Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

Bachelor's thesis
HS16 Studiengang Informatik

Design and Implementation of an Alternative to SSH

Authors Raphael Emberger, Kal-El,
Musashi Miyamoto

Date May 19, 2019

Abstract

Preface

The Secure Shell(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 Zurich University of Applied Sciences(ZHAW) for granting me the opportunity to do my Bachelors thesis here and Dr. Stephan Neuhaus for helping me along the way of this Bachelors thesis.

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

1.1. Initial Position

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.

1.1.1. 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 1.1.2) and *rlogin(1)* (1999)/*rsh(1)* (1999)(see 1.1.3).
SSH uses Transport Layer Security(TLS) 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.
SSHs 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 login 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 Pluggable Authentication Modules(PAMs). Another feature is whitelisting of clients via their public keys, which barres intrusion 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.

1.1.2. 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

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

1.1.3. 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` spawns a shell without the log in process.
- *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.

1.2. Task

The official formulation of the tasks can be found in the appendix A.1.1.
The objective of this thesis is as follows:

- Design and implementation of a client-server protocol that can manage interactive sessions.
- Design and implementation of 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.
- In the software, a privilege-separation architecture.

Incorporating the following components will improve the grade:

- 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`.

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 6.5.

2. Design

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 lead to a few problems in the implementation process(See 3.1).

2.2. Authentication via PAM

For authentication of users on Linux systems, PAM exists. I provides a clean separation between a program and the sensitive part of authentication. PAM operates using transactions, which represent a link to a PAM context. Using this transaction, an application can do various actions regarding account management, authentication or session management. To initialize such a context and 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 context

Similarly, the created context 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 context

After creating a context, 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).

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

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 2.4.1). To finish, the application terminates the context properly and all associated memory gets invalidated.

2.3. 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.

2.4. Post-Login Actions

When a user logs into a Unix machine, a session and a timestamp 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
```

```c
#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 2.6: 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 `ùtmpx` 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 login, 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 getters.

```c
#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 2.7: `utmpx` API functions

2.4.1. 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):

```c
#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 2.8: PAM session management

2.5. 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.

2.6. Pseudoterminal

pseudoterminals(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 2.1).
A shell expects to be connected to a full fledged teletype(tty)(or a tty emulator). To check whether it is inside such an environment, it uses *isatty(3)* (2019) on the file descriptor(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

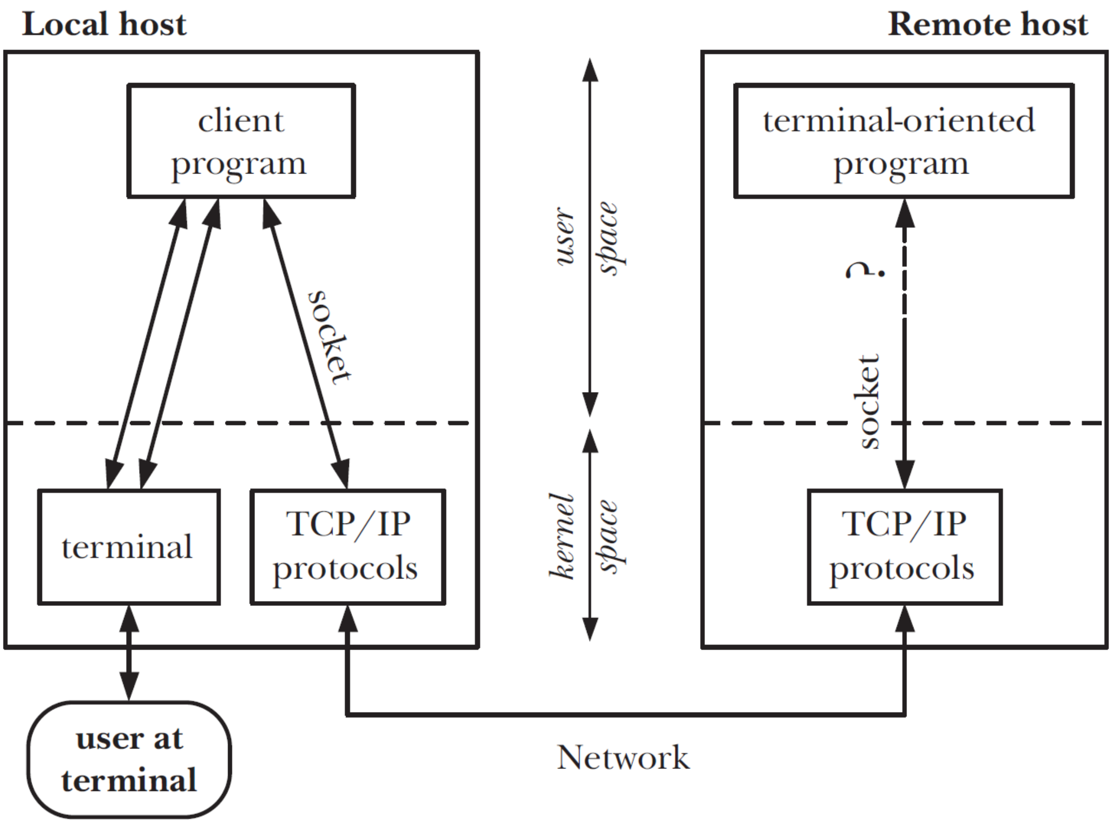


Figure 2.1.: How to operate a tty-oriented program over a network?(Kerrisk 2010, p.1376)

pseudoterminal slave(pts)(see figure 2.2). 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):

```c
#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 2.9: pty related Linux API functions

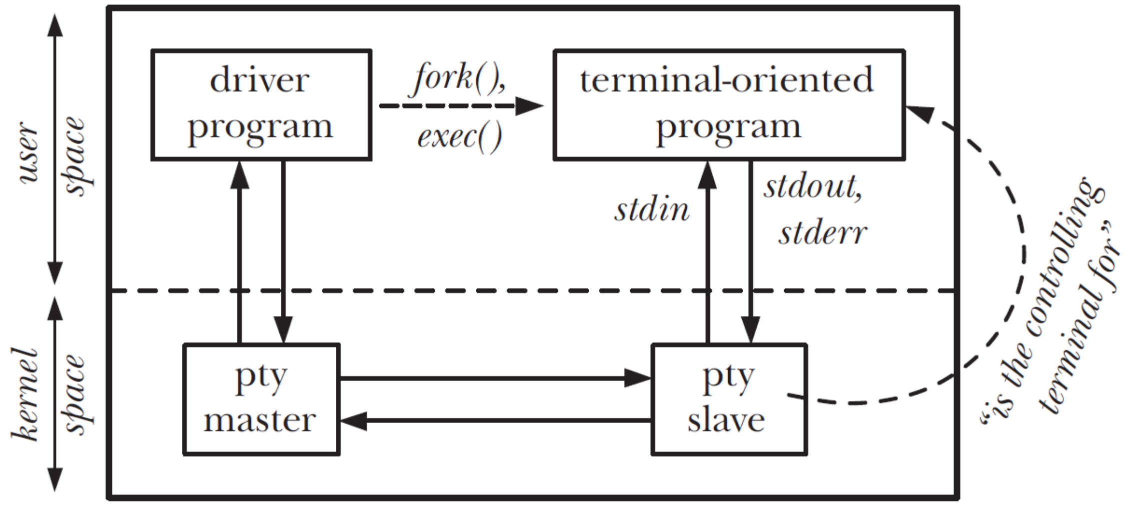


Figure 2.2.: Two programs communicating via a pty(Kerrisk 2010, p.1377)

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 2.3.

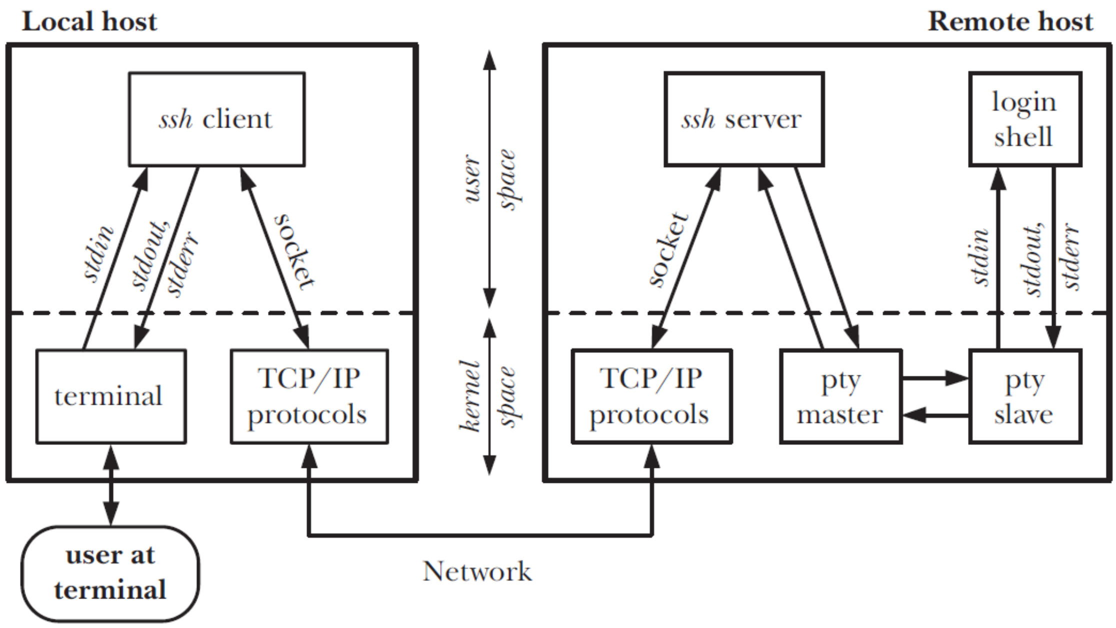


Figure 2.3.: How ssh uses a pty(Kerrisk 2010, p.1378)

2.7. Flow of Action

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

1. Server listens for incoming connection.
2. Client dials server.
3. Server spawns the users login shell and forwards all traffic between shell and Client.
4. Client uses shell on remote machine.
5. Client terminates session.
6. Host terminates.

However, there are multiple security concerns to be satiated:

1. The Client must authenticate itself for a user of the remote system with the appropriate credentials.
2. The Server has to drop privilege after a successful login to prevent privilege escalation.
3. The Server has to spawn the login shell of the logged in user with the appropriate rights.

Furthermore, the current design does not allow for multiple sessions to be run parallel. Therefore it was decided to spawn a new process to handle everything beginning after the connection has been established.
This lead to the following general flow of actions:

1. Server listens for incoming connection.
2. Client dials server.
3. Server spawns a Host upon established connection.
4. Host sets up connection.
5. Host asks Client to authenticate himself.
6. Client authenticates himself.
7. Host drops privilege to logged in user.
8. Host spawns the users login shell with the same credentials and forwards all traffic between shell and Client.
9. Client uses shell on remote machine.
10. Client terminates session.
11. Host terminates.

2.7.1. Old Design

An earlier draft of the flow of actions was designed as a sequence diagram and can be seen in figure 2.4.

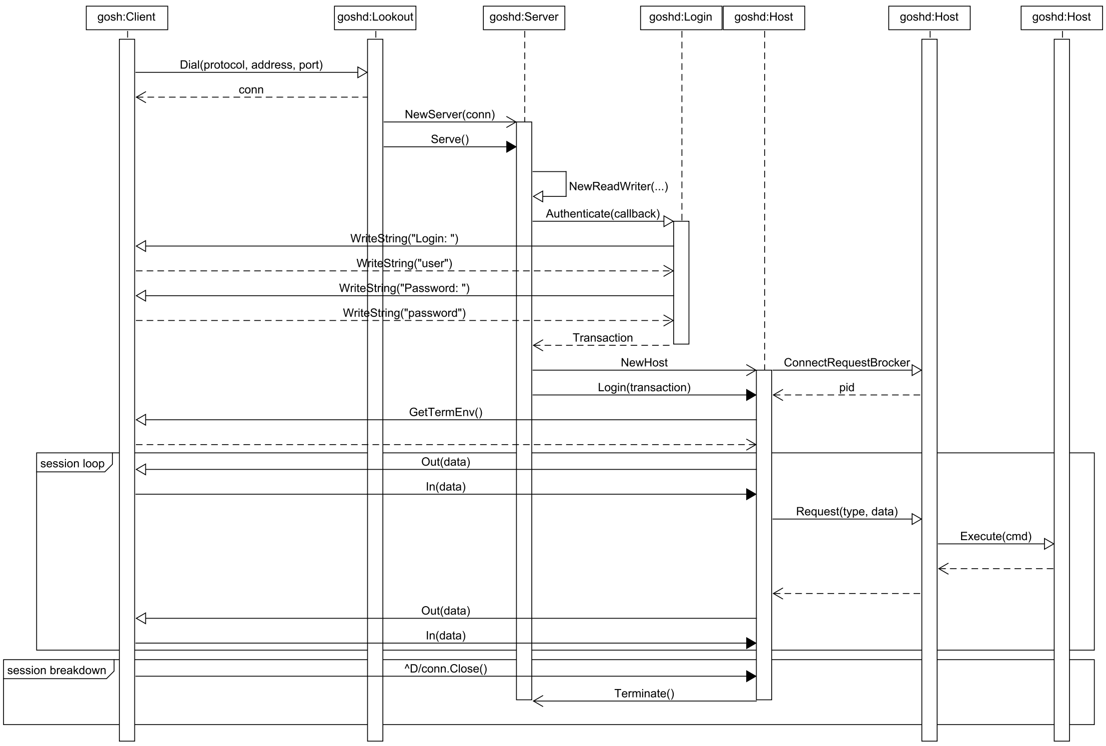


Figure 2.4.: Sequence diagram draft.

This design used a Request Broker, which would put another master-slave-relationship between a shell and the Request Broker. This would have been an over-complication of the use of a shell, which, if set correctly, already runs only with the rights of the corresponding user and its groups.

2.7.2. New Design

The new and current design of the architecture can be seen in figure 2.5.
This new design of the `goshd` server calls the external binary `goshh` from the server process and lets it handle the connection itself. The server then asks for some environment variables from the client. Then the server tries to figure out whether the client can authenticate himself via keys. If authentication with keys is a viable option, it performs the key authentication according to 2.3. If it succeeds, it notifies the client of the authentication result. The host then spawns the user's shell and forwards the traffic between the two.
If key authentication does not succeed, the host performs the classical authentication with user-password-credentials which also ends in spawning a user shell if successful.
If the shell exits or the host process receives an interrupt signal(SIGINT), it cleans up behind the exited shell and exits as well.
The client performs the transfer of the environment variables until it receives the termination packet. Upon arrival of said packet, it prepares for the forwarding of the traffic between the user and the server. It does so by setting the terminal in raw-mode and waits until the connections dies(it receives `EOF`) and then restores the terminal to the default cooked state.

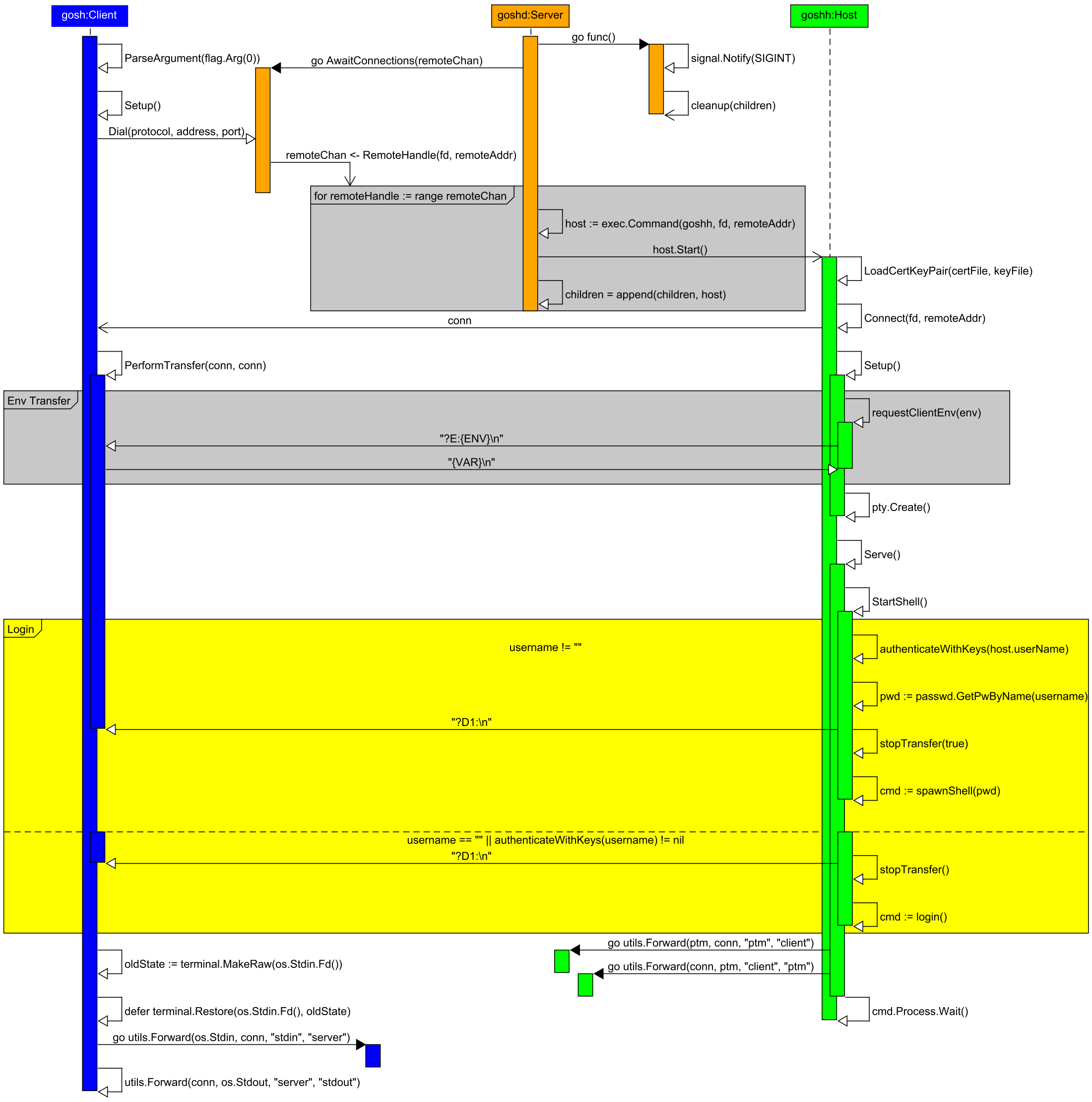


Figure 2.5.: Sequence diagram of current implementation.

3. Implementation

3.1. Problems

3.1.1. Forking

To handle new established connections, it was deemed important to `fork` 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 Gos standard library capabilities to handle connections. However, there were several problems with this approach:

Forking not supported

The Go standard library does not support the classical C-like forking. It only has a `syscall.ForkExec` method, which is documented as:

> Combination of fork and exec, careful to be thread safe.

But since it uses `exec` 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)`.

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 lead 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 2 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 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:

```go
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:

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

```go
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: Gos 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: 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.

4. Results

5. Discussion And Prospects

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

grantpt(3) (2017).
URL: `http://man7.org/linux/man-pages/man3/grantpt.3.html` 11

isatty(3) (2019).
URL: `http://man7.org/linux/man-pages/man3/isatty.3.html` 11

Kerrisk, M. (2010), *The Linux Programming Interface*, No Starch Press, San Francisco, CA, USA.
URL: `http://www.man7.org/tlpi/` 10, 11, 12, 25, 27

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` 6

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

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

pam_authenticate(3) (2016).
URL: `http://man7.org/linux/man-pages/man3/pam_authenticate.3.html` 9

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

pam_end(3) (2016).
URL: `http://man7.org/linux/man-pages/man3/pam_end.3.html` 8

pam_get_item(3) (2016).
URL: http://man7.org/linux/man-pages/man3/pam_get_item.3.html 8

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

pam_setcred(3) (2016).
URL: http://man7.org/linux/man-pages/man3/pam_setcred.3.html 9

pam_set_item(3) (2016).
URL: http://man7.org/linux/man-pages/man3/pam_set_item.3.html 8

pam_start(3) (2016).
URL: http://man7.org/linux/man-pages/man3/pam_start.3.html 8, 9

pam_strerror(3) (2016).
URL: http://man7.org/linux/man-pages/man3/pam_strerror.3.html 8

posix_openpt(3) (2017).
URL: http://man7.org/linux/man-pages/man3/posix_openpt.3.html 11

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

pts(4) (2013).
URL: http://man7.org/linux/man-pages/man4/pts.4.html 23

ptsname(3) (2017).
URL: http://man7.org/linux/man-pages/man3/ptsname.3.html 11

pty(7) (2017).
URL: http://man7.org/linux/man-pages/man7/pty.7.html 23

pututxline(3) (2017).
URL: http://man7.org/linux/man-pages/man3/pututxline.3.html 10

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

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

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

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

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

unlockpt(3) (2017).
URL: http://man7.org/linux/man-pages/man3/unlockpt.3.html 11

6.2. Glossary

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

CGo C support for Go. 16

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

file descriptor
A file descriptor is an integer that represents the handle to a file. 11

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

Inter-Process-Communication
Communication between processes. 11

interrupt signal
A signal supported by Unix based Operating Systems(OSes) which signals to a process to interrupt it's work. 14

Linux A Unix based OS which uses Linus Torvalds kernel and was inspired by Minix. 11, 27

Object Oriented Programming
A programming paradigm which uses objects to model real life entities. 24

Pluggable Authentication Module
Modules for user authentication. 6

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, 24

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). 11

pseudoterminal master
The master of a pty. 11

pseudoterminal slave
The slave of a pty(*pts(4)* 2013). 11

Request Broker A service that oversees the action requests of a program and decides whether to permit and execute them or not, based on various factors. 14

Secure Shell
An client-server-application that allows remote login and interaction with a shell. See 1.1.1. 3

Secure Sockets Layer
Cryptographic protocol to secure the communication between two peers via symmetric cryptography. Deprecated. 24

shell A CLI program, that reads user input line-by-line and executes those commands. 6, 7, 10–14, 23, 24

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). 11

Telnet Secure
Telnet with Secure Sockets Layer(SSL) encryption. 6

terminal An user interface to interact with CLI programs like shells. 6, 10–12, 14, 24

the C programming language
Low-level programming language originally invented by Dennis Ritchie. 8

the C++ programming language
Descendant of C which implemented Object Oriented Programming(OOP). 8

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

Transport Layer Security
Newer and recommended version of SSL. 6

Unix An originally free OS family called Unics from AT&T that re-imagined an older OS by the name of Multics. 9, 23

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

Zurich University of Applied Sciences
Name of my university of trust. 3

6.3. List of Figures

6.4. List of Tables

6.5. List of Listings

6.6. Acronym Glossary

API *Application Programming Interface* 6, 9–11, 17, 27, *See* Application Programming Interface

C *the C programming language* 8, 16, 23, 24, *See* the C programming language

C++ *the C++ programming language* 8, *See* the C++ programming language

CLI *Command Line Interface* 3, 24, *See* Command Line Interface

fd *file descriptor* 11, 16, 17, 27, *See* file descriptor

Go *the Go/Golang programming language* 3, 6, 8, 16, 17, 23, 24, 27, *See* the Go/Golang programming language

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

IPC *Inter-Process-Communication* 11, *See* Inter-Process-Communication

OOP *Object Oriented Programming* 24, *See* Object Oriented Programming

OS *Operating System* 23, 24

PAM *Pluggable Authentication Module* 6, 8–10, 27, *See* Pluggable Authentication Module

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

ptm *pseudoterminal master* 11, *See* pseudoterminal master

pts *pseudoterminal slave* 11, 12, *See* pseudoterminal slave

pty *pseudoterminal* 11, 12, 23, 25, 27, *See* pseudoterminal

SIGINT *interrupt signal* 14, *See* interrupt signal

SSH *Secure Shell* 3, 6, 12, *See* Secure Shell

SSL *Secure Sockets Layer* 24, *See* Secure Sockets Layer

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

TLS *Transport Layer Security* 6, 7, *See* Transport Layer Security

tty *teletype* 11, 23–25, *See* teletype

ZHAW *Zurich University of Applied Sciences* 3, *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 by showing that it 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.

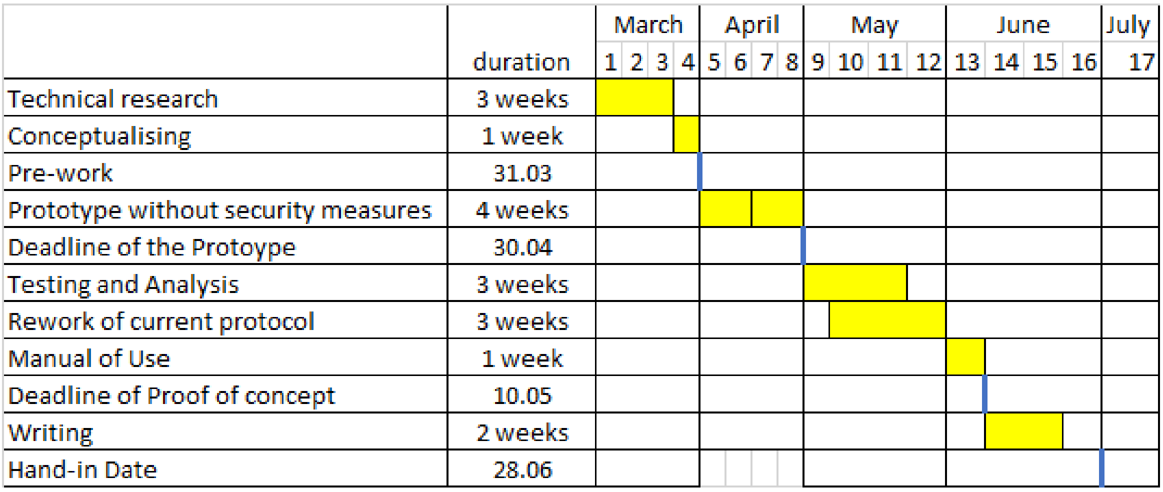
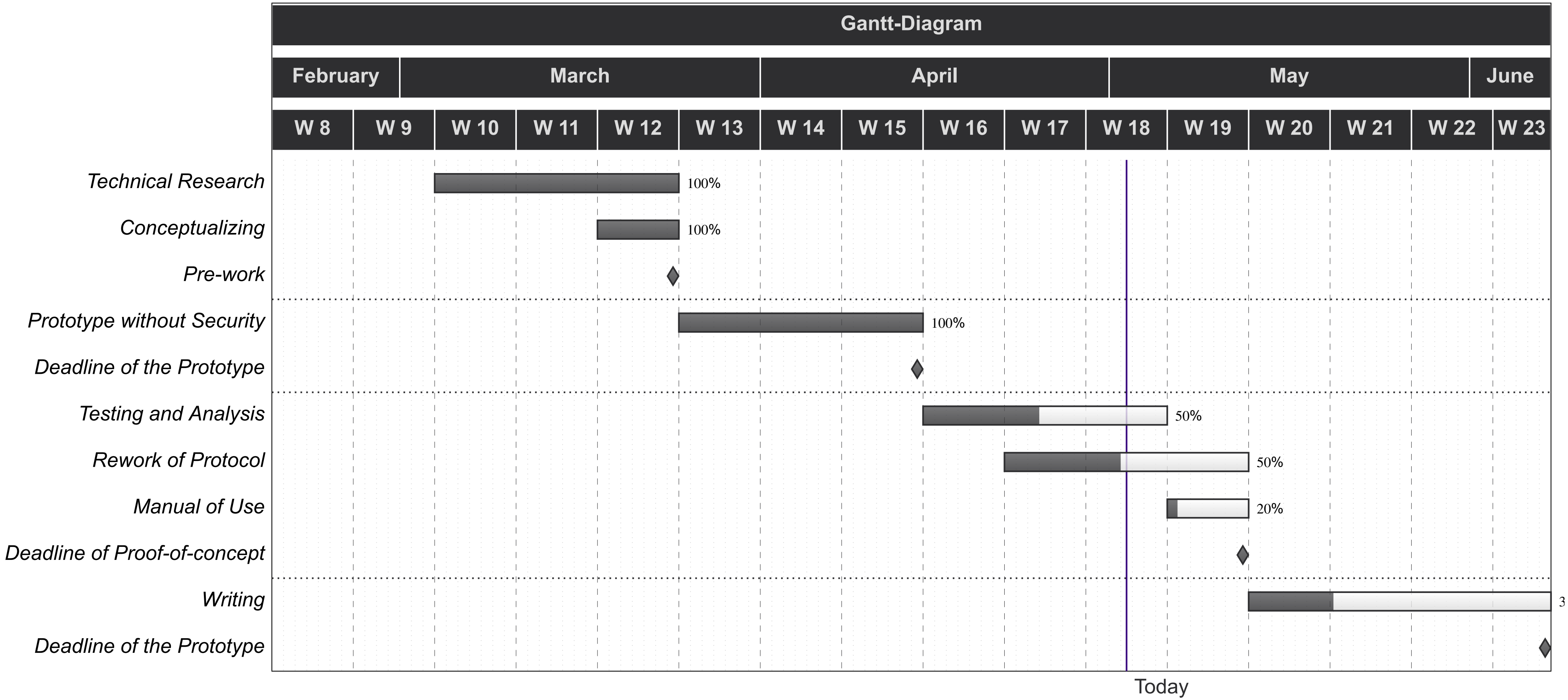


Figure 1: Project plan


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
50%
Rework of Protocol
50%
Manual of Use
20%
Deadline of Proof-of-concept
Writing
3
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

Participent
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.

A.2. Others