaggi inviati dal server, come quello di chiusura del server, e di

visualizzare in sequenza tutti i post ricevuti. Se invece fosse stato adottato un modello monothread, in cui ogni richiesta attende necessariamente una risposta prima di procedere, sarebbe risultato complesso gestire in modo efficace questi scenari, compromettendo la fluidità della comunicazione.

Alcune funzioni utili

Nel programma sono presenti diverse funzioni di utilità, tra cui le seguenti due, fondamentali per la gestione dell'input e della comunicazione cifrata:

`myscanf(stringa, min, max)`
La funzione `myscanf` verifica che la stringa digitata dall'utente abbia una lunghezza compresa tra `min` e `max`. Inoltre, controlla che la stringa non contenga il carattere speciale `"::"`, utilizzato per separare i campi nei messaggi inviati al server.

`send_encrypted_message(socket, chiave, iv, messaggio)`
La funzione `send_encrypted_message` si occupa automaticamente della cifratura e dell'invio di un messaggio al server. Utilizza la `chiave` e il vettore di inizializzazione `iv` per crittografare il `messaggio` con l'algoritmo AES prima di trasmetterlo attraverso il `socket`.

Thread di invio dei messaggi

La funzione `send_thread` inizia criptando la chiave AES con la chiave pubblica utilizzando l'algoritmo RSA e inviandola al server. Successivamente, invia in chiaro il valore dell'IV.

Dopo questa fase iniziale, la funzione entra in un ciclo in cui viene impostato un timer: se il client non invia comandi entro un tempo limite, il client si chiude automaticamente.

Il client attende quindi un comando dall'utente e, attraverso una serie di strutture `if-elseif`, esegue l'operazione appropriata in base al comando ricevuto. Se il comando non è riconosciuto, il client non esegue alcuna operazione.

L'uscita dal ciclo avviene quando l'utente digita il comando `close`, che provoca la chiusura del client.

I comandi

Nel codice vi è una variabile atomica `loggedin` che indica se vi è un utente loggato oppure no. I comandi si comportano in maniera diversa a seconda del valore di `loggedin`.

Comando `register`

```
register
```

Il comando `register` serve per effettuare la registrazione di un utente. Se `loggedin == 0`, allora l'utente può registrarsi. Il comando invoca la funzione `register` per compilare il messaggio, durante la quale verrà richiesto all'utente di inserire un'email, un nome utente e una password. Tuttavia, per semplicità, non viene eseguito alcun controllo sulla validità dei valori inseriti (ad esempio, non si verifica se il campo email contenga effettivamente un indirizzo valido).
I tre valori vengono concatenati utilizzando il carattere `"::"` come separatore. Successivamente, viene aggiunto il prefisso `"register::"` all'inizio del messaggio prima di inviarlo al server.
Di conseguenza, il messaggio inviato ha il seguente formato:

`register::email::nickname::password`

Infine, all'utente viene chiesto di inserire un valore numerico relativo alla sfida, che viene successivamente inviato al server.

Comando `login`

```
login
```

Se `loggedin == 0`, il comando `login` invoca la funzione `login` per compilare il messaggio di autenticazione. Durante l'esecuzione del comando, verrà richiesto all'utente di inserire il proprio nickname e la password.
Il messaggio finale verrà formattato nel seguente modo:

`login::nickname::password`

Comando `logout`

```
logout
```

Se `loggedin == 1`, il comando `logout` invia al server il messaggio corrispondente per effettuare il logout dell'utente, e il valore di `loggedin` viene impostato a 0.

Comando `add`

```
add
```

Il comando `add` permette di scrivere un post. Se `loggedin == 1`, viene invocata la funzione `write_post` per formalizzare il messaggio. Durante questa fase, verranno richiesti all'utente il titolo e il testo del post.
Tuttavia, come per ogni messaggio scritto dall'utente, esiste un limite massimo di caratteri oltre il quale il post non è considerato valido. Dopo aver inserito il contenuto, all'utente viene chiesto: `Sei sicuro di voler aggiungere questo post?` Se l'utente digita `"Y"`, il messaggio viene formattato nel seguente modo:

```
add::autore::titolo::testo
```

e successivamente inviato al server.

Comando `list`

```
list
```

Il comando `list` serve per richiedere gli ultimi post pubblicati nel BBS. Se `loggedin == 1`, viene invocata la funzione `flist`, che chiede quanti post caricare. Il messaggio inviato al server sarà formattato nel seguente modo:

```
list::num
```

Comando `get`

```
get
```

Il comando `get` permette di richiedere un post specifico. Se `loggedin == 1`, viene invocata la funzione `fget`, che chiede all'utente di inserire l'ID del post desiderato. Il messaggio inviato al server sarà formattato come segue:

```
get::id
```

Comando `close`

```
close
```

Il comando `close` serve per chiudere il client. Non invia alcun messaggio al server, ma imposta a `1` la variabile di controllo del ciclo, causando così l'uscita dal ciclo principale del programma.

Comando `hack`

```
hack
```

Se il programma è stato compilato in modalità **debug**, è possibile digitare questo comando ”segreto” per facilitare il debugging.
Nello specifico, la scrittura della riga successiva bypassa il controllo dei caratteri effettuato dalla funzione `myscanf`. Il messaggio viene quindi cifrato e inviato al server esattamente come è stato scritto, senza alcuna verifica.
Questo permette di testare il comportamento del server quando riceve *messaggi artefatti*, simulando possibili tentativi di attacco o errori nella gestione dell'input.

Thread di ricezione dei messaggi

Il thread di ricezione dei messaggi rimane in esecuzione in un ciclo fino a quando il client non termina, come descritto precedentemente. Questo thread è sempre in attesa di messaggi provenienti dal server.
Quando riceve un messaggio, il thread esegue le seguenti operazioni:

- Prima di qualsiasi altra operazione, il thread decifra il messaggio ricevuto.
- Se il messaggio ricevuto è `"LOGIN_OK"`, imposta il valore della variabile `loggedin` a 1, indicando che l'utente è correttamente loggato.
- Se il messaggio contiene un post, e perciò inizia con `"LIST::"` o `"GET::"`, il thread esegue un'operazione di `split` per ottenere i seguenti campi: `id`, `autore`, `titolo` e `testo`. Successivamente, stampa ogni campo su una riga separata.
- Altrimenti il thread stampa il messaggio decifrato così com'è, senza alcuna modifica.

Sicurezza del sistema

Protezione delle credenziali degli utenti

Per garantire che le password degli utenti non vengano mai archiviate o trasmesse in chiaro:

- Sul lato server, le password non vengono salvate direttamente, ma solo il loro **hash** accompagnato da un **salt** casuale.
- Questo garantisce che due utenti con la stessa password abbiano hash diversi, rendendo inefficaci attacchi basati su tabelle precomputate (*rainbow tables*).

Sicurezza della comunicazione

Tutte le comunicazioni tra client e server sono cifrate per garantire **confidenzialità, integrità e protezione da attacchi di replay**:

- Quando il client si connette al server, genera una chiave **AES** e un **IV** per la crittografia simmetrica.
- La chiave AES viene cifrata con la chiave pubblica **RSA** del server e inviata al server, garantendo così uno **scambio sicuro della chiave**.
- Dopo questo primo scambio, tutte le comunicazioni successive sono cifrate con AES, compresa l'invio della password dell'utente.
- Ogni connessione tra client e server utilizza chiavi AES differenti, rendendo **impossibili attacchi di replay**, poiché un messaggio riutilizzato non può essere decriptato in una sessione diversa.

Non-malleability e protezione da input malevoli

Il server è stato progettato per ignorare messaggi malformati o manipolati intenzionalmente:

- Se un messaggio ricevuto non è ben formato o mancano dei parametri richiesti, il server **non esegue alcuna operazione**.
- Ad esempio, se un client tenta di inviare un comando `add` senza specificare titolo e testo, il server rifiuterà l'operazione.
- Se l'autore specificato non coincide con il nickname dell'utente loggato, il server non accetterà il post.

- Durante la fase di registrazione, il server verifica che il codice di sfida inviato dal client abbia il formato corretto (sei cifre numeriche).
- Il client stesso esegue controlli sugli input prima di inviare i messaggi, quindi se il server riceve input errati, si tratta con molta probabilità di un tentativo di attacco.

Per facilitare il debug e testare il comportamento del server con input anomali, è stato aggiunto il comando `hack` (attivo solo in modalità debug). Questo comando consente di inviare messaggi arbitrari al server, bypassando i controlli del client.

Gestione delle chiavi RSA e sicurezza a lungo termine

Ad ogni avvio del server, viene generata una nuova coppia di chiavi **RSA (pubblica-privata)**, con la precedente coppia che viene eliminata. Questo migliora la sicurezza per diversi motivi:

- Se un attaccante compromettesse la chiave privata, potrebbe decriptare solo le comunicazioni **della sessione corrente**, ma non quelle future.
- Anche se un attaccante registrasse il traffico crittografato per lunghi periodi, **non potrebbe in futuro decriptare sessioni passate**, perché ogni avvio del server utilizza una chiave diversa.
- Questo riduce l'efficacia di attacchi persistenti e limita la finestra temporale in cui una chiave rubata è utile.

Sebbene questa strategia **non garantisca Perfect Forward Secrecy (PFS) nel senso più stretto**, è comunque una misura di sicurezza importante. Infatti, PFS richiederebbe che nemmeno durante la sessione corrente un attaccante con accesso alla chiave privata possa decriptare i messaggi precedenti. Nel nostro caso, se un attaccante riuscisse a sottrarre la chiave **mentre il server è in esecuzione**, potrebbe comunque leggere il traffico della sessione attuale.

Test e Debugging

Non è stata implementata una suite di test automatizzata, ma il comportamento del sistema è stato verificato manualmente:

- Il server viene avviato e successivamente si avvia il client.
- Si eseguono operazioni di registrazione, login, aggiunta di post e recupero di post per verificare il comportamento atteso.
- È possibile compilare il programma in modalità debug con il comando `make debug`, il quale abilita parti di codice condizionate da `#ifdef DEBUG`. In questa modalità, vengono stampati a schermo i valori di alcune variabili per monitorarne il contenuto.
- Inoltre, in modalità debug, è possibile utilizzare il comando segreto `hack`, che disabilita i controlli sugli input digitati dall'utente, consentendo l'invio di messaggi arbitrari al server. Questo permette di testare il comportamento del server in presenza di input anomali o malformati.

Conclusioni

Il progetto implementa un sistema di comunicazione sicuro tra client e server, adottando tecniche avanzate di crittografia per garantire la riservatezza, l'integrità e l'autenticità dei dati trasmessi. L'uso combinato di RSA per lo scambio sicuro della chiave AES e di AES per la cifratura simmetrica assicura che ogni sessione sia isolata e protetta da attacchi di intercettazione o replay.

Dal punto di vista della sicurezza, il sistema protegge le credenziali degli utenti attraverso l'hashing delle password con *salt* casuali, prevenendo così attacchi basati su tabelle precomputate. Inoltre, la gestione delle chiavi RSA a rotazione per ogni avvio del server riduce la possibilità di compromissioni a lungo termine, rendendo ogni sessione indipendente dalle precedenti.

L'architettura multithread del client garantisce un'esperienza fluida e reattiva, permettendo di ricevere e inviare messaggi in modo asincrono senza blocchi operativi. Allo stesso tempo, i meccanismi di validazione degli input e il sistema di gestione degli errori lato server impediscono il verificarsi di exploit basati su input malevoli o tentativi di attacco per manipolazione dei dati.

L'integrazione di tecniche crittografiche moderne e di buone pratiche di gestione delle credenziali conferisce al sistema un elevato livello di sicurezza, riducendo i rischi derivanti da attacchi informatici e garantendo la protezione dei dati trasmessi.

Note e Possibili Miglioramenti

Perfect Forward Secrecy (PFS)

Attualmente, lo scambio della chiave simmetrica avviene cifrando la chiave AES con la chiave pubblica RSA del server. Questo sistema garantisce la sicurezza della trasmissione, ma non offre Perfect Forward Secrecy in senso stretto. Se un attaccante riuscisse a ottenere la chiave privata del server, potrebbe decifrare tutte le comunicazioni passate intercettate. Un miglioramento possibile sarebbe l'uso di Diffie-Hellman per lo scambio della chiave simmetrica, il che garantirebbe che anche conoscendo la chiave privata del server, un attaccante non potrebbe recuperare le chiavi di sessioni passate.

Nota sulla gestione della sfida durante la registrazione

Durante il processo di registrazione, come descritto inizialmente nella sezione *Introduzione*, l'utente si connette in modo sicuro al server BBS e fornisce un indirizzo email, un nickname e una password. Il server genera una sfida, la stampa sul terminale e attende che il client inserisca il valore corretto. Una volta che la sfida è superata, il server salva le informazioni e completa la registrazione.

Va notato che, sebbene la traccia richiedesse che la sfida venisse inviata via email, per motivi di semplicità, nel mio progetto la sfida è soltanto stampata sul terminale del server.

Gestione della memoria

Nel sistema di scambio dei messaggi tra client e server, il buffer ha una dimensione massima definita dalla costante `BUFFER_SIZE`, e quindi anche la dimensione massima di ogni singolo messaggio è limitata a `BUFFER_SIZE`. Ogni parametro all'interno del messaggio ha una dimensione massima predefinita, e la somma di tutte le dimensioni dei parametri non supera mai `BUFFER_SIZE`, evitando così il rischio di overflow. Tuttavia, ciò può comportare una piccola quantità di memoria inutilizzata, dovuta alla dimensione predefinita dei parametri, che può risultare in un lieve spreco di memoria.

Gestione dell'accesso ai file

L'accesso ai file `test.txt`, `post.txt` e `counter.txt` è protetto da lock, impedendo che più thread possano accedere simultaneamente in modo non controllato. Questo garantisce la consistenza dei dati ed evita corruzione o accessi concorrenti indesiderati.

Sicurezza nei thread del client

Nel client, la variabile di controllo che regola l'esecuzione del ciclo principale è dichiarata atomica. Questo garantisce che, quando il client deve terminare, la modifica della variabile venga vista correttamente dall'altro thread, evitando condizioni di competizione.

Anche la variabile che memorizza il nickname dell'utente collegato è atomica. Questo assicura che la modifica del nickname, da parte di un thread, venga correttamente sincronizzata e visibile all'altro thread, evitando conflitti nel suo aggiornamento durante l'esecuzione del programma.

Limitazione degli utenti simultanei e prevenzione del sovraccarico

Il server è progettato per gestire un numero limitato di connessioni simultanee. Oltre questa soglia, il server rifiuta nuove connessioni. Inoltre, il client si chiude automaticamente dopo un certo tempo di inattività, liberando risorse in caso un utente lasci aperta la connessione senza interagire. Tuttavia, nonostante siano state implementate delle misure di sicurezza, il sistema non è sufficiente per prevenire attacchi di tipo Flooding, in cui un numero elevato di richieste simultanee viene inviato al server, saturando le sue risorse.

Gestione della Concorrenza e Uso di `sleep()`

L'uso di `sleep()` è stato impiegato in alcuni punti critici del codice per evitare errori legati alla concorrenza tra i thread. Nonostante l'implementazione di meccanismi di sincronizzazione nel server, in alcune situazioni è stato necessario introdurre un ritardo esplicito per garantire il corretto funzionamento del sistema.

Un esempio di tale problema si è verificato durante l'esecuzione del comando `List`, quando il client richiede al server diversi post in sequenza. Il server invia i post uno alla volta, ma a causa della latenza tra i messaggi inviati tramite socket, talvolta i post si sovrappongono tra loro. Questo significa che i dati relativi a un post possono essere parzialmente o completamente mescolati con quelli di un altro, creando confusione nel client che riceve i dati.

Per mitigare questa condizione di race, è stato introdotto un ritardo tramite `sleep()` tra le operazioni. Sebbene questa soluzione abbia ridotto il problema, la sovrapposizione dei messaggi si verifica ancora occasionalmente, suggerendo che una sincronizzazione più robusta tra il client e il server potrebbe essere necessaria.

Una possibile soluzione a questo problema potrebbe essere l'adozione di un sistema di conferma dei messaggi, simile a un meccanismo SYN-ACK, in cui il client riceve conferma della corretta ricezione di un post prima di passare a quello successivo. In questo modo, si eviterebbe che i post inviati dal server si sovrappongano tra loro, migliorando la sincronizzazione tra i due.