Minimalistic SSH implementation

Rozimovschii Denis, 2B2

Retele de calculatoare
Universitatea "Alexandru Ioan Cuza"
Romania, Iasi

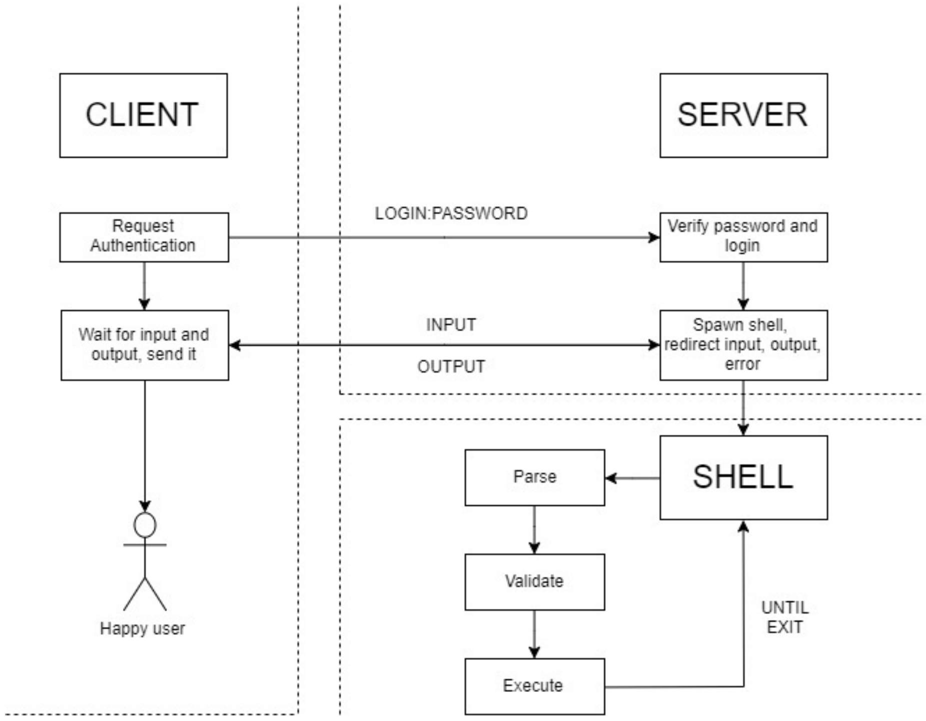
Abstract. This paper is supposed to provide an extensive documentation on the protocol, internal algorithms and structures used in the project *mySSH*.

Table of Contents

1 Introduction

mySSH is a minimalistic cryptographic network protocol for **remote** control of a system over an unsecured network. A *mySSH* server will listen on a machine waiting for a number of clients to connect over a netwrok and securely trasmit commands. The commands (also called *command lines*) will have a specific format and the server will ensure the correct execution and forwading of the output to the client.

2 General Structure



3 Outline

The project is divided in three main parts: **Shell**, **Client**, **Server**

- **Shell**
  A standalone application that reads from stdin and outputs to stdout. On receiving input, it will tokenize it, create an AST using Shunting Yard algorithm and then pass it to a recursive function *Evaluate_Tree* that will evaluate it.
  On recursive evaluation, depending on the node type, it will execute its children, open files, creates pipes.
  It will terminate upon receiving **exit**
- **Client**
  It will read from standard input and send everything to the server, as well as receive all the output from the server and redirect it to standard output
- **Server**
  It will continuously read what it has received from the summoned shell and redirect it to a secured channel with the client. Also, it will read secured input from the client and pass it to the shell.

4 Protocol

1. Client and Server establish a secured connection
2. Client asks the user for password and sends it to the server in a the form of **Login_Payload**.
3. Server verifies the password and authenticates the user into the system. Sends a "OK" response.
4. Client will continue only if it receives a "OK" response.
5. Server spawns a shell with redirected input, error and output, starts continuosly redirecting client input/output/error *to* the shell, as well as *from* the shell.
6. Client continuously reads and writes to standard sockets what it has received from the server
7. Upon receiving the "exit" command, shell with return and the server will send **PACKET_TERMINATE** signaling that the communication should be ended.
8. Server and Client will clear their contexts, connections and close the sockets.

5 Command Line Format

The command line can contain the following:

(a) Any Linux program with any number of arguments.
(b) Commands "cd", "exit"
(c) Any number of commands from (a) chained with:

```
"|" "&&" "||" ";".
```

(d) Any chaining of commands with redirection operators:

```
">", ">>", "2>>"
```

5.1 Examples of Correct Command Lines

```
[remote input]   echo "hello world"
[remote output]  hello world


[remote input]   echo "I am a file" > file.txt; cat file.txt
[remote output]  I am a simple file


[remote input]   touch file.txt; echo "file" > file.txt;
                 rm file.txt; cat file.txt
[remote output]  cat: file.txt: No such file or directory
```

```
[remote input]   echo "int main() { return 1; }" > file.c;
                 gcc file.c; ./a.out || echo "I dieded"
[remote output]  I dieded
```

```
[remote input]   echo "int main() { return 0; }" > file.c;
                 gcc file.c; ./a.out && echo "I dieded"
[remote output]  I dieded
```

5.2 Examples of Incorrect Command Lines

```
[remote input]   echo "hello world" &
[remote output]  [Parsing error]
```

```
[remote input]   echo "I am a file" && && cat file.txt
[remote output]  [Parsing error]
```

6 Packet format

The packet formed by the client and the server before encryption has the following format:

```
struct SSH_Packet
{
    1. Packet_Type          packet_type
    2. unsigned int         payload_length
    3. unsigned char        sha256_verification[SHA256_DIGEST_LENGTH]
    struct SSH_Payload
    {
        4. unsigned int             content_length
        5. unsigned int             padding_length
        6. unsigned char*           content
        7. unsigned char*           padding
    } payload
}
```

1. Used for recognition and sending messages. Each packet has a type.
2. Length of the **payload** member.
3. A *SHA256* hash of the content, for verification of the integrity of **payload.content** buffer after recieving it
4. Length of the content buffer
5. Length of the padding buffer
6. The actual data that the packet transports
7. Security padding with random length and random bytes

7 Data structures and enums

```
enum Packet_Type
{
    PACKET_QUERY            Complex command from the Client. Server must parse and execu
    PACKET_RESPONSE         A repsponse either from the Client or from the Server
    PACKET_REQUEST          A request from the Client or from the Server
    PACKET_AUTH_REQUEST     Authentication request from the Server. Client must provide 
    PACKET_AUTH_RESPONSE    Authentication response from the Client. Server must check c
    PACKET_READY            Notification from the Server that it is ready for another co
    PACKET_ERROR            Notification about an error from the server
    PACKET_TERMINATE        Signal from server that it wants to terminate the connection
    PACKET_SIGNAL           Signal from client to executed application
}
```

```
struct Login_Payload
{
    unsigned int login_length
    unsigned int password_length
    char login[MAX_LOGIN_LENGTH + 1]
    char password[MAX_PASSWORD_LENGTH + 1]
}
```

8 Authentication

For authentication, linux user and password will be used. **seteuid** will be used to set the current shell user.

9 Technologies Used

The application will use:

(*) Network socket programming using TCP/IP sockets.
Because it is simple, stable and fast. Also reliable for transmitting encrypted data over a network.

(*) An encryption suite for Transport Layer Protocol.
OpenSSL with RSA-1024 encryption. Because it is reliable, flexible and it is used by default in multiple Linux distros.

10 Conclusion

It is a hard task to create a secured channel for remote execution of commands, especially using a hand-made shell. A log of error checking and work is required to acomplish it.
A stable protocol need to be constructed, that will not allow errors and will also be coherent.
The shell should aware of different sort of errors and inform the user about them. That is why I mostly used exceptions for error-checking across the project.
A better solution would securize the shell execution as well as implement non-blocking I/O from server and client. It would generate a private key for every client to enchance security.

References

1. `https://en.wikipedia.org/wiki/Secure_Shell`
2. `http://www.allsyllabus.com/aj/note/Computer_Science/Computer%20Networks%20-%20II/Unit5/Secure%20Shell%20[SSH].php`
3. `https://aticleworld.com/ssl-server-client-using-openssl-in-c/`

4. `https://gist.github.com/MartinMReed/6395285`
5. `https://gist.github.com/MartinMReed/6393150#file-openssl_server-c-L71`
6. `http://h41379.www4.hpe.com/doc/83final/ba554_90007/ch04s03.htmls`
7. `https://www.ibm.com/developerworks/library/l-openssl/index.html`
8. `http://man7.org/tlpi/code/online/diff/users_groups/check_password.c.html`