Tema 2, Rețele de Calculatoare
DemiSecSH

Aldea Andrei

Decembrie 2021

1 Introducere

Protocolul **SSH** poate fi folosit pentru a oferi servicii sigure în rețea, chiar dacă rețeaua este compromisă. Protocolul descris în acest document, de aici denumit **DemiSecSH**, oferă posibilitatea de transmitere în siguranța a datelor prin rețea. Acesta este o simplificare a protocolului **SSH**, omițând pași precum procedura de schimb de chei *Diffie Hellman* și negocierea protocolului ultilizat[4].

2 Tehnologii utilizate

O scurta lista a tehnologiilor utilizate, și motivele pentru care au fost alese:

a) **TCP**
Ne asigura transmiterea fără pierdere a datelor prin rețea. Avem nevoie de aceasta asigurare, deoarece cheile pentru algoritmii RSA și AES au un număr relativ mare de biți pe care nu ne pute permite sa-i pierdem, sau sa ii primim într-o ordine greșita.

b) **AES**
Folosim algoritmul AES pentru criptarea mesajelor ce vor fi transmise prin rețea intre clienți și procesele worker din cadrul serverului. Acesta asigura o criptare și decriptare relativ rapida a mesajelor.

c) **RSA**
Folosim algoritmul RSA pentru a evita transmiterea în text-simplu a cheilor folosite de algoritmul AES. Acesta este folosit doar în prima comunicație, cheile publice fiind hard-codate atât în clienți cat și server.

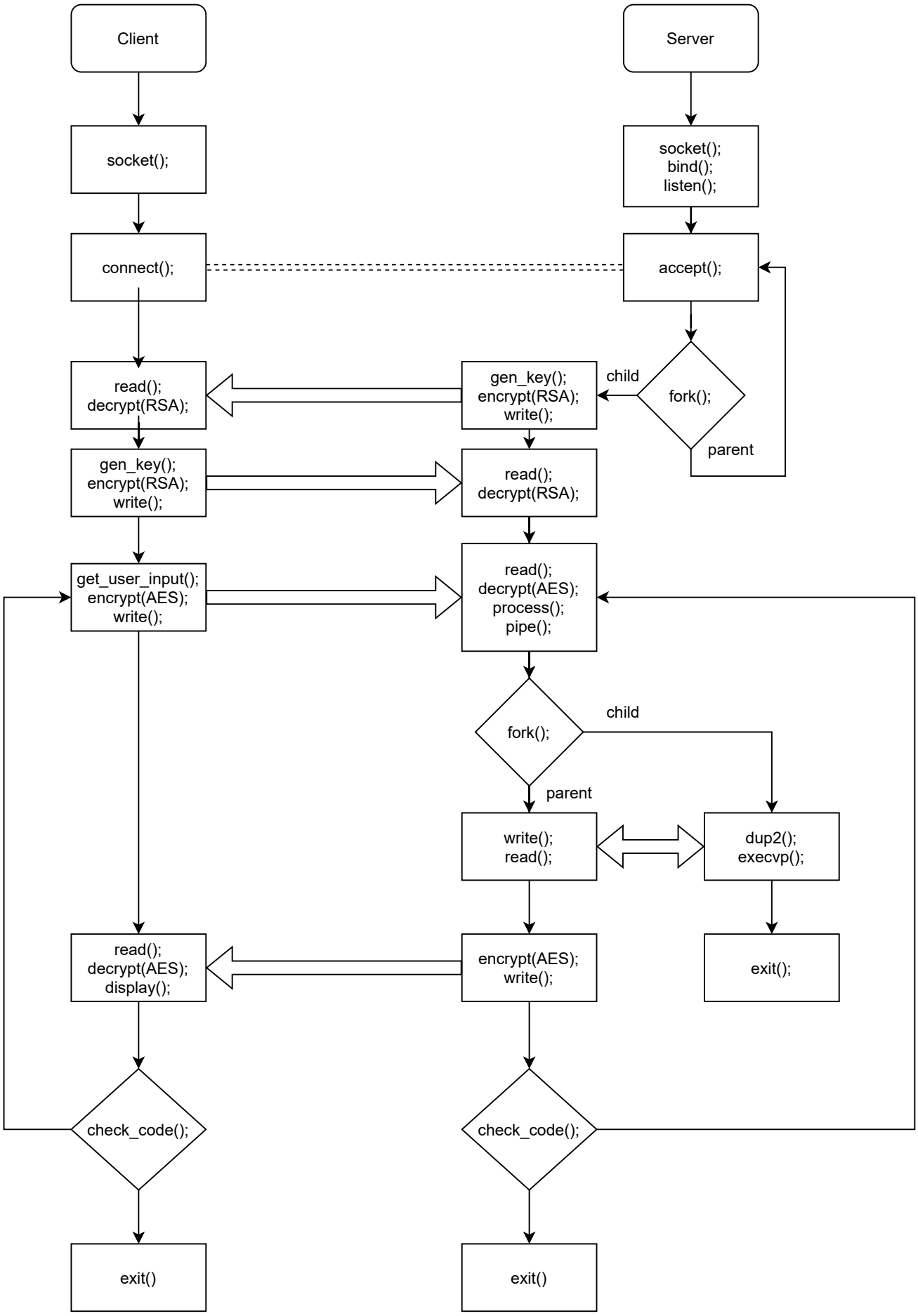
d) **Libgcrypy**[1]
Aceasta librărie open-source, ne furnizează funcțiile necesare pentru generarea de chei și criptarea / decriptarea mesajelor atât pentru algoritmul RSA cat și pentru AES.

e) **FieldKit**
Este o librărie scrisa de mine, în cadrul acestui proiect, ce furnizează o mulțime de funcționalități ce includ: parsarea argumentelor de la linia de comanda, management-ul erorilor, un canal de comunicații securizat.

f) **Xmake**[2]
Este build-system-ul folosit în acest proiect. Oferă o interfața, în Lua, ușor de folosit pentru creerea librăriilor, integrarea librăriilor într-un executabil și management-ul mai multor executabile într-un proiect.

3 Arhitectura aplicației

DemiSecSH este structurat sub forma unei arhitecturi client-server, unde clienții comunica cu procese worker, copii ai serverului. Comunicarea este făcuta prin rețea, folosind TCP, toate mesajele fiind criptate. O diagrama simplificata a programului:


Client
Server
socket();
socket();
bind();
listen();
connect();
accept();
read();
decrypt(RSA);
gen_key();
encrypt(RSA);
write();
child
fork();
parent
gen_key();
encrypt(RSA);
write();
read();
decrypt(RSA);
get_user_input();
encrypt(AES);
write();
read();
decrypt(AES);
process();
pipe();
fork();
child
parent
write();
read();
dup2();
execvp();
read();
decrypt(AES);
display();
encrypt(AES);
write();
exit();
check_code();
check_code();
exit()
exit()

4 Detalii de implementare

Atât clientul cat și serverul folosesc abstracții simple peste primitivele UNIX/linux. De exemplu: serverul folosește un tcp_listener pentru a accepta confecțiuni de la clienți:

```
typedef struct fk_tcp_listener_t {
    int sock;
    struct sockaddr_in addr;
} fk_tcp_listener_t;
```

Acesta este inițializat cu tcp_listener_new() și folosit de către tcp_accept():

```
int fk_tcp_listener_new(fk_tcp_listener_t* listener, int port) {
    if ((listener->sock = socket (AF_INET, SOCK_STREAM, 0)) == -1) {
      fk_errorln("Could not create listener socket: %s", strerror(errno));
      return -1;
    }

    bzero (&listener->addr, sizeof (listener->addr));
    listener->addr.sin_family = AF_INET;
    listener->addr.sin_addr.s_addr = htonl(INADDR_ANY);
    listener->addr.sin_port = htons(port);

    if (bind (listener->sock, (struct sockaddr *) &listener->addr,
        sizeof (struct sockaddr)) == -1) {
      fk_errorln("Could not bind addr to socket in tcp_listner: %s", strerror(errno));
      return -1;
    }

    if (listen (listener->sock, 5) == -1) {
      fk_errorln("Could not start listnening: %s", strerror(errno));
      return -1;
    }

    return 0;
}

int fk_tcp_accept(fk_tcp_listener_t* listener, fk_tcp_connection_t* con) {
    bzero(&con->addr, sizeof(con->addr));
    int len = sizeof(con->addr);
    con->sock = accept (listener->sock, (struct sockaddr *) &con->addr, &len);

    if(con->sock < 0 ) {
        fk_errorln("Could not accept: %s", strerror(errno));
        return -1;
    }
    return 0;
}
```

Exista astfel de abstracții pentru tcp_connect(), fork(), read(), write() etc.. Acestea oferă interfețe mai simple și ușor de folosit (specializate pentru situația noastră). Pentru transferul datelor de la server la client și vice-versa folosim următorul protocol simplu:

1) Imediat după stabilirea conecțiunii, atât clientul cat și serverul vor transmite lungimea mesajului criptat în text-simplu apoi un mesaj criptat cu cifrul RSA (folosind cheia publica a interlocutorului) ce conține o cheie proaspăt-generată pentru a fi utilizata în cifrul AES.

2) Apoi atât clientul cat și serverul vor citi mesajul criptat și îl vor decripta(folosind cheia privata proprie). Acum ca ambii interlocutori au cate doua chei pentru cifrul AES, Poate începe comunicarea propriu-zisa

3) Clientul trimite lungimea mesajului criptat în text-simplu apoi un mesaj criptat cu cifrul AES ce conține textul pentru o comanda.

4) Serverul citește, decriptează și procesează mesajul de la client.

5) Serverul trimite lungimea mesajului criptat în text-simplu apoi un mesaj criptat cu cifrul AES ce conține rezultatul rulării comenzii primite și un cod de status

6) Clientul citește, decriptează și procesează mesajul de la server.

7) Dacă clientul dorește sa continue comunicarea se reia de la pasul 3).

Este important de menționat ca cheile pentru cifrul RSA, atât cea publica a interlocutorului cat și cea privata proprie, sunt hard-codate atât în client cat și în server. Nu transmitem cheile publice în text-simplu pentru a evita posibilitatea unui atac MITM[3]. Aceste chei au fost generate o singura data si sunt salvate sub forma unor S-expresii:

```
const char* CLIENT_PUBLIC_KEY = "(public-key
(rsa
(n #00C534FB9FF6E7728A1BB068F8B8B6AC7D7351225DF149076E4B13A23765C7B73B3DA0AA3880D8A2978FF3050
BB91DABF5B1CF536D31F9D00A08F800CD85856AFFB388F04883DD2536EC0140572D6D41C2A8626094620559BA10BE
0B964B1C887817255FD46C86F7BE860F11CD981D218F8140E864D2AB4C739B17ABAC2D9E89EEBAD35294ED9EC7374
274D3AD38D77E9BD21B013CF921BB625633171C8B2FDE258A4468252AF2DF099FE667309104029705D929491AE1A2
C2CBA440B70ED549B2991F58B61E45733C59091748FC719725C1D2593986C06E5B489A440FAEC683BC7C77B165B60
0E1B073C6C43B34B5D541A7825CD79C2861D79CD5E047BAE4415F#)
(e #010001#)
))";
const char* SERVER_PUBLIC_KEY = "(public-key
(rsa
(n #00E16F15FF8FD8D493D410D447EABF68AFCD316D7D2B66189C29A63C661152C5A089018C635AE4C4B5DEDA592
9740E6AAA495719DC6C0966268C8303951B6D3EBF460FD475EAE3F05D5AA1BA1B66FF4E9ED8A3D909F600864D111A
878C45A72C637C053AC241DC68E3E97C363ECF4008418728B2787BC8FFCCF30CA498FCF7DBE07C16AF4E84CA79B0C
89C950E2B0E722D4A7115C5823922BFC382544F249F5E411E792090257F712A94B4759EBF53E8557BDF0FBC3BAC4A
DE9F23EF39D341534A5AC46EBF5B894CC31196C9CA6F0852D71CA7499E2C9E2B304CA79F9B30C35518C6AEC1B811E
3E13D56143B352F1032682C5EE5EE4CDB1E9A308A54255F029A85#)
(e #010001#)
))";
```

Cheile pentru cifrul RSA sunt regenerate la fiecare comunicare, folosind generatorul oferit de Libgcrypt:

```
gcry_randomize(key, 16, GCRY_STRONG_RANDOM);
```

Mesajele transmise si raspunsurile primite sunt stocate sub forma unor structuri de date ce au propriile metode de read/write.

```
typedef struct fk_message_t {
    int code;
    char metadata[fk_message_metatada_len];
    int dlen;
    char* data;
}fk_message_t;

int fk_tcp_plain_message_read(fk_tcp_connection_t con, fk_message_t* msg);
int fk_tcp_plain_message_write(fk_tcp_connection_t con, fk_message_t msg);
```

Dar acestea trimit datele in text-simplu asa cum sunt stocate. De aceea exista functii pentru a genera mesaje cu datele criptate.

```
int fk_crypto_aes_message_read(fk_crypto_tunnel_t tun,fk_message_t* msg);
int fk_crypto_aes_message_write(fk_crypto_tunnel_t tun,fk_message_t msg);
int fk_crypto_rsa_message_read(fk_crypto_tunnel_t tun,fk_message_t* msg);
int fk_crypto_rsa_message_write(fk_crypto_tunnel_t tun,fk_message_t msg);
```

5 Concluzii

Protocolul folosit poate suferi câteva îmbunătățiri:

1) Putem folosi procedura de schimb de chei Diffie Hellman, pentru a rămâne cu o singura cheie pentru cifrul AES și a simplifica implementarea.

2) Putem asigna cheii pentru cifrul AES o data de expirare, și realiza un nou schimb de chei când aceasta nu mai este utilizabilă.

3) Utilizatorul (clientul) ar trebuii autentificat pe mașina care rulează serverul.

Arhitectura serverului ar beneficia de utilizarea unui worker-pool, pentru procesele worker pentru a elimina overhead-ul impus de creerea unui nou proces de fiecare data.

References

[1] Koch W., Schulte M. : The Libgcrypt Reference Manual, Version 1.9.0 (18 Ianuarie 2021)

[2] Pagina principala xmake, `https://xmake.io`. Ultima data accesat pe 9 Decembrie 2021

[3] Pagina Wikipedia pentru atacul MITM, `https://en.wikipedia.org/wiki/Man-in-the-middle_attack`. Ultima data accesat pe 9 Decembrie 2021

[4] RFC-ul principal pentru protocolul SSH, `https://datatracker.ietf.org/doc/html/rfc4253`. Ultima data accesat pe 10 Decembrie 2021