Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

# Bachelor’s thesis
# Information Science

# Design and Implementation of an Alternative to SSH

| | |
|---|---|
| **Authors** | Raphael Emberger |
| **Date** | June 3, 2019 |

# Abstract

As Secure Shell (SSH) is an old application that has loads of features. Not all of those features are used in everyday business. In this thesis a simpler protocol prototype that can do the same as SSH was designed and implemented. The developed solution can replace SSH in its core feature: Connecting to a shell on a remote system.

This project developed an application called “oh-my-gosh” that is capable of the same core functionality that SSH provides. This includes a client application called “gosh” and a server counterpart with the name “goshd”. The latter was designed to be run as a background process on a Unix system. This solution uses a secure connection as channel to provide more safety in the entire process. A user can authenticate itself on the remote system either via a password or using public key cryptography. All the usual work flows are possible on the shell such as:

- Navigating through the file system
- Running scripts and applications
- Using applications that use *ncurses(3X)* (2019)

The solution relies on a number of Unix specific technologies like pseudoterminals (ptys) and Pluggable Authentication Module (PAM), which is used in *login(1)* (2012) as well, to realize its use cases.

# Preface

The SSH protocol (Moorer 1971, Bider & Baushke 2012, Baushke 2017, Bider 2018*a*,*b*) is a system that allows a user to log in on a remote machine and perform tasks on that remote machine via a Command Line Interface (CLI). SSH is widely known and used in everyday tasks. However: It is now over twelve years old in its current form. One of the problems with SSH is its complexity, both in the initial phase when key material is exchanged, but also later, for example because the server must always decide whether to return a character that has been sent to it or not (echo).

The goal of this work is a radically simplified protocol, which in its functions is similar to SSH (N.B. the similarity concerns the functions, not necessarily the protocol details). I develop the protocol, as well as a client and a server - all in the Go/Golang programming language (Go). I demonstrate that the software can replace SSH by showing that it can handle several common use cases, among them:

- Interactive session
- `Rsync` with my solution as transport protocol

This bachelor thesis was proposed by Dr. Stephan Neuhaus (Neuhaus 2018) and aroused my interest as it is a challenge in the domain of information security and will produce a palpable result.

On this note I would like to thank Dr. Stephan Neuhaus for helping me along the way of this Bachelors thesis.

As an additional remark, I would like to point out that the team for this Bachelors thesis originally consisted of two people, but after some months of uncertainty, Mr Schwarz decided to opt out of the project as he said he underestimated the workload from the modules he booked on top of the Bachelors thesis. This happened halfway through the project and had great impact on the project itself: The expected work from him had to be done by me, Raphael Emberger. This also meant less time for designing and implementing all the features described in the tasks (see 1.4). Fortunately Dr. Neuhaus adapted those to fit a one-man project.

Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

*Oh-My-Gosh*

## DECLARATION OF ORIGINALITY
## Bachelor's Thesis at the School of Engineering

By submitting this Bachelor's thesis, the undersigned student confirms that this thesis is his/her own work and was written without the help of a third party. (Group works: the performance of the other group members are not considered as third party).

The student declares that all sources in the text (including Internet pages) and appendices have been correctly disclosed. This means that there has been no plagiarism, i.e. no sections of the Bachelor thesis have been partially or wholly taken from other texts and represented as the student's own work or included without being correctly referenced.

Any misconduct will be dealt with according to paragraphs 39 and 40 of the General Academic Regulations for Bachelor's and Master's Degree courses at the Zurich University of Applied Sciences (Rahmenprüfungsordnung ZHAW (RPO)) and subject to the provisions for disciplinary action stipulated in the University regulations.

| City, Date: | Signature: |
|---|---|
| .......................................... | .......................................... |
| | .......................................... |
| | .......................................... |

The original signed and dated document (no copies) must be included after the title sheet in the ZHAW version of all Bachelor thesis submitted.

# Contents

# 1. Introduction

There was no thesis done on this subject that could have been used as reference. There are however several software projects that deal with a similar problem.

## 1.1. OpenSSH

The most noteworthy work to mention is of course SSH itself. *OpenSSH* (1999) is the name of the open source project which provides millions of administrators and developers with the ability to securely connect to a remote host. It replaces the up until then widely used protocols like *telnet(1)* (1994) (see 1.2) and *rlogin(1)* (1999)/*rsh(1)* (1999) (see 1.3).

SSH uses an own protocol to secure the communication channel between two peers and has earned itself a spot on the low end of the port table: It occupies port 22.

SSH's features can be used very flexibly: After it builds up a secure connection between a client and a server, it can be used to remotely log in and use a terminal on that machine. It can also forward traffic on local ports to the remote host through the secure channel. This is also used by third party programs such as *rsync(1)* (2018).

When it comes to the log in procedure itself, SSH allows for standard user log in using the Application Programming Interface (API) of the PAMs. Another feature is white listing of clients via their public keys, which stops intrusion attempts via hijacked user-password-credentials.

After a secure connection could be established, there are multiple possibilities to use the opened channel. One is to forward the Graphical User Interface (GUI) of a remote program to the client. Another one is to use this channel to tunnel more connections through it: For example can the traffic of an application which uses a specific port be forwarded to the remote host. This can obscure and secure this traffic between the host and the server.

The envisioned solution will simplify all the overloaded features of SSH and provide a light alternative.

## 1.2. Telnet

Telnet (C. Stephen 1969, Postel & Reynolds 1983) is an old (1969) and deprecated communication protocol which doesn't feature any security. However, in other implementations, Telnet Secure (TELNETS) was proposed, which features encryption over the communication channel. Telnet still has 23 as its very own port assigned to it.

Go-Telnet (Krempeaux 2016) is a TELNETS supporting client-server-application which has been implemented in Go.

In comparison to the envisioned project, the official Telnet solution does not provide a secure channel, which sets those two apart.

## 1.3. Berkeley r-commands

The Berkley r-commands are a set of commands to do certain tasks on remote hosts. Those tasks are similar to their counterparts without a leading "`r`".

- *rlogin(1)* (1999)

  This command connects to the host and performs a *login(1)* (2012) command, which includes authentication and if successful, spawning a user shell.

- *rsh(1)* (1999)

  `rsh` executes a command on the remote host. If no command is specified, the user gets logged in on the host with *rlogin(1)* (1999).

- *rexec(1)* (1996)

  With this command, the user can log in to a remote machine and execute one command.

- *rcp(1)* (1999)

  Using this command gives the user the ability to copy from and to a remote host.

- *rwho(1)* (1996)

  This command tells the user what users are currently logged in on the remote machine.

- *rstat(1)* (1996)

  `rstat` displays file system information from remote hosts.

- *ruptime(1)* (1996)

  With this command, the user can see the uptime, number of logged in users and current work load of the remote machine.

The envisioned solution will provide a secure channel to operate on a remote server, which the r-commands (specifically the *rlogin(1)* (1999) command) do not provide.

## 1.4. Task

This project's objective is to design and implement a prototype for an alternative to SSH. It has to be able to provide the user with the ability to have an interactive session on a remote machine. The solution has to be able to connect to a remote server and use a shell there. Both client and server-side have to be designed and implemented.

The official formulation of the tasks can be found in the appendix A.1.1.

The chapter "Design" (see 2) is meant to give a detailed overview of how the solution initially was envisioned and designed. In the course of the project's development, many aspects and views changed, as can be read in the minutes (see A.1.3). To clarify these changes in detail is the "Implementation" chapter's objective. In there, another overview has been worded, which should give sufficient insight on the project's development.

This thesis has been worded with technically literate readers in mind. However: For core concepts and special terms, a glossary can be found at 6. Used acronyms are listed in 6.4.

# 2. Design

To create a client-server protocol for remote login and interactive sessions, a clear cut architecture is mandatory. The flow of a typical use case should look like this:

1. The server listens for incoming connection.
2. The client dials server.
3. The server spawns the users login shell and forwards all traffic between shell and Client.
4. The client uses the shell.
5. The client terminates the session.
6. The server listens for new incoming connections.

However, there are multiple security concerns to be satiated:

- The connection between the client and server has to be secured from exposure to or manipulation from third parties.
- The client must authenticate itself for a user of the remote system with the appropriate credentials.
- The server has to drop privilege after a successful login to prevent privilege escalation.
- The server has to spawn the login shell of the logged in user with the appropriate rights.

Furthermore, this design does not allow for multiple sessions to be run in parallel. Therefore it was decided to run a new kernel-level thread to handle everything beginning after the connection has been established. This thread could also drop its privileges after the login succeeded.

Another action the server would have to take is to properly manage the session. That means the server has to make sure the login time stamp, who logged in, what device was used and other parameters are stored on the system. This is to act according to the Unix specification. This login accounting will allow for the logged in user to be displayed with commands like `w`.

This led to the following general flow of actions:

1. The server listens for incoming connection.
2. The client dials server.
3. The host handles the established connection.
4. The host sets up a secure connection (see 2.2).
5. The host requests necessary environment variables from the client.
6. The client responds with the values of said variables.
7. The host initiates a login procedure.
8. The client authenticates himself (see 2.3).
9. The host gathers essential data about the logged in user (see 2.4).
10. The host performs the necessary post-login actions that are expected in a Unix environment (see 2.6).

11. The host drops privilege to that of the logged in user (see 2.7).
12. The host spawns the users login shell (see 2.5) with the same credentials and forwards all traffic between shell and client.
13. The client uses the shell on remote machine.
14. The client terminates session.
15. The host terminates.

The sequence diagram in figure 2.1 is an illustration of how the flow of actions was envisioned.

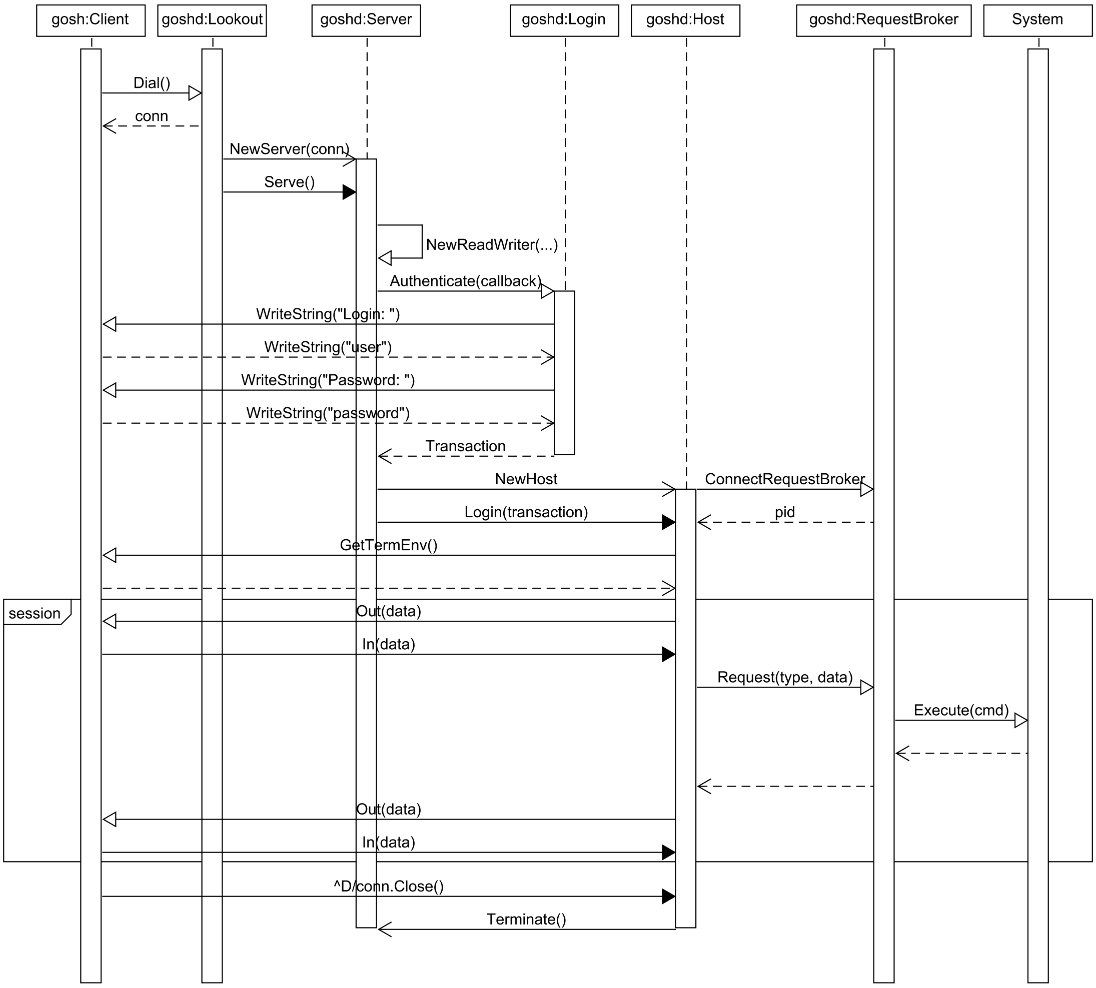



Figure 2.1.: Sequence diagram draft.

## 2.1. Implementation Language

In the beginning of the project, Dr Neuhaus suggested the use of Go as a modern low-level language over interpreted languages for considerations of security. He emphasized that the use of other low-level languages (like the C programming language (C) or the C++ programming language (C++)) was permissible. In the end the project was implemented in Go as suggested.

However, this led to a few problems in the implementation process (See 3.1).

## 2.2. Secure Connection

When building an application that communicates via the network, certain security measures are mandatory to ensure a secure communication. If such measures are not taken, the communication between the client and the server can be fully read and even altered by a third party. To prevent this, the communication can be encrypted with Transport Layer Security (TLS) (originally known as Secure Sockets Layer (SSL)). Today's state-of-the-art is TLS 1.3(Rescorla 2018).

When connecting, the client checks the server's X.509-certificate to authenticate the peer. Optionally, the client can also authenticate himself to the server. After this, the two start an encrypted channel by for example using the Diffie-Hellman key exchange or letting the server decrypt a random secret that has been encrypted with the servers public key. After this, every message can be transferred between client and server in an encrypted way.

## 2.3. Authentication via Password

For authentication of users on Linux systems, PAM exists. It provides a clean separation between a program and the sensitive part of the authentication. PAM communicates with an application using a transaction structure. This structure represents a conversation with PAM. Using this transaction, an application can do various actions regarding account management, authentication or session management. To initialize such a transaction, an application has to call the *pam_start(3)* (2016) function.

```
#include <security/pam_appl.h>

struct pam_message {
   int msg_style;
   const char *msg;
};

struct pam_response {
   char *resp;
   int resp_retcode;
};

struct pam_conv {
   int (*conv)(int num_msg, const struct pam_message **msg,
            struct pam_response **resp, void *appdata_ptr);
   void *appdata_ptr;
};
int pam_start(const char *service_name, const char *user, const struct
    pam_conv *pam_conversation, pam_handle_t **pamh);
```

Listing 2.1: Initializing a PAM conversation

Similarly, the created conversation has to be terminated with *pam_end(3)* (2016):

```
#include <security/pam_appl.h>

int pam_end(pam_handle_t *pamh, int pam_status);
```

Listing 2.2: Terminating a PAM conversation

After creating a conversation, the application can prepare the transaction with *pam_set_item(3)* (2016) and *pam_get_item(3)* (2016). Using those function, fields like `PAM_RUSER` and `PAM_RHOST`, which together (`PAM_RUSER@PAM_RHOST`) represent the requesting user (remote or local). If in any of the prior or later steps any errors occur, the corresponding error message can be received with *pam_strerror(3)* (2016).

```
#include <security/pam_appl.h>

```

```c
int pam_get_item(const pam_handle_t *pamh, int item_type, const void **item);
int pam_set_item(pam_handle_t *pamh, int item_type, const void *item);
const char *pam_strerror(pam_handle_t *pamh, int errnum);
```

Listing 2.3: PAM functions

Now that the transaction has been appropriately prepared, the application can request to authenticate a user via *pam_authenticate(3)* (2016):

```c
#include <security/pam_appl.h>

int pam_authenticate(pam_handle_t *pamh, int flags);
```

Listing 2.4: PAM authentication

Now PAM uses the `pam_conv` struct given in *pam_start(3)* (2016) that points to a function to interactively ask the application to authenticate the user. The application can then ask the user for the user name if not provided already and the password. After successful authentication, the function returns `PAM_SUCCESS` and the application should call *pam_setcred(3)* (2016) to establish and maintain the credentials of the logged in user.

```c
#include <security/pam_appl.h>

int pam_setcred(pam_handle_t *pamh, int flags);
```

Listing 2.5: PAM credential setting

Now the application can do other PAM related actions like session management (see 2.6.1). To finish, the application terminates the conversation properly and all associated memory gets invalidated.

## 2.4. User Data Querying

To spawn the default shell of a user, the path to said shell is required. This and other information can be found inside the `/etc/passwd` file. The file holds information about a users login-shell, password hash, groups, user information, User ID (UID), Group ID (GID) and home directory. To spawn a shell, the first and the last of these pieces of information are substantial to successfully log in a user. To query such data, Unix offers *getpwnam(3)/getpwuid(3)* (2019), which doesn't just read the `passwd` file, but also draws information about users from other sources, if they exist. This has been used in the implementation phase (see 3.5):

```c
#include <sys/types.h>
#include <pwd.h>

struct passwd {
   char *pw_name; /* username */
   char *pw_passwd; /* user password */
   uid_t pw_uid; /* user ID */
   gid_t pw_gid; /* group ID */
   char *pw_gecos; /* user information */
   char *pw_dir; /* home directory */
   char *pw_shell; /* shell program */
};

struct passwd *getpwnam(const char *name);
struct passwd *getpwuid(uid_t uid);
```

Listing 2.6: Definition of passwd and *getpwnam(3)/getpwuid(3)* (2019)

## 2.5. Starting the Shell

A shell requires a multitude of information before it can normally function:

- The name of the user and the host.
- The TERM variable to allow for the use of *ncurses(3X)* (2019) dependent applications.
- The size of the teletype (tty) window (including updates to the window's size).
- It has to be the session leader.

The user information depends on the user that starts the shell or more precisely: The UID and GID, with which it has been started. Note that this is not the same as the display name of the user in the USER variable. The host name is the same as the HOSTNAME environment variable of the parent process.

There is also to note that for the usage of the correct terminal sequences, the TERM environment variable has to be known to the shell. This is needed to use the correct escape sequences for example or for applications that use *ncurses(3X)* (2019).

A terminal or terminal emulator sends a window change signal (SIGWINCH) to its child process to notify a change in its window size. If this is not forwarded, then the width and size the shell assumes might collide with the actual values and lead to overflowing lines and unused space.

To take full control of the terminal, the shell has to be the session leader. This can be achieved by letting it set the Session ID (SID). If it is not set, the shell will print some errors regarding the ioctl device.

## 2.6. Login Accounting

When a user logs into a Unix machine, a session and a time stamp gets created.

For this, the two files /var/run/utmp and /var/log/wtmp provide the appropriate storage. The utmpx API offers appropriate functions. The utmpx struct represents a single entry in those files:

```c
#define _GNU_SOURCE
/* Without _GNU_SOURCE the two field names below are prepended by "__" */
struct exit_status {
   short e_termination; /* Process termination status (signal) */
   short e_exit;   /* Process exit status */
};
#define __UT_LINESIZE 32
#define __UT_NAMESIZE 32
#define __UT_HOSTSIZE 256
struct utmpx {
   short ut_type;     /* Type of record */
   pid_t ut_pid;     /* PID of login process */
   char ut_line[__UT_LINESIZE]; /* Terminal device name */
   char ut_id[4];     /* Suffix from terminal name, or ID field from
       inittab(5) */
   char ut_user[__UT_NAMESIZE]; /* Username */
   char ut_host[__UT_HOSTSIZE]; /* Hostname for remote login, or kernel
       version for run-level messages */
   struct exit_status ut_exit; /* Exit status of process marked as
       DEAD_PROCESS (not filled in by init(8) on Linux) */
   long ut_session;    /* Session ID */
   struct timeval ut_tv;   /* Time when entry was made */
   int32_t ut_addr_v6[4];   /* IP address of remote host (IPv4 address uses
       just ut_addr_v6[0], with other elements set to 0) */
   char __unused[20];    /* Reserved for future use */
};
```

Listing 2.7: Definition of the utmpx structure (Kerrisk 2010, p.819)

On login, a record has to be written to the `utmp` file to indicate that the user logged in. If there is already a record for the active terminal, then the entry has to be updated, otherwise a new entry has to be appended. A call to *pututxline(3)* (2017) should suffice in performing these steps properly. The application has to set the `ut_type` field of the `utmpx` struct to `USER_PROCESS` to mark a user login.

Similarly to the `utmp` update after login, the application has to report to the `utmpx` API that the session ended. This procedure consists of almost the same actions as the one after logging in, with exception of `ut_user` being zeroed out and `ut_type` being set to `DEAD_PROCESS`(Kerrisk 2010, p.828).

A program can also query the `utmp` file with the according `get` methods.

```
#include <utmp.h>

struct utmp *getutent(void);
struct utmp *getutid(const struct utmp *ut);
struct utmp *getutline(const struct utmp *ut);

struct utmp *pututline(const struct utmp *ut);
```

Listing 2.8: `utmpx` API functions

It is worth mentioning that the Linux command *login(1)* (2012) also uses PAM to authenticate a user.

### 2.6.1. Login Accounting with PAM

PAM supports binding login accounting to its own session functions *pam_open_session(3)* (2016) and *pam_close_session(3)* (2016):

```
#include <security/pam_appl.h>

int pam_open_session(pam_handle_t *pamh, int flags);
int pam_close_session(pam_handle_t *pamh, int flags);
```

Listing 2.9: PAM session management

## 2.7. Privilege Separation

Allowing a user to log in as any user, the authenticating process has to have `root` rights. However: After performing the authentication procedure and after successfully finishing the latter, a shell will be spawned for the user to interact with. This implies that without dropping the privilege of the process down to the appropriate privileges of the logged in user, the shell would run with `root` rights, which is a privilege escalation and not permissible.

Linux provides *setuid(2)* (2019) to set the owning user of a process and therefore also changing it's permissions. Unless the user is `root`, the process' owner cannot be changed.

In comparison: SSH chains the authentication of a user between an unprivileged child process that forwards the information between the two processes (see figure 2.2) - thereby putting a privilege separation in place.

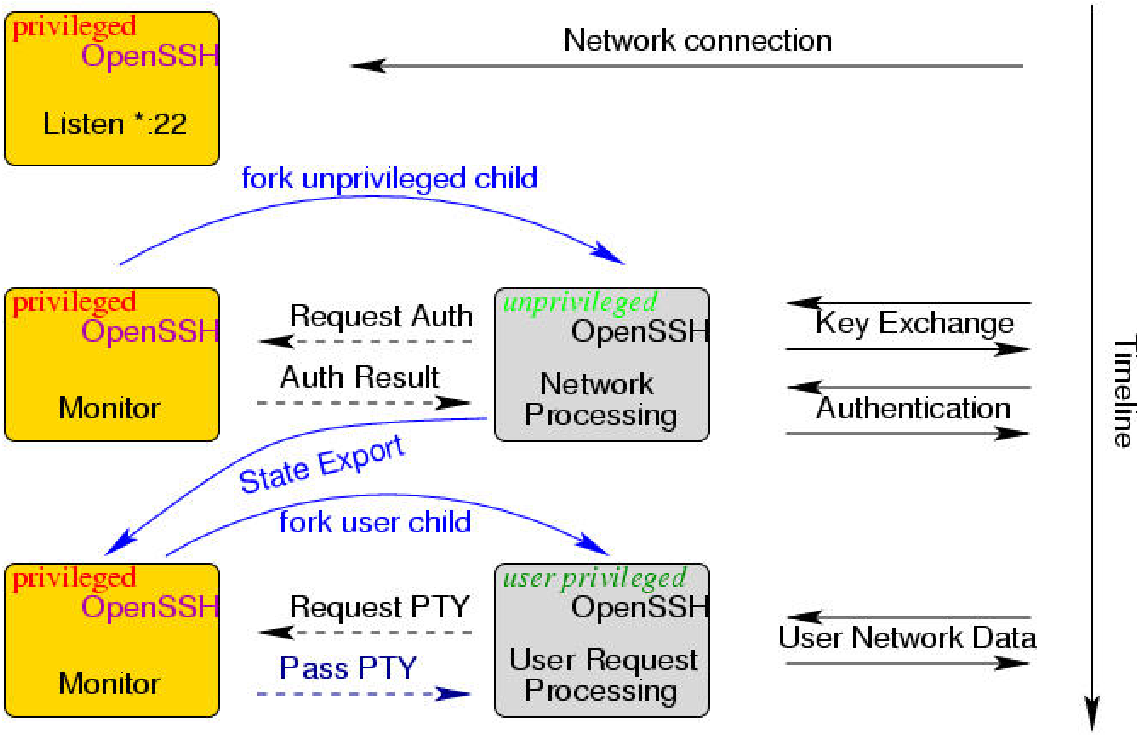



Figure 2.2.: Privilege separation in SSH (Provos 2003)

# 3. Implementation

The implementation of the project differs from the envisioned design. This new version of the server has two executables:

- One is `goshd`, a daemon (also refer to 3.10), which only has one function: To listen for incoming connections and then to start a child process of the second executable.
- This second executable is the host (`goshh`) and it is responsible for handling the new connection.

Now, the work flow of the solution looks as follows (note that unchanged entries are gray):

1. The server listens for incoming connection.
2. The client dials server.
3. The server executes `goshh` as a child process.
4. The host handles the established connection.
5. The host sets up a secure connection (see 3.2).
6. The host requests necessary environment variables from the client.
7. The client responds with the values of said variables.
8. The host notifies the client to prepare itself for the session.
9. the client sets the appropriate tty mode (see 3.8).
10. The host initiates a login procedure
    - The host finds a public key that belongs to the client. The host now initiates authentication via keys (see 3.4):
        a) The host encrypts a random secret with it and then sends that to the client.
        b) The client decrypts the secret and sends back the answer.
        c) The host checks whether the received answer matches the original secret:
            – If it matches, the authentication has succeeded.
                i. The host gathers essential data about the logged in user (see 3.5).
                ii. The host spawns the users login shell (see 2.5 and 3.6) with the credentials of the logged in user and forwards all traffic between shell and client.
            – If it fails, the authentication via keys has failed and instead, the authentication via password gets initiated-
    - The host does not find a public key that matches the client or the authentication with keys did not succeed.
        a) The host starts an instance of *login(1)* (2012), which asks the client to authenticate itself (see 3.3).
        b) After successful login, *login(1)* (2012) drops privileges, takes post-login actions and starts a user login shell.
11. The client uses the shell on remote machine.

12. The session ends in particular ways:
    - The client terminates session by logging out of the shell.
        a) The child process of the host terminates.
        b) The host terminates.
    - The client dies.
        a) The connection dies.
        b) The shell receives `EOF` and terminates.
        c) The host terminates.
    - The host dies or receives a interrupt signal (`SIGINT`).
        a) The connection dies.
        b) The client receives `EOF` and terminates.
    - The server dies or receives a `SIGINT`.
        a) The server sends a `SIGINT` to all the active hosts.
        b) The host terminates.
        c) The connection dies.
        d) The client receives `EOF` and terminates.

A new sequence diagram was created after the rough finishing of the project to display it's new work flow (see figure 3.1).

In the process of implementing this project, several problems arose that had to be addressed. The reason being that some envisioned features or mechanisms could not be implemented as originally thought. This is also the reason why the work flow described in the beginning of this chapter differs from that of the design in 2.

## 3.1. Problems

### 3.1.1. Forking

To handle new established connections, it was deemed important to *fork(2)* (2017) the process, as this duplicated the current process' memory and returns the Process ID (PID): 0 for the child process and a number greater than 0 for the parent to have the PID of the child.

In theory, this should have enabled the program to use Go's standard library capabilities to handle connections. However, there were several problems with this approach, which break the design discussed in 2.7:

#### Forking not supported

The Go standard library does not support the classical C-like forking. According to Google, Go does not have such mechanics, as Go was designed with Go-routines in mind instead. Go-routines however do not support privilege dropping. Instead, it only has a `syscall.ForkExec` method, which is documented as:

> Combination of *fork(2)* (2017) and *exec(3)* (2019), careful to be thread safe.

But since it uses *exec(3)* (2019) as well, it is the same as calling arbitrary binaries/scripts with the `exec.Cmd` function.

However: Go has a feature called CGo, which allows programs to call and interact with native C-routines. This opens up the possibility of using *fork(2)* (2017).

### Forking breaks Go objects

Forking with the functionality of CGo does not solve the problem either. The reason is that after forking there are two programs with a `net.Conn` object. This led to both connection objects being corrupted and turning unusable. Therefore, it was necessary to abandon the clean solution of forking and instead creating a new executable that can handle new connections by its own.

### Sharing Data with Child

The question then was: How can a process instantiate a child process and hand over all resources to it necessary for handling the new connection?

Since they are two separate processes now, they don't share any memory anymore. Hence the parent has to give the child the information about the connection via arguments. The most direct way to deal with this is to use file descriptors (fds), which can be passed as integer arguments to the child.

### Go Connection Cannot Be Transferred

Getting a `net.Conn` interface from a fd is supported in Go via:

```
fd := uintptr(0) // Dummy fd
conn, err := net.FileConn(os.NewFile(fd, "conn"))
if err != nil {
    panic(err.String())
}
```

Listing 3.1: Getting a `net.Conn` interface from a fd

Getting the fd from a connection is also possible:

```
file, err := conn.(*net.TCPConn).File()
if err != nil {
    panic(err.String())
}
fd := file.Fd()
```

Listing 3.2: Getting the fd from a `net.Conn` object

However: Creating a connection with the high-level API of Go and handing over the fd to the child to derive a `net.Conn` object from, fails.

This had some implications for the project: The listener on the server could not be created with the high-level like:

```
ln, err := net.Listen("tcp", ":8080")
if err != nil {
    // handle error
}
for {
    conn, err := ln.Accept()
    if err != nil {
        // handle error
    }
    go handleConnection(conn)
}
```

Listing 3.3: Go's high level API for listener

Instead the project had to rely on the low-level socket. The x-package `Unix` provides the necessary wrapper functions, which can be used instead.

The obvious drawback being having to rely on a x-package which is subject to change and not being able to use the higher-level methods which **are** part of the standard library.

### 3.1.2. Privilege Dropping

To have a sensible privilege separation, the host process should drop the `root` privileges to the privileges of the logged in user. But after performing the login procedure, setting the UID or GID both resulted in an error about insufficient permissions. This problem could not be solved in the course of this project. Instead, it was relied upon spawning the shell already with set UID and GID values to ensure appropriate permissions. This worked out well, as looking at the process in a process monitor like `htop` showed that the spawned shell does have the right UID and GID.

## 3.2. Secure Connection

As described in 2.2, TLS was used to secure the connection between the client and the server. For this, the Go-package `crypto/tls` has been used. This package requires both client and server to extend the `GODEBUG` variable with a flag to activate support for TLS 1.3:

```go
func init() {
    os.Setenv("GODEBUG", os.Getenv("GODEBUG")+",tls13=1")
}
```

Listing 3.4: Activating TLS 1.3 in Go

The server-side of the solution has to have a certificate, which has to be generated newly when actually installing the solution on a machine. For testing purposes, a self-signed certificate was created using *openssl(1)* (n.d.):

```
openssl req -newkey rsa:2048 -nodes -keyout key.pem -x509 -days 365 -out
    certificate.pem
```

Listing 3.5: Generating a self-signed certificate and private key

The files `key.pem` and `certificate.pem` can be found inside the `test` folder in the project's root folder.

The current implementation is set to skip verification of insecure certificates. Handling such a case has been decided to be handled in future improvements.

## 3.3. Authentication via Password

There are no official packages in Go that provide a wrapper to PAM (refer to 2.3). In earlier stages of the project, a wrapper from Steinert (2015) has been used. However, login using PAM failed on one of the test environments (namely Arch Linux). After some trial and error, switching to *login(1)* (2012) solved said issue, even though this command also uses PAM. This came with additional desired features like the login-accounting and other post-login actions (see 2.6 and 2.6.1), which the command already covers. After roughly a month of switching to *login(1)* (2012), the login procedure failed again on said environment. This occurred in a later stage of the project and it was deemed to time consuming, returning to the old implementation, which still lacked login accounting. Fortunately this only affected one of the environments: The other environments (i.e. Windows Subsystem for Linux (WSL)) were still functioning.

## 3.4. Authentication via Keys

A user can also authenticate himself without a password but instead using public key cryptography. For this, a user has to create a key pair consisting of a private and a public key. Before the actual authentication, a hashed version of the public key has to be stored on the server.

When starting an authentication, the user sends his public key to the server, which compares its hash with the stored keys that are deemed permissible for authentication. If it matches, the server encrypts a random secret (with high entropy) with the public key to the user. The user decrypts the message with his private key and sends it back to the server. If the returned secret matches the original secret, the user proved that he is the legitimate owner of the public key.

This authentication is sufficiently secure from third parties which do not have access to the private key of the user, as it can prove the authenticity of a user.

This mechanism has been implemented but with some alterations.

The first change was to store the public key in a plain text format, which was done out of convenience and can be changed in the future. Not storing the hashed public key doesn't pose any security threat, so fixing this deviation was deemed of low priority.

The second and last alteration was to have the structure of authorized keys on the server be stored exclusively in the `root` user's home directory. This is a sub-optimal approach, as his requires users to store their public keys in the super user's directory, which can lead to mistakes. This will be fixed in the future.

To test the application, a test user was created and given a key pair which was created as follows using *openssl(1)* (n.d.):

```
openssl genpkey -out client.pem -algorithm rsa -pkeyopt rsa_keygen_bits:2048
openssl rsa -in client.pem -out client.pub -pubout
```

Listing 3.6: Generating a key pair for the client

## 3.5. User Data Querying

Querying user data (see 2.4) is *partially* supported by the Go standard library. It can give all the information about a user on the system but its login shell. Therefore it was necessary to obtain said information in a different manner. One way to get the login shell of a user is to read in the `passwd` file of the system. This originally has been implemented by using a private Go package that parsed the `passwd` file. This was deemed incomplete, as a system can acknowledge users that are not listed in said file. An example of this is Network Information Service (NIS) or Lightweight Directory Access Protocol (LDAP), which would not be covered by solely relying on the `passwd` file.

In later stages, this has been corrected by switching to using CGo and calling the Unix API routines *getpwnam(3)/getpwuid(3)* (2019).

## 3.6. Starting the Shell

In Go, a process can be started with special settings. These allow for setting the UID, GID and even letting the process set itself as the session leader.

```go
cmd := exec.Command("/bin/bash", "--login")
cmd.SysProcAttr = &syscall.SysProcAttr{
   Setsid: true,
   Credential: &syscall.Credential{
      Uid: pwd.Uid,
      Gid: pwd.Gid,
   },
}
cmd.Env = []string{"TERM=xterm-256color"}
```

Listing 3.7: Starting a process in Go

## 3.7. Login Accounting

The current implementation either outsources the post-login actions described in 2.6 to *login(1)* (2012)(see 3.3) or omits them completely, as is the case in 3.4. The implementation of this feature is not part of the current project.

## 3.8. Terminal Mode

The client expects to send all its input directly to the shell without prior interpretation. To achieve this, the client-side terminal has to be set from the default `cooked`- into the `raw`-mode(Kerrisk 2010, p.1309). This sends each key stroke to the server-side shell as is. After the termination of the session, the original `cooked`-mode has to be restored to ensure operation as per usual.

Setting the terminal mode as described above in Go can be achieved with the functionality of the x-package "`golang.org/x/crypto/ssh/terminal`":

```
import "golang.org/x/crypto/ssh/terminal"
//...
oldState, err := terminal.MakeRaw(0)
if err != nil {
   panic(err)
}
defer terminal.Restore(0, oldState)
```

Listing 3.8: Setting the terminal mode in Go

## 3.9. Pseudoterminal

ptys are an Inter-Process-Communication (IPC) mechanism that help solving the problem of how two remote programs can communicate as if they were directly connected via a terminal (see figure 3.2).

A shell expects to be connected to a full fledged tty(or a tty emulator). To check whether it is inside such an environment, it uses *isatty(3)* (2019) on the fd it is connected to. Therefore the fd it is connected to should behave like a real terminal. This is where ptys come into play.

To do this, two files are created: The pseudoterminal master (ptm) and the corresponding pseudoterminal slave (pts)(see figure 3.3). Linux provides a pty generator at `/dev/ptmx`, which creates a ptm and a pts. This works by simply opening the `/dev/ptmx` file using *posix_openpt(3)* (2017). After this, a program has to grant the pts file ownership and permissions with *grantpt(3)* (2017), unlock it with *unlockpt(3)* (2017) and retrieve its file name with *ptsname(3)* (2017):

```
#define _XOPEN_SOURCE 500
#include <stdlib.h>

int posix_openpt(int flags);
int grantpt(int fd);
int unlockpt(int fd);
char *ptsname(int fd);
```

Listing 3.9: pty related Linux API functions

With the pty properly set up, a child bound to the pts side will assume it is connected to a terminal and also behave as such. This mechanism is key to running a shell and forwarding all traffic between a remote client and the shell. In case of SSH and the objective of this thesis, the layout looks like in figure 3.4.

Go does have a wrapper for ptys in an official package, but the code is inside an internal package, called `os/signal/internal/pty`, which uses CGo to call the required API routines described in 3.9:

```
// Open returns a master pty and the name of the linked slave tty.
func Open() (master *os.File, slave string, err error) {
   m, err := C.posix_openpt(C.O_RDWR)
   if err != nil {
      return nil, "", ptyError("posix_openpt", err)
   }
   if _, err := C.grantpt(m); err != nil {
      C.close(m)
```

```
        return nil, "", ptyError("grantpt", err)
    }
    if _, err := C.unlockpt(m); err != nil {
        C.close(m)
        return nil, "", ptyError("unlockpt", err)
    }
    slave = C.GoString(C.ptsname(m))
    return os.NewFile(uintptr(m), "pty-master"), slave, nil
}
```

Listing 3.10: Go's pty wrapper

For the project, this code has been altered to better fit the flow of the application.

### 3.9.1. Performance

The pty is an interface that needs data to be forwarded to and received from. This holds true for both sides (ptm and the pts) and therefore needs a mechanism to transfer said data. On the slave side, the shell itself already seamlessly communicates with the device via its standard pipes. A similar concept is applied for the client and its connection to the server, where the application communicates via the connection with the server. There, the client uses a Go-routine to asynchronously forward the standard input of the program to the connection.

The master side has to transfer the input from the remote client to the ptm and the output from the latter to the former as well. For this, the server uses two Go-routines that asynchronously forward the data from each stream to the other and waits until the shell process terminates.

Both these transfers are performed using Go's `WriteTo` method:

```
// WriteTo implements io.WriterTo.
// This may make multiple calls to the Read method of the underlying Reader.
// If the underlying reader supports the WriteTo method,
// this calls the underlying WriteTo without buffering.
func (b *Reader) WriteTo(w io.Writer) (n int64, err error)
```

Listing 3.11: `WriteTo` method of Go

This mechanism performed well enough for the project, as it didn't take up too much resources and could be conveniently parallelized.

## 3.10. Service Hosting

The `goshd` program has to run on a server to accept incoming requests. For ease of use, the program should be able to be run as a daemon. On Linux, this can be achieved using the service manager *systemd(1)* (n.d.), for which a unit file has been created, which is located inside the `init` directory of the project. After deployment of the software, the service can be controlled using *systemd(1)* (n.d.) commands:

```
# Setup of goshd
sudo systemctl enable goshd.service
sudo systemctl start goshd.service
sudo systemctl info goshd.service

# Breakdown of goshd
sudo systemctl stop goshd.service
sudo systemctl disable goshd.service
```

Listing 3.12: `goshd` service control

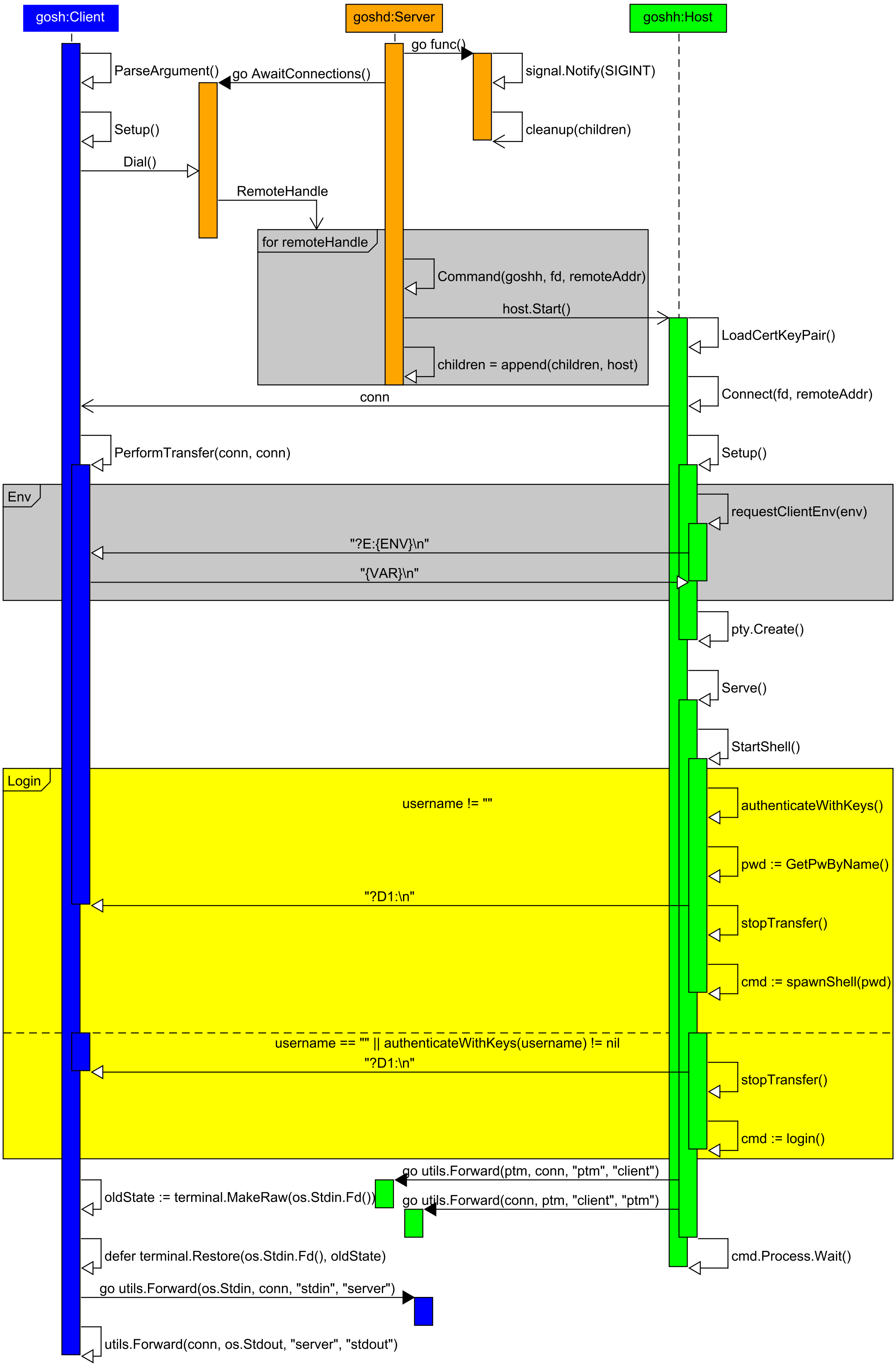



Figure 3.1.: Sequence diagram of current implementation.

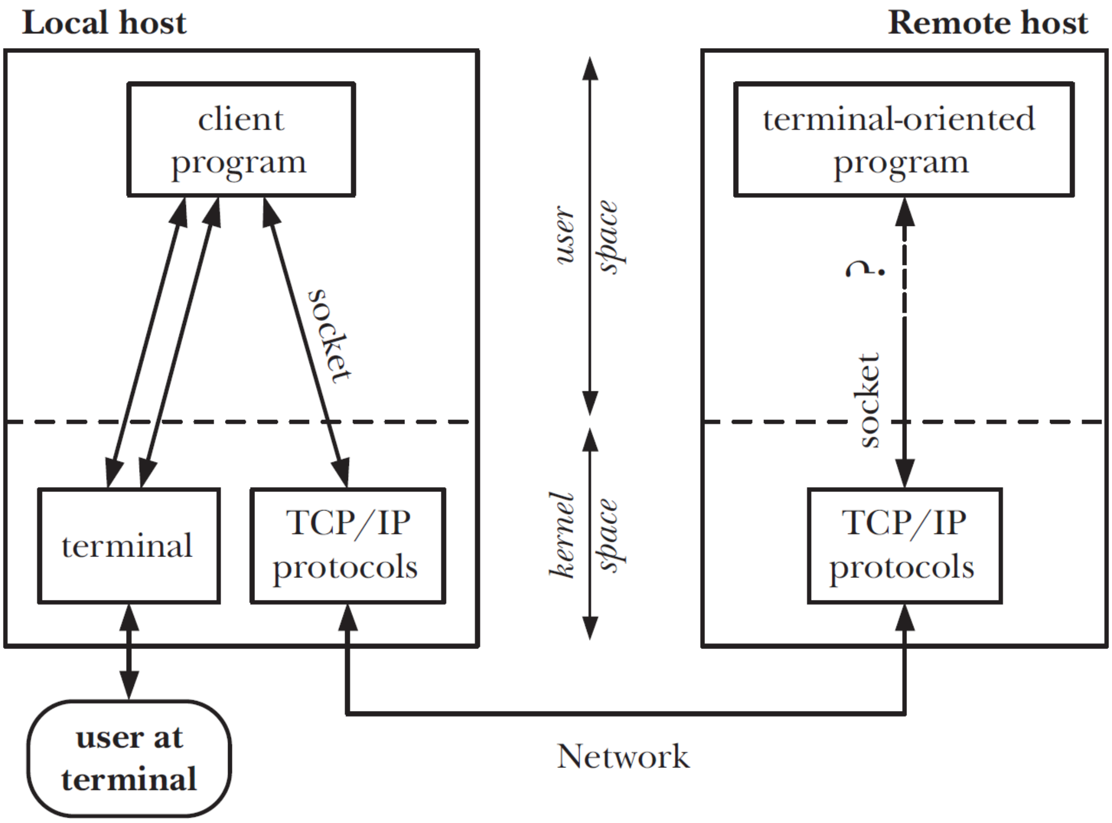



Figure 3.2.: How to operate a tty-oriented program over a network? (Kerrisk 2010, p.1376)

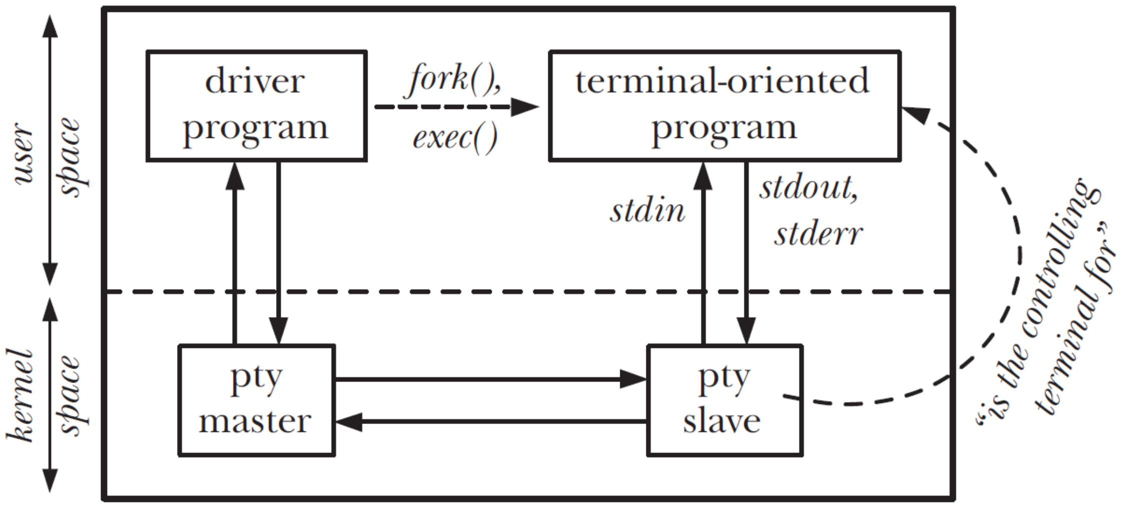



Figure 3.3.: Two programs communicating via a pty (Kerrisk 2010, p.1377)

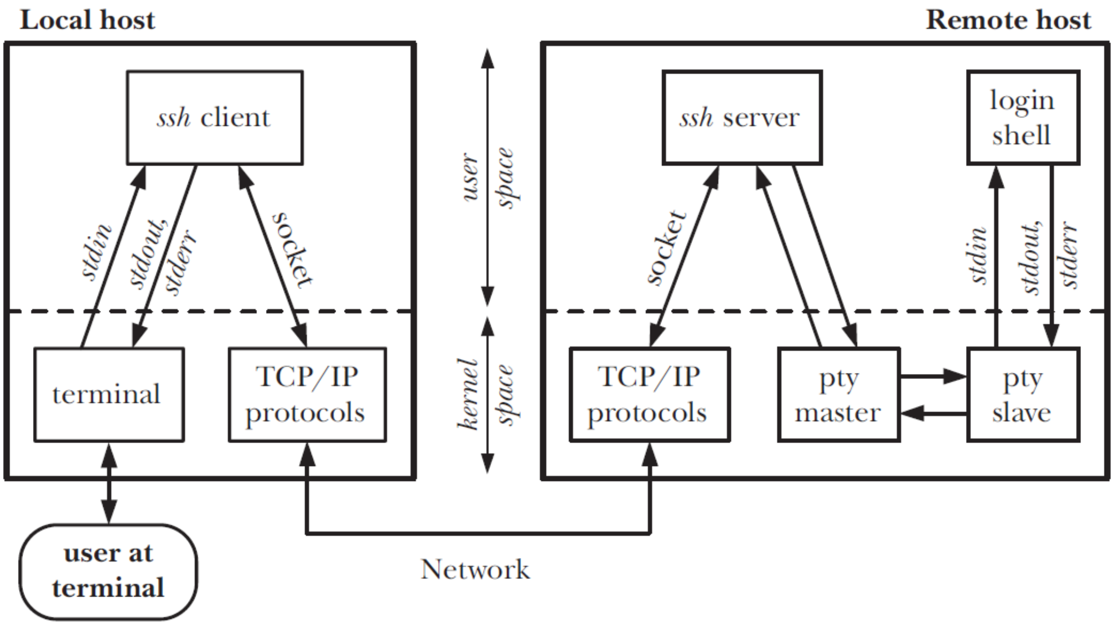



Figure 3.4.: How `ssh` uses a pty (Kerrisk 2010, p.1378)

# 4. Results

The project provides 3 applications:

- `gosh` is the application for the client side use case. It takes the following arguments:
  - `--help`: Displays the help text.
  - `--conf`: Sets the path where the configuration file is stored (defaults to “`/etc/gosh`”).
  - `--auth`: Sets the path where the key pair is stored (defaults to “ `/.gosh`”).
  - *string*: An optional address with optional credentials (defaults to `localhost`).
- `goshd` is the daemon on the server side that awaits incoming requests. It takes the following arguments:
  - `--help`: Displays the help text.
  - `--conf`: Sets the path where the configuration file is stored (defaults to “`/etc/gosh`”).
  - `--auth`: Sets the path where the key pair is stored (defaults to “ `/.gosh`”).
  - `--cert`: Sets the path of the server certificate (defaults to “`/etc/gosh/certificate.pem`”).
  - `--key`: Sets the path of the server key file (defaults to “`/etc/gosh/key.pem`”).
- `goshh` is a host that handles a new connection. This binary is only executed by the server and takes the following arguments:
  - `--help`: Displays the help text.
  - `--conf`: Sets the path where the configuration file is stored (defaults to “`/etc/gosh`”).
  - `--auth`: Sets the path where the key pair is stored (defaults to “ `/.gosh`”).
  - `--cert`: Sets the path of the server certificate (defaults to “`/etc/gosh/certificate.pem`”).
  - `--key`: Sets the path of the server key file (defaults to “`/etc/gosh/key.pem`”).
  - `--remote`: Sets the address of the peer (defaults to “`localhost:2222`”).
  - `--fd`: Provides the fd of the connection the host has to handle.

To configure the applications, the client side has a configuration file called `gosh_config.toml`. The server's configuration file is called `goshd_config.toml` respectively, where both `goshd` and `goshh` refer to the same file. In it, the user can specify the following parameters and a few more:

- The default port to communicate over.
- The default log level.
- The location of the authorized keys.
- The maximum amount of allowed sessions (server-side only).

The project allows a user to connect to a server and enter an interactive session with a remote user's login shell (see 3.8, 3.3, 3.4, 3.9 and 3.7). On the server side an appropriate privilege separation is performed upon user login (see 2.7 and 3.1). The communication is secured in the beginning of the transaction using TLS (see 3.2).

To test the applications, the user can execute the binaries from within the root folder of the project on both the client and the server (they can be the same machine, if so desired). To do this, the user can make use of the CLI flags described above to redirect the dependencies to the test files in the project:

```
# Setup environment: Create test user, its home directory and the login shell.
./scripts/setup.sh

# Start the server
sudo goshd --conf configs --auth test --cert test/certificate.pem --key
    test/key.pem

# Start the client
gosh --conf configs --auth test
# Or
gosh --conf configs --auth test 192.168.0.100
# Or
gosh --conf configs --auth test test@localhost
```

Listing 4.1: Running the applications for test purposes

To change the log level, giving the application the environment variable `LOG_LEVEL` set to the desired log level (i.e. “trace”, “debug”, etc.) should suffice.

# 5. Discussion And Prospects

As described in 4, the developed solution is capable of providing a TLS secured channel to an interactive session of a remote user's shell, of which the privilege has been dropped to the appropriate level for said user. It can be said that with this, the main goal of this thesis could be achieved. In the following list, an overview of the solutions to the official tasks are given and explained:

- **Design and implement a client-server protocol that can manage interactive sessions**
  This has been completed in 2 and 3 respectively.
- **Design and implement a privilege-separation architecture on the server side that allows safe dropping of privileges once a client establishes a connection**
  The design has been discussed in 2.7 and the implementation and problems encountered displayed in 3.1 respectively. Despite the encountered problems, a privilege separation was still realized.

As described in 1.4, the following tasks are required for a passing grade:

- **An introduction to the problem and why the envisaged solution will solve it**
  This has been discussed in 1.
- **A survey of related work in the area**
  Related works are listed and discussed in 1 as well (specifically in 1.1, 1.2 and 1.3).
- **A detailed design of the solution**
  The entire design can be found at 2.
- **An evaluation of the performance of the implemented solution**
  An explanation to this task can be found at 3.9.1.
- **A privilege-separation architecture**
  This is the same as the first task and can be considered solved as of 2.7 and 3.1.

In the following lists, there are additional tasks described in 1.4:

- **A comparison of all the related work with the envisaged solution, outlining why the envisaged solution is better**
  A small comparison to the surveyed solutions can be found at 1.1, 1.2 and 1.3.
- **A detailed analysis of the security of the solution, including possible attacks and defenses**
  This task will be cleared in this chapter.
- **Use of TLS as the transport layer**
  This topic is also considered solved as of 2.2 and 3.2.
- **A proof-of-concept client that can handle interactive sessions**
  A usable client has been developed and can be used after compilation of the source code. An overview of the developed applications can be found in 4.
- **A proof-of-concept client that works as a transport for rsync**
  This is a task that has not been solved in this thesis but its addition is discussed later in this chapter.

# 6. Index

## 6.1. Bibliography


Baushke, M. D. (2017), 'More Modular Exponentiation (MODP) Diffie-Hellman (DH) Key Exchange (KEX) Groups for Secure Shell (SSH)', *RFC* **8268**, 1–8.
**URL:** https://doi.org/10.17487/RFC8268 3

Bider, D. (2018*a*), 'Extension Negotiation in the Secure Shell (SSH) Protocol', *RFC* **8308**, 1–14.
**URL:** https://doi.org/10.17487/RFC8308 3

Bider, D. (2018*b*), 'Use of RSA Keys with SHA-256 and SHA-512 in the Secure Shell (SSH) Protocol', *RFC* **8332**, 1–9.
**URL:** https://doi.org/10.17487/RFC8332 3

Bider, D. & Baushke, M. D. (2012), 'SHA-2 Data Integrity Verification for the Secure Shell (SSH) Transport Layer Protocol', *RFC* **6668**, 1–5.
**URL:** https://doi.org/10.17487/RFC6668 3

C. Stephen, C. (1969), 'Network subsystem for time sharing hosts', *RFC* **15**, 1–5.
**URL:** https://doi.org/10.17487/RFC0015 6

*exec(3)* (2019).
**URL:** http://man7.org/linux/man-pages/man3/exec.3.html 16

*fork(2)* (2017).
**URL:** http://man7.org/linux/man-pages/man2/fork.2.html 16

*getpwnam(3)/getpwuid(3)* (2019).
**URL:** http://man7.org/linux/man-pages/man3/getpwnam.3.html 11, 19, 34

*grantpt(3)* (2017).
**URL:** http://man7.org/linux/man-pages/man3/grantpt.3.html 20

*isatty(3)* (2019).
**URL:** http://man7.org/linux/man-pages/man3/isatty.3.html 20

Kerrisk, M. (2010), *The Linux Programming Interface*, No Starch Press, San Francisco, CA, USA.
**URL:** http://www.man7.org/tlpi/ 13, 20, 23, 24, 33, 34

Krempeaux, C. I. (2016), 'go-telnet', Github.
**URL:** https://github.com/reiver/go-telnet 6

*login(1)* (2012).
**URL:** http://man7.org/linux/man-pages/man1/login.1.html 2, 6, 13, 15, 18, 19

Moorer, J. A. (1971), 'Second Network Graphics meeting details', *RFC* **253**, 1.
**URL:** https://doi.org/10.17487/RFC0253 3

*ncurses(3X)* (2019).
**URL:** http://man7.org/linux/man-pages/man3/ncurses.3x.html 2, 12

Neuhaus, S. (2018), 'Bachelorarbeit 2019 - FS: BA19_neut_03'.
**URL:** https://tat.zhaw.ch/tpada/arbeit_vorschau.jsp?arbeitID=16096 3

*OpenSSH* (1999).
**URL:** `https://www.openssh.com/` 6

*openssl(1)* (n.d.).
**URL:** `https://linux.die.net/man/1/openssl` 18, 19

*pam_authenticate(3)* (2016).
**URL:** `http://man7.org/linux/man-pages/man3/pam_authenticate.3.html` 11

*pam_close_session(3)* (2016).
**URL:** `http://man7.org/linux/man-pages/man3/pam_close_session.3.html` 13

*pam_end(3)* (2016).
**URL:** `http://man7.org/linux/man-pages/man3/pam_end.3.html` 10

*pam_get_item(3)* (2016).
**URL:** `http://man7.org/linux/man-pages/man3/pam_get_item.3.html` 10

*pam_open_session(3)* (2016).
**URL:** `http://man7.org/linux/man-pages/man3/pam_open_session.3.html` 13

*pam_setcred(3)* (2016).
**URL:** `http://man7.org/linux/man-pages/man3/pam_setcred.3.html` 11

*pam_set_item(3)* (2016).
**URL:** `http://man7.org/linux/man-pages/man3/pam_set_item.3.html` 10

*pam_start(3)* (2016).
**URL:** `http://man7.org/linux/man-pages/man3/pam_start.3.html` 10, 11

*pam_strerror(3)* (2016).
**URL:** `http://man7.org/linux/man-pages/man3/pam_strerror.3.html` 10

*posix_openpt(3)* (2017).
**URL:** `http://man7.org/linux/man-pages/man3/posix_openpt.3.html` 20

Postel, J. & Reynolds, J. K. (1983), 'Telnet protocol specification', *RFC* **854**, 1–15.
**URL:** `https://doi.org/10.17487/RFC0854` 6

Provos, N. (2003), 'Privilege Separated OpenSSH'.
**URL:** `http://citi.umich.edu/u/provos/ssh/privsep.html` 14, 33

*pts(4)* (2013).
**URL:** `http://man7.org/linux/man-pages/man4/pts.4.html` 32

*ptsname(3)* (2017).
**URL:** `http://man7.org/linux/man-pages/man3/ptsname.3.html` 20

*pty(7)* (2017).
**URL:** `http://man7.org/linux/man-pages/man7/pty.7.html` 31

*pututxline(3)* (2017).
**URL:** `http://man7.org/linux/man-pages/man3/pututxline.3.html` 13

*rcp(1)* (1999).
**URL:** `https://linux.die.net/man/1/rcp` 7

Rescorla, E. (2018), 'The transport layer security (TLS) protocol version 1.3', *RFC* **8446**, 1–160.
**URL:** `https://doi.org/10.17487/RFC8446` 10

*rexec(1)* (1996).
**URL:** `https://linux.die.net/man/1/rexec` 7

*rlogin(1)* (1999).
**URL:** https://linux.die.net/man/1/rlogin 6, 7

*rsh(1)* (1999).
**URL:** https://linux.die.net/man/1/rsh 6, 7

*rstat(1)* (1996).
**URL:** https://linux.die.net/man/1/rstat 7

*rsync(1)* (2018).
**URL:** http://man7.org/linux/man-pages/man1/rsync.1.html 6

*ruptime(1)* (1996).
**URL:** https://linux.die.net/man/1/ruptime 7

*rwho(1)* (1996).
**URL:** https://linux.die.net/man/1/rwho 7

*setuid(2)* (2019).
**URL:** http://man7.org/linux/man-pages/man2/setuid.2.html 14

Steinert, M. (2015), 'Go pam', Github.
**URL:** https://github.com/msteinert/pam 18

*systemd(1)* (n.d.).
**URL:** http://man7.org/linux/man-pages/man1/systemd.1.html 21

*telnet(1)* (1994).
**URL:** https://linux.die.net/man/1/telnet 6

*tty(4)* (2019).
**URL:** http://man7.org/linux/man-pages/man4/tty.4.html 32

*unlockpt(3)* (2017).
**URL:** http://man7.org/linux/man-pages/man3/unlockpt.3.html 20

## 6.2. Glossary

**Application Programming Interface**
Accessible interface for developers to use external code. 6

**CGo** C support for Go. 16, 17, 19, 20

**Command Line Interface**
A text based interface centered around commands to perform specific tasks. 3

**daemon** A (possibly latent) background process on a Unix machine that handles certain events when said events occur. 15, 21, 25

**Diffie-Hellman key exchange** A key exchange algorithm that prevents exposure to eavesdropping third parties. 10

**file descriptor**
A file descriptor is an integer that represents the handle to a file. 17

**Graphical User Interface**
Graphical interface for the user to visually interact with a program. 6

**Group ID**
The unique identifier of a group represented as an integer. 11

**Inter-Process-Communication**
Communication between processes. 20

**interrupt signal**
A signal supported by Unix based Operating Systems(OSes) which signals to a process to interrupt it's work. 16

**Lightweight Directory Access Protocol**
A protocol for querying and managing information of distributed directory information services. 19

**Linux** A Unix based OS which uses Linus Torvalds kernel and was inspired by Minix. 13, 14, 18, 20, 21, 32, 34

**login** The action of logging in. plural 8, 9, 11, 13, 15, 18, 19, 25

**Network Information Service**
A service for distributing configurations like user information among a network. 19

**Object Oriented Programming**
A programming paradigm which uses objects to model real life entities. 32

**Pluggable Authentication Module**
Modules for user authentication. 2

**port** A point for traffic to flow, represented by an unsigned integer of up to 2 bytes. The name was chosen as an analogy to ports for ships. 6, 32

**Process ID**
The unique identifier of a process represented as an integer. 16

**pseudoterminal**
A mechanism of Unix to allow programs to communicate as if the other was inside a tty (*pty(7)* 2017). 2

**pseudoterminal master**
The master of a pty. 20

**pseudoterminal slave**
The slave of a pty (*pts(4)* 2013). 20

**Secure Shell**
An client-server-application that allows remote login and interaction with a shell. See 1.1. 2

**Secure Sockets Layer**
Cryptographic protocol to secure the communication between two peers via symmetric cryptography. Deprecated. 10

**Session ID**
The unique identifier of a session represented as an integer. 12

**shell** A CLI program, that reads user input line-by-line and executes those commands. 2, 6–9, 11, 12, 14–16, 18–21, 25, 27, 32

**socket** A network socket is an endpoint for communication over ports. 17

**teletype**
Originally a device that could send and receive text messages. Nowadays tty refers to terminals, which emulate that behaviour (*tty(4)* 2019). 12

**Telnet Secure**
Telnet with SSL encryption. 6

**terminal** An user interface to interact with CLI programs like shells. 6, 13, 20, 32, 34

**the C programming language**
Low-level programming language originally invented by Dennis Ritchie. 9

**the C++ programming language**
Descendant of C which implemented Object Oriented Programming (OOP). 9

**the Go/Golang programming language**
Google's programming language. 3

**Transport Layer Security**
Newer and recommended version of SSL. 10

**Unix** An originally free OS family called Unics from AT&T that re-imagined an older OS by the name of Multics. 2, 8, 11, 12, 19, 31, 32

**uptime** The time a computer has been running. 7

**User ID**
The unique identifier of a user represented as an integer. 11

**window change signal**
A signal supported by Unix based OSes which signals to a process that the terminal window it is running in has changed size. 12

**Windows Subsystem for Linux**
A subsystem provided by Windows that is able to run headless Linux distributions. 18

**x-package** A Go package that is not part of the standard library and that is subject to change or even entirely disappear. plural 17

**X.509** An international standard for certificates in public key infrastructures. 10

## 6.3. List of Figures

## 6.4. List of Listings

## 6.5. Acronym Glossary

**`SIGINT`** *interrupt signal* 16, *See* interrupt signal

**`SIGWINCH`** *window change signal* 12, *See* window change signal

**API** *Application Programming Interface* 6, 12, 13, 17, 19, 20, 34, *See* Application Programming Interface

**C** *the C programming language* 9, 16, 31, 32, *See* the C programming language

**C++** *the C++ programming language* 9, *See* the C++ programming language

**CLI** *Command Line Interface* 3, 25, 32, *See* Command Line Interface

**fd** *file descriptor* 17, 20, 25, 34, *See* file descriptor

**GID** *Group ID* 11, 12, 18, 19, *See* Group ID

**Go** *the Go/Golang programming language* 3, 6, 9, 16–21, 31, 32, 34, *See* the Go/Golang programming language

**GUI** *Graphical User Interface* 6, *See* Graphical User Interface

**IPC** *Inter-Process-Communication* 20, *See* Inter-Process-Communication

**LDAP** *Lightweight Directory Access Protocol* 19, *See* Lightweight Directory Access Protocol

**NIS** *Network Information Service* 19, *See* Network Information Service

**OOP** *Object Oriented Programming* 32, *See* Object Oriented Programming

**OS** *Operating System* 31, 32

**PAM** *Pluggable Authentication Module* 2, 6, 10, 11, 13, 18, 34, *See* Pluggable Authentication Module

**PID** *Process ID* 16, *See* Process ID

**ptm** *pseudoterminal master* 20, 21, *See* pseudoterminal master

**pts** *pseudoterminal slave* 20, 21, *See* pseudoterminal slave

**pty** *pseudoterminal* 2, 20, 21, 23, 24, 32–34, *See* pseudoterminal

**SID** *Session ID* 12, *See* Session ID

**SSH** *Secure Shell* 2, 3, 6, 7, 14, 20, *See* Secure Shell

**SSL** *Secure Sockets Layer* 10, 32, *See* Secure Sockets Layer

**TELNETS** *Telnet Secure* 6, *See* Telnet Secure

**TLS** *Transport Layer Security* 10, 18, 25, 27, 34, *See* Transport Layer Security

**tty** *teletype* 12, 15, 20, 23, 31–33, *See* teletype

**UID** *User ID* 11, 12, 18, 19, *See* User ID

**WSL** *Windows Subsystem for Linux* 18, *See* Windows Subsystem for Linux

**ZHAW** *Zurich University of Applied Sciences See* Zurich University of Applied Sciences

# A. Appendix

## A.1. Project Management

### A.1.1. Official Statement of Tasks

Bachelor Thesis
Preventing Supply Chain Insecurity
by Authentication on Layer 2

Stephan Neuhaus

2017-06-15

# 1 Introduction

The SSH protocol [RFC253, RFC6668, RFC8268, RFC8308, RFC8332] is now over twelve years old in its current form. One of the problems with SSH is its complexity, both in the initial phase when key material is exchanged, but also later, for example because the server must always decide whether to return a character that has been sent to it or not (echo).

The goal of this work is a radically simplified protocol, which in its functions is similar to SSH. (N.B. the similarity concerns the functions, not necessarily the protocol details). You develop the protocol, as well as a client and a server. You demonstrate that your software can replace SSH. For a merely passing grade, this may be done by specifying and implementing a suitable protocol. For an improved grade, you show that your replacement can handle several common use cases, among them:

- Interactive session
- Rsync with the SSH replacement as transport protocol

# 2 Task

To this end, this thesis will

- design and implement a client–server protocol that can manage interactive sessions
- design and implement a privilege-separation architecture on the server side that allows safe dropping of privileges once a client establishes a connection

For a passing grade (4.0), the work must contain at least the following:

- in the thesis, an introduction to the problem and why the envisaged solution will solve it;
- in the thesis, a survey of related work in the area;
- in the thesis, a detailed design of the solution;

- in the thesis, an evaluation of the performance of the implemented solution; and
- in the software, a privilege-separation architecture.

These requirements do not contain anything related to security. This is not an accident.

Incorporating the following components will improve the grade. The more components are included, the better the grade will be.

- In the related work section of the thesis, a comparison of all the related work with the envisaged solution, outlining why the envisaged solution is better;
- in the thesis, a detailed analysis of the security of the solution, including possible attacks and defenses;
- use of TLS as the transport layer;
- a proof-of-concept client that can handle interactive sessions;
- a proof-of-concept client that works as a transport for rsync;

ZHAW's School of Engineering no longer provides formal language lessons for its students as part of the curriculum. I am therefore giving notice that submitting a thesis with large amounts of orthographical or grammatical errors lead to a lower grade.

The thesis can be submitted in German or English. English is prefered, but submitting in German will not lead to a lower grade.

### A.1.2. Project Plan

# Revised Project Plan

March 2019

## 1 Introduction

The work on the final thesis is divided into several sub tasks. The individual tasks and respective time planning was defined early.
As this is a field of work, we as a team are not familiar with, we have decided to change our original plan: "Project Plan" to a revised version.
The biggest difference is that the Protype is developed earlier but we are removing some security measures respectively moving it to an optional goal We are subdividing the following area of development:

- Technical research: The research starts with a collection of examined current solution to a secure data transference, followed by a list of pro and cons for the approaches that includes a preferred selection. Moreover, we would survey the current state development within that field.
- Conceptualising: Look for alternative solutions and compare them. Plan the development.
- Prototype: A unsecure SSH Client
- Testing: Do generalized tests, identify possible security vulnerability
- Rework of the secure shell: Modify the secure shell based on the newly discovered needs

Furthermore there are specific milestone within the project process, which we would use to realign and discuss our time division.

## 2 Visualization

The project plan is documented in the form of a chart and is updated throughout the project. This way, deviations can be detected early and can be discussed with the supervisor and within our team.

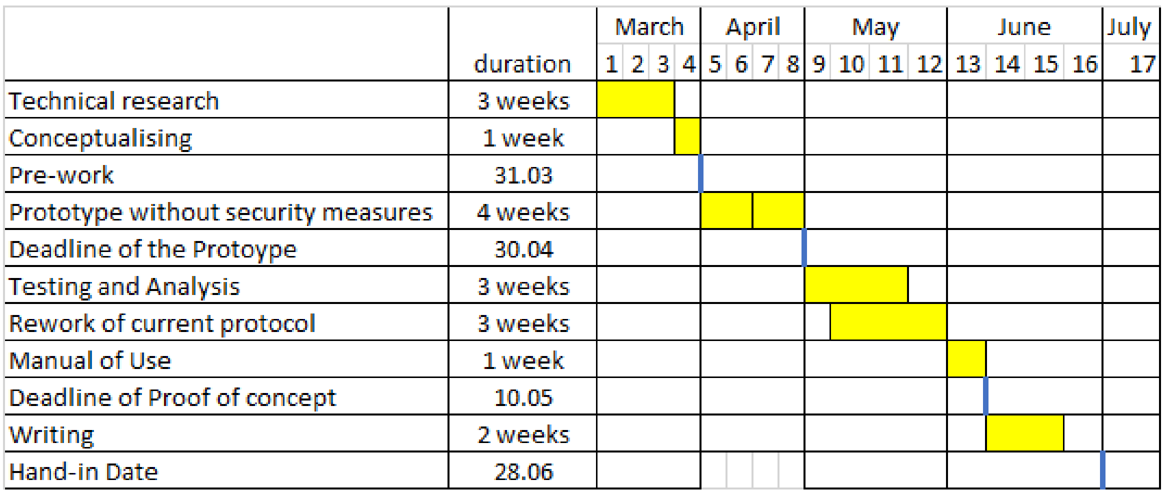
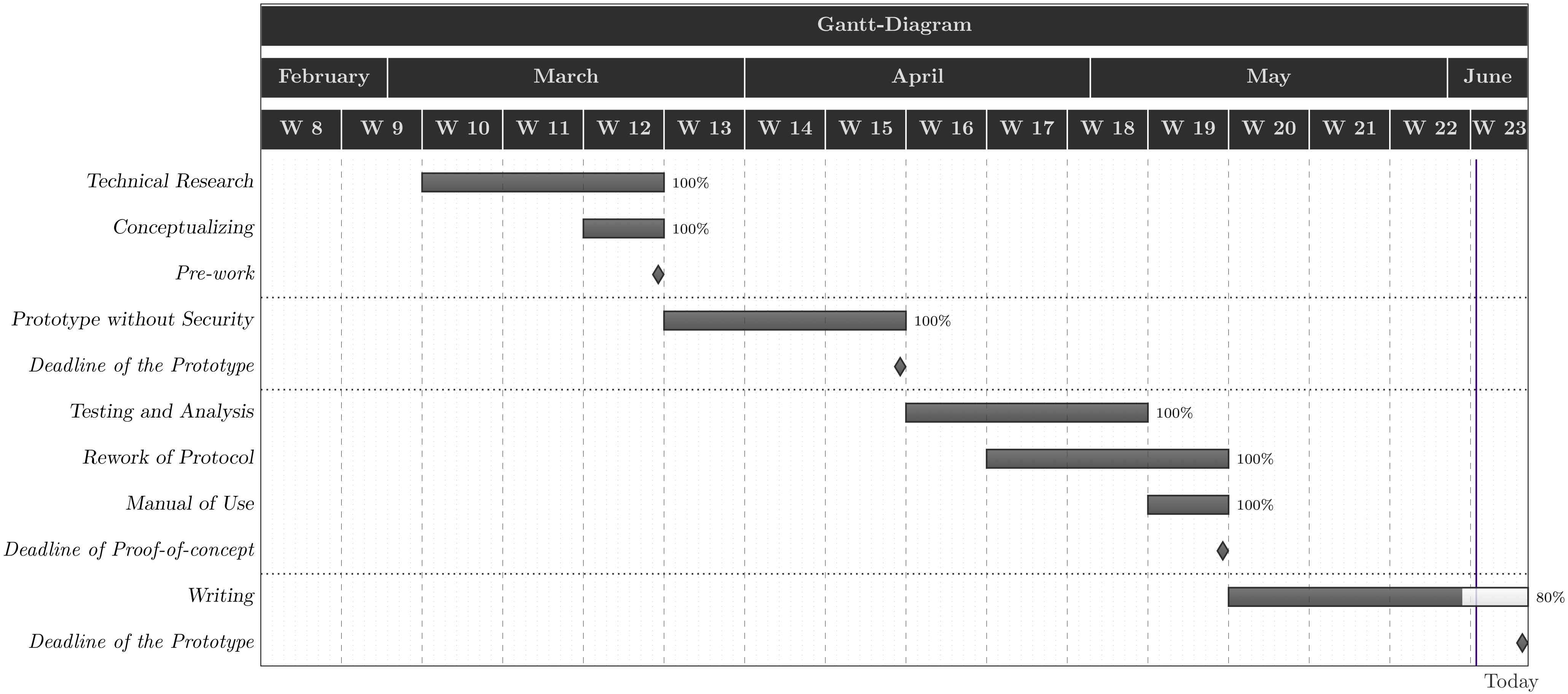



Figure 1: Project plan


Gantt-Diagram
February
March
April
May
June
W 8
W 9
W 10
W 11
W 12
W 13
W 14
W 15
W 16
W 17
W 18
W 19
W 20
W 21
W 22
W 23
Technical Research
100%
Conceptualizing
100%
Pre-work
Prototype without Security
100%
Deadline of the Prototype
Testing and Analysis
100%
Rework of Protocol
100%
Manual of Use
100%
Deadline of Proof-of-concept
Writing
80%
Deadline of the Prototype
Today

### A.1.3. Meeting Minutes

The meeting minutes have a disruption in style and execution beginning from the 6th meeting. Reason for this is because in the beginning of the project, Mr Schwarz was responsible for keeping the minutes, but he opted out of the project.

# Oh my Gosh - Meeting protocol

## 1 2nd Meeting

**Participant**
*Bachelor thesis supervisor - Stephan Neuhausen*
*Bachelor student - Raphael Emberger*
*Bachelor student - Kevin Schwarz*
**Time duration of the meeting**
*1 hour and 15 minutes*

### 1.1 Objectives

1. Keeping the bachelor thesis supervisor informed on the state of affairs
2. Getting an overview on how the progress relate to the scheduled progress

### 1.2 Summary

In this meeting the following things were achieved:

1. Decision towards ITC and UDP was achieved
2. A rough draft of a generalized process was finalized and presented to the supervisor
   (a) Smaller misconceptions were resolved
   (b) Fields where further research is warranted was shown
3. Reiteration of the project goal
4. Further Delimitation of the project extent.
   (a) Smaller misconceptions were resolved

### 1.3 Tasks and resources

The following were left as tasks or as a research subject for the next meeting in descending priority:

1. Understanding shell forwarding
2. Researching the limitation of IOCTL - raw and device specific output
3. Pseudo terminals
4. Persistence in relation to Environment variables

### 1.4 Next meeting

The next meeting plan were not changed, the formerly decided weekly scheduled date still stands.
The next meeting is dated: [14 / 03 / 19] on the first floor of the Zürich location of the ZHAW within the Room 0.03.

# Oh my Gosh - Meeting protocol

## 1 3rd Meeting

**Participant**
*Bachelor thesis supervisor - Stephan Neuhausen*
*Bachelor student - Raphael Emberger*
*Bachelor student - Kevin Schwarz*
**Time duration of the meeting**
*1 hour*

### 1.1 Objectives

1. Keeping the bachelor thesis supervisor informed on the state of affairs
2. Getting an overview on how the progress relate to the scheduled progress

### 1.2 Summary

In this meeting the following things were achieved:

1. General overview of a PTY
2. Certain foundation towards developing a non-secure secure shell were explained:
   (a) Smaller misconceptions were resolved
   (b) Fields where further research is warranted was shown
3. Reiteration of the project goal
4. Further Delimitation of the project extent.
   (a) Smaller misconceptions were resolved

### 1.3 Tasks and resources

The following were left as tasks or as a research subject for the next meeting in descending priority:

1. Understanding shell forwarding
2. Researching the limitation of IOCTL - raw and device specific output
3. Pseudo terminals
4. Persistence in relation to Environment variables

### 1.4 Next meeting

The next meeting plan were not changed, the formerly decided weekly scheduled date still stands.
The next meeting is dated: [14 / 03 / 19] on the first floor of the Zürich location of the ZHAW within the Room 0.03.

# Oh my Gosh - Meeting protocol

## 1 4$^{th}$ Meeting

**Participent**
*Bachelor thesis supervisor - Stephan Neuhausen*
*Bachelor student - Raphael Emberger*
*Bachelor student - Kevin Schwarz*
**Time duration of the meeting**
*25 Minutes*

### 1.1 Objectives

1. Keeping the bachelor thesis supervisor informed on the state of affairs
2. Gain an introduction to encryption

### 1.2 Summary

In this meeting the following things were achieved:

1. Introduction to Bash was given
2. Move up of the prototype deadline
3. Change of scheduled development plan:
   (a) Decrease of the SSH scope
   (b) Reduction of the security measures to an optional goal
   (c)

### 1.3 Tasks and resources

The following were left as tasks or as a research subject for the next meeting in descending priority:

1. Researching the infrastructure of GO-order
2. Researching Bash and PTY on Windows enviroment
3. First Server - Client Demo
4. Crafting a simple process diagram for the next meeting

### 1.4 Next meeting

The next meeting plan were not changed, the formerly decided weekly scheduled date still stands.
The next meeting is dated: [28 / 03 / 19] on the first floor of the Zürich location of the ZHAW within the Room 0.03.

# Oh my Gosh - Meeting protocol

## 1 5th Meeting

**Participent**
*Bachelor thesis supervisor - Stephan Neuhausen*
*Bachelor student - Raphael Emberger*
*Bachelor student - Kevin Schwarz*
**Time duration of the meeting**
*45 Minutes*

### 1.1 Objectives

1. Keeping the bachelor thesis supervisor informed on the state of affairs
2. Demonstrate Demo

### 1.2 Summary

In this meeting the following things were achieved:

1. The Prototype was tested.
2. Process Diagram was explained
3. 4 open Problems were discussed:
   (a) PAM Struct and how they work
   (b) Generalized Certkey location
   (c) Use of the Prototype within Linux
   (d) Correct pipe lining and forking

### 1.3 Tasks and resources

The following were left as tasks or as a research subject for the next meeting in descending priority:

1. Further testing of both Linux, Apple and Windows environment
2. Solving of Pam Struct problem
3. Further development

### 1.4 Next meeting

The next meeting plan were modified, the formerly decided weekly scheduled date still stands.
The next meeting is dated: [5 / 04 / 19] on the first floor of the Zürich location of the ZHAW within the Room 0.13.

# Design and Implementation of an Alternative to SSH

**Meeting Minutes**

## 1 Attendees

**Present:** Stephan Neuhaus, Raphael Emberger
**Absent:** Kevin Schwarz*(illness)*

## 2 Initiation

The meeting took place on the *Friday, 5th of April 2019, 13:00* in *ZL0.13, Lagerstrasse 45, Zürich*. Raphael Emberger was resonsible for the minutes.

## 3 Points of discussion

### 3.1 Process forking unsuccessful

Attempts on forking a sub-process were unsuccessful. The reason for this was that the standard library of Go doesn't allow such mechanics, as Go was designed with go-routines in mind instead.

**Solution** A quick test with cgo yielded a viable solution to the problem: Using the C-routine fork() a fork was successful.

### 3.2 Shell instantiating and forwarding

Attempts in forwarding the client connection to a server-side shell's stdin and its stdout and stderr to the connection of the client were unsuccessful.

**Solution** One quick tests showed that hooking up the std* pipes to a local shell process with Go worked just fine. Therefore it was deemed feasible to transfer the entire interface to the client.

### 3.3 Participation of Mr. Schwarz

Up until this date, the participation of Mr Schwarz was remarkable little in terms of writing on the code base of the project. The present parties agreed on this matter.

**Solution** It was decided to give Mr Schwarz a choice of action: Either he starts to participate heavily in the project from now on or he opts out of the project entirely.

## 4 Old Business

- **Login attempts in Linux fail:** This problem was deemed lower priority, as Login works on the WSL and can still be dealt with in later stages of the project.

## Next Meeting

*Friday, 5th of April 2019, 13:00* in *ZL0.13, Lagerstrasse 45, Zürich*

# Design and Implementation of an Alternative to SSH

**Meeting Minutes**

## 1 Attendees

**Present:** Stephan Neuhaus, Raphael Emberger, Kevin Schwarz

## 2 Initiation

The meeting took place on the *Friday, 12th of April 2019, 13:00* in *ZL0.13, Lagerstrasse 45, Zürich*. Raphael Emberger was resonsible for the minutes.

## 3 Points of discussion

### 3.1 Participation of Mr. Schwarz

Mr Schwarz decided to opt out of the project because of time issues. Mr Neuhaus will therefore adapt the outline of the project.

### 3.2 Reading user data works

The new module to read user data via the getpwnam(3) API has been implemented using cgo. It can read all the required data(i.e. the user shell which wasn't supported in the go standard library).

### 3.3 Forking implemented, but causes problems

Forking has been implemented via cgo but after forking, the net.Conn object cannot be used by the child process. There is also the to further investigate, whether after forking a new process actually gets started, as a quick look at the processes didn't reveal that a fork has been processed.

**Solution** To counter this problem it is suggested to do the connection build up via cgo using the C-socket API. This returns an integer as a file descriptor, which shouldn't cause problems when forking.

### 3.4 Remote start and handling of a shell has issues

After successfully hooking up the channels from the client to the shell process, almost all mechanics work as expected with exception of missing characters like the PS{1,2,3,4} prompts.

**Solution** It is suggested to compare the environment variables of the child shell process and the usual terminals to see if there are deal breaking differences. Adjusting the child shells environment variables might fix the problem.

## 4 Old Business

- **Login attempts in Linux fail:** This problem was deemed lower priority, as Login works on the WSL and can still be dealt with in later stages of the project.

## Next Meeting

*Friday, 26th of April 2019, 13:00* in *ZL0.13, Lagerstrasse 45, Zürich*

# Design and Implementation of an Alternative to SSH

**Meeting Minutes**

## 1 Attendees

**Present:** Stephan Neuhaus, Raphael Emberger

## 2 Initiation

The meeting took place on the *Friday, 26th of April 2019, 13:00* in *ZL0.13, Lagerstrasse 45, Zürich*. Raphael Emberger was resonsible for the minutes.

## 3 Points of discussion

### 3.1 Login and shell usage

The remote login, starting and usage of a user shell works now. It still does not behave like intended, as there are warnings printed on the screen and every line written gets echoed back, but overall, it works.

### 3.2 Pty echoes stdin back to stdout on client

As described in the point above, when entering shell commands on the shell after remote login, the written lines gets echoed back after hitting enter.

**Solution** The reason this was so, is because the terminal on the client was still in the *cooked* mode rather than the *raw* mode, which behaves differently from the default *cooked* mode, which read line by line and catches and interprets signals like [Ctrl]+[C] or [Ctrl]+[D]. Setting the terminal into *raw* mode should solve the issue as explained in "The Linux Programming Interface".

### 3.3 Transfer of SIGWINCH/ioctl

As described in the first point of discussion, when dropping into the remote shell, there are warnings displayed about a problem with ioctl.

**Solution** In "The Linux Programming Interface" it is also mentioned that making the child the session leader would solve this issue.

### 3.4 Login with test user fails

Trying to login with the test user results in an error when starting the shell because of missing files.

**Solution** This issue was easily solved as the script for setting up the test user was faulty: It didn't properly create the home directory of the test user and since the process which dropped privilege after login didn't have root rights anymore, it couldn't enter the home directory. Therefore, the directory owner and permissions were amended.

### 3.5 Forking abandoned

3.4 of the last meeting suggested using the C-style sockets to pass the file descriptors to the child process, which should solve the issue with forking and still using the net.Conn object. This has been implemented and after some adjustment worked out well.

## 4 Old Business

- **Login attempts in Linux fail** This problem was deemed lower priority, as Login works on the WSL and can still be dealt with in later stages of the project.

## Next Meeting

*Friday, 3th of March 2019, 12:30* in *ZL0.13, Lagerstrasse 45, Zürich*

# Design and Implementation of an Alternative to SSH

**Meeting Minutes**

## 1 Attendees

**Present:** Stephan Neuhaus, Raphael Emberger

## 2 Initiation

The meeting took place on the *Friday, 3th of May 2019, 13:00* in *ZL0.13, Lagerstrasse 45, Zürich*. Raphael Emberger was resonsible for the minutes.

## 3 Points of discussion

### 3.1 Pty echoes stdin back to stdout on client

The suggested solution from last meetings 3.2 of setting the client terminal into raw mode worked out well and has since been implemented like this.

### 3.2 SockAddr

The x-package `unix` provides wrappers for common Unix API calls. Most of the handling with sockets could be transferred to their methods, except `getpeername(3)`, which was implemented in the old way using CGo. The reason being that the returned `SockAddr` interace from `accept(3)` and `getpeername(3)` couldn't be casted(or rather: type-asserted) to be used as `SockAddrInet4` struct.

**Solution** After some experiments and searching the internet, it was revealed that the type assertion had to use pointers instead.

### 3.3 Role of login in the application

Just relying on `login` to perform all the important steps to log a user in is not a viable solution, as this would make log in via keys impossible. Therefore, it was made so the server decides whether to attempt to log in via keys or just to call `login`. Key authentication has been implemented as well and doesn't rely on `login`.

### 3.4 Documentation and bug-fixes before new features

It was deemed better to stop implementing new features as time is about to run out. Therefore, it is suggested to invest 80% of the remaining time to write on the documentation and 20% on bug-fixing. Only after everything has been finished, can new features be implemented - if at all.

### 3.5 Documentation chapter renaming

The original template for the BSc thesis has chapters named ”Theoretical Principles” and ”Method” in them, which will be renamed to the more appropriate ”Design” and ”Implementation”. The latter will not reflect the progress in a chronological order but instead order it point by point.

### 3.6 Login attempts in Linux fail

Relying on `login` finally solved the problem of logging in on a specific linux distribution. The log in via keys also works, as the problem was originally PEM dependent, which gets omitted when authenticating via keys.

## Next Meeting

*Friday, 10th of March 2019, 12:30* in *ZL0.13, Lagerstrasse 45, Zürich*

# Design and Implementation of an Alternative to SSH

**Meeting Minutes**

## 1 Attendees

**Present:** Stephan Neuhaus, Raphael Emberger

## 2 Initiation

The meeting took place on the *Friday, 10th of May 2019, 12:20* in *ZL0.13, Lagerstrasse 45, Zürich*. Raphael Emberger was resonsible for the minutes.

## 3 Points of discussion

### 3.1 File descriptors and Go

After resolving last weeks issue with `SockAddr`(3.2), the same approach enabled us to get the fd of a connection: By type-asserting it as `*net.TCPConn`, which had a function that returns the underlying file, which in turn had a function for returning the underlying fd.

### 3.2 Privilege dropping

After authenticating the user via keys, the host process spawns a shell with the appropriate privileges. For more secure handling, it was deemed best to drop privilege for the entire host process after authenticating via keys. Despite running with root rights(to enable a user to login as any user), the process receives an exception when calling `unix.Setuid` or `unix.Setgid`. The reason was assumed to be related to Go's infamous Go-routines.

### 3.3 Current State of Documentation

The current state of the documentation is still insufficient. There are 4 weeks until the deadline. It is suggested to write more on the documentation.

## Next Meeting

*Friday, 17th of March 2019, 13:00* in *ZL0.13, Lagerstrasse 45, Zürich*

# Design and Implementation of an Alternative to SSH

**Meeting Minutes**

## 1 Attendees

**Present:** Stephan Neuhaus, Raphael Emberger

## 2 Initiation

The meeting took place on the *Friday, 17th of May 2019, 12:30* in *ZL0.13, Lagerstrasse 45, Zürich*. Raphael Emberger was resonsible for the minutes.

## 3 Points of discussion

### 3.1 Referencing Linux Man Pages

As the code base heavily relies on the Linux API, it was decided to include bibtex references to the manual pages to improve readability.

### 3.2 Clarification of Design of Public Key Cryptography

When describing public key cryptography, the current state of the documentation doesn't go into further detail other than describing the flow of actions when authenticating via public key cryptography. This part has to be improved.

### 3.3 Login as Root possible

The current state of the implementation allows a client to authenticate itself and log in as root on the server. This is a point that should be improved, but can be postponed for now.

**Solution** When using `login(1)` to authenticate and log in a user, the `-f` flag can be used to specify a specific user. The client could already ask the user before communication with the server, which user should be used when logging in. This is partially already implemented.

### 3.4 Keys remain in memory

The current state of the implementation of the client still retains the private key in memory when performing key authentication. This could be a possible vulnerability which could be used by a third party as an attack vector to obtain said key.

### 3.5 Keys not stored optimally

The public key up to today stored the full public key inside a directory structure in the root user's home directory. This could cause problems when storing keys and should be changed to the way `ssh` stores authorized public keys: By storing the authorized keys for a server-side user in hashed format in its home directory.

### 3.6 RSync not implemented yet

The layout of the tasks mentions `rsync` compatibility as both a required use-case for a passing grade and an optional extra feature. This has been amended by Mr Neuhaus by stating explicitly, that `rsync` compatibility is optional.

### 3.7 Separate bibliography for man page references

As references to the Linux manual pages take up a considerable part of the overall references, it was considered to split it up into a manual-only bibliography and a normal bibliography. This suggestion was rejected as it was deemed tolerable to have one single bibliography.

### 3.8 Picture for the Publication Tool

The official Publication Tool requires a picture to be attached to the abstract when handing it in to the online tool. As this thesis is centered around a CLI application, no picture has been made so far. It was suggested to simply take a screenshot of the application in operation.

## Next Meeting

The next meeting doesn't have a set date, time or place, as it was deemed a better option to organize a new meeting whenever the need for one arises.

# Design and Implementation of an Alternative to SSH
**Meeting Minutes**

## 1 Attendees

**Present:** Stephan Neuhaus, Raphael Emberger

## 2 Initiation

The meeting took place on the *Friday, 31th of May 2019, 12:30* in *ZL0.13, Lagerstrasse 45, Zürich*. Raphael Emberger was responsible for the minutes.

## 3 Points of discussion

### 3.1 Time Issues

Because of the lack of participation from a peer, Raphael was occupied with the labs of the AI2 course for over two days, which led to more pressure in finishing the thesis.

### 3.2 Project stopped working

When connecting to the server, the host application cannot open the connection from the file descriptor and throws a "Bad file-descriptor" error. This happens on the WSL.

Using a native Linux distribution solved this issue.

### 3.3 Formal Requirements

It was stressed to find out more about the formal requirements, lest the whole thesis might face tribulations due to formal details.

### 3.4 Attendance of Mr. Schwarz at the Presentation

It was suggested to include Mr. Schwarz in the presentation of the thesis as he was part of the team for a considerable amount of time. He has no obligation to present himself but may attend the presentation itself.

## Next Meeting

There will not be any further meetings.

## A.2. Others

Please refer to the USB-stick that has been handed in with this thesis.