CSAP project 2022/2023

Davide Brian Di Campi, 1754338

1 Introduction

The assignment required to develop a client-server application, which allows a client to manage files and execute commands on a remote system. The client had to be implemented as a daemon, running in background, listening on a TCP socket. It could be developed either as a separated set of programs, or as a monolithic app implementing all the required features. My personal choice was the latter.

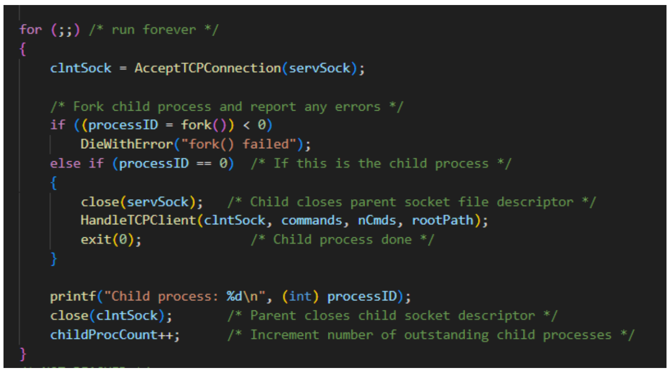
2 Design choices

I chose to develop the project as a monolithic application, so there are only two executables: the client and the server.
Describing the general behaviour, the server is listening on a TCP port, waiting for incoming connections from clients. When a connection is accepted, the server fork(), creating another process that will handle the new connection with the client. After an authentication process, needed to execute the commands with the remote user privileges, it will listen forever for commands from the client.
Commands are executed with a fork-execve paradigm. Whenever a command has to be executed, the process forks: the child will execve() the given command, while the father will wait for the child to end, collect its output and exit status (using pipes) and then sending back the response to the client. Typing "exit" will close the connection and end the process.

3 Important requirements

There were many important requirements to be kept in mind while developing the application:

- The server must allow for multiple concurrent connections. As stated in the design choices, this was implemented using processes: whenever a connection is accepted, a new process is created as a child of the server and it will handle the interaction with the client (Figure 1).

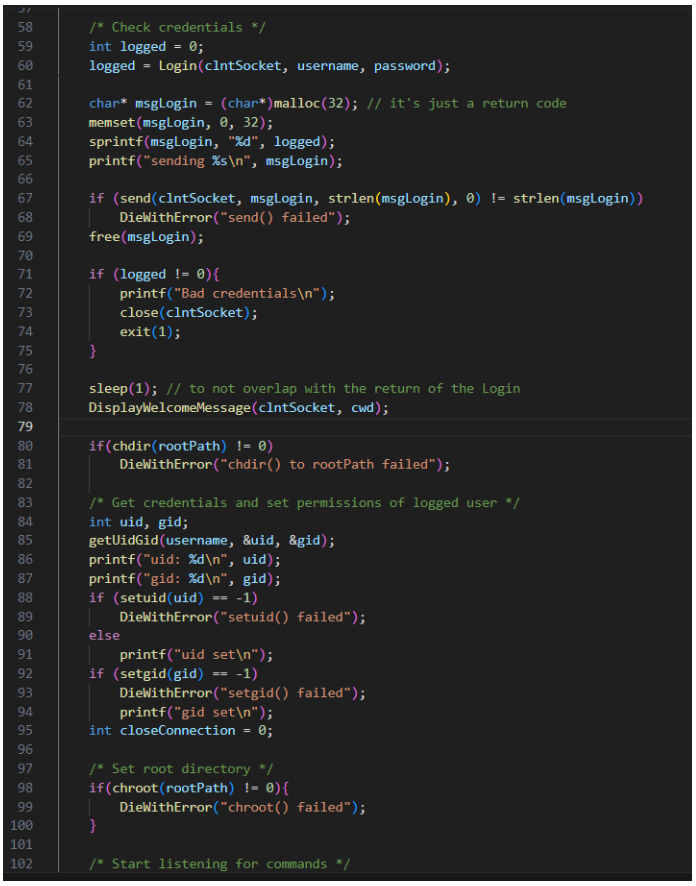


```c
for (;;) /* run forever */
{
    clntSock = AcceptTCPConnection(servSock);

    /* Fork child process and report any errors */
    if ((processID = fork()) < 0)
        DieWithError("fork() failed");
    else if (processID == 0)  /* If this is the child process */
    {
        close(servSock);   /* Child closes parent socket file descriptor */
        HandleTCPClient(clntSock, commands, nCmds, rootPath);
        exit(0);              /* Child process done */
    }

    printf("Child process: %d\n", (int) processID);
    close(clntSock);       /* Parent closes child socket descriptor */
    childProcCount++;      /* Increment number of outstanding child processes */
}
```

Figura 1: Server listening for multiple concurrent connections

- Actions must be performed with credentials (uid, gid) of the user which executed the command. This is done with an authentication process: when the client connects to the server, it sends username and password of the user. If the user is on the server and the password is correct, the program retrieve uid and gid of that user and performs setuid() and setgid(), thus the commands will be executed with those privileges (Figure 2).

- Navigation between directories must be limited to subtree contained under the a pseudo-root, not following symlinks. This part was implemented using chroot(). As requested, the pseudo-root folder is specified in a configuration file and used as argument of a chroot() call, performed right after the login and the privilege setting (Figure 2). The catch of using chroot are that I had to write a bash script for copying the needed executables inside the pseudo-root folder and that I had to set the capabilities to the server executable (done in the Makefile).



```c
/* Check credentials */
int logged = 0;
logged = Login(clntSocket, username, password);

char* msgLogin = (char*)malloc(32); // it's just a return code
memset(msgLogin, 0, 32);
sprintf(msgLogin, "%d", logged);
printf("sending %s\n", msgLogin);

if (send(clntSocket, msgLogin, strlen(msgLogin), 0) != strlen(msgLogin))
    DieWithError("send() failed");
free(msgLogin);

if (logged != 0){
    printf("Bad credentials\n");
    close(clntSocket);
    exit(1);
}

sleep(1); // to not overlap with the return of the Login
DisplayWelcomeMessage(clntSocket, cwd);

if(chdir(rootPath) != 0)
    DieWithError("chdir() to rootPath failed");

/* Get credentials and set permissions of logged user */
int uid, gid;
getUidGid(username, &uid, &gid);
printf("uid: %d\n", uid);
printf("gid: %d\n", gid);
if (setuid(uid) == -1)
    DieWithError("setuid() failed");
else
    printf("uid set\n");
if (setgid(gid) == -1)
    DieWithError("setgid() failed");
    printf("gid set\n");
int closeConnection = 0;

/* Set root directory */
if(chroot(rootPath) != 0){
    DieWithError("chroot() failed");
}

/* Start listening for commands */
```

Figura 2: Procedures before handling commands

- Access to files must be possible maintaining consistency. This was implemented using semaphores. First, a sem_init() initialize the mutex, then sem_wait() is called before handling the command. Once done, a call to sem_post() will unlock the resources. Finally, when closing the connection a call to sem_close() will release the semaphore before exiting.

.

4 Main modules

Here is a brief description of the main modules and what they do.

- client : this is the client module. Its only source code is client.c. The syntax to start it is
  ./client <Server IP> <Port>
  After initializing the connection, it requests the credential to send to the server for the authentication process. If the login is successful, it listens forever for commands from STDIN, receiving and displaying the response from the server, until "exit" is typed.
  It is also compiled with DieWithError.c module, that is just a custom wrapper around perror() function.

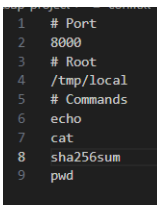
- server : this is the server module. The syntax to start it is
  ./server <CONFIG_FILE>
  As already described, it listens for incoming connections from clients. It is composed of the following .c modules:

- server.c : parse the configuration file, initialize the socket and wait for connections. Has also a ChildExitSignalHandler() function to handle zombie children.
- CreateTCPServerSocket.c : initialize the socket, bind it to the port and mark it to listen for incoming connections, setting the address to be reused.
- AcceptTCPConnection.c : accept incoming connections.
- HandleTCPClient.c : this handles the connection with a client. It performs the authentication, chroot() and handles the commands from the client, maintaining consistency between calls.
- DieWithError.c, already described above

.

Apart from the c source code files, there are other files in the project:

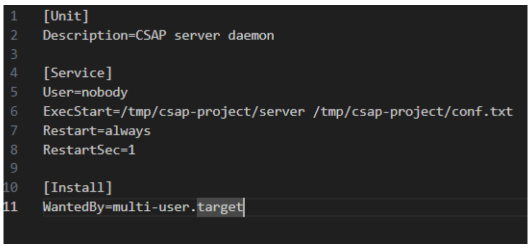
- conf.txt : this is the configuration file. It stores the TCP port of the server, the pseudo-root path and the additional commands allowed (Figure 3).



```
# Port
8000
# Root
/tmp/local
# Commands
echo
cat
sha256sum
pwd
```

Figura 3: Configuration file

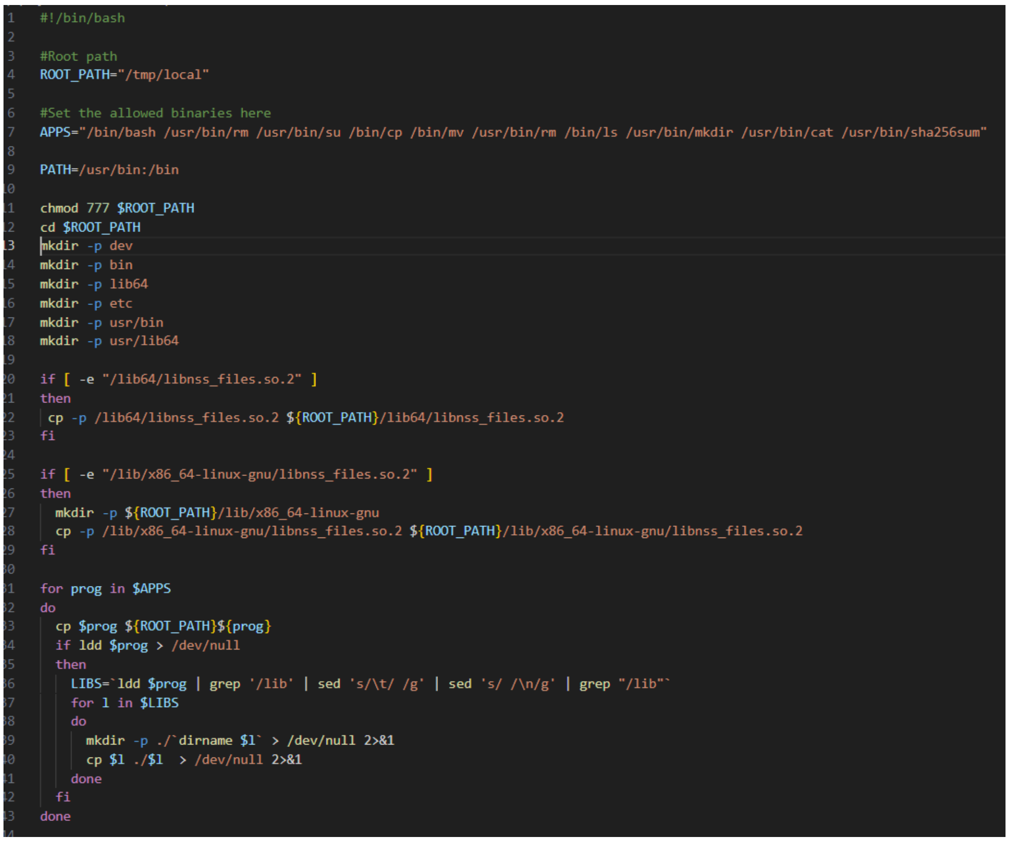
- daemon.service : configuration of the service, needed to run the server as a daemon with systemd (Figure 4).



```
[Unit]
Description=CSAP server daemon

[Service]
User=nobody
ExecStart=/tmp/csap-project/server /tmp/csap-project/conf.txt
Restart=always
RestartSec=1

[Install]
WantedBy=multi-user.target
```

Figura 4: Daemon.service file

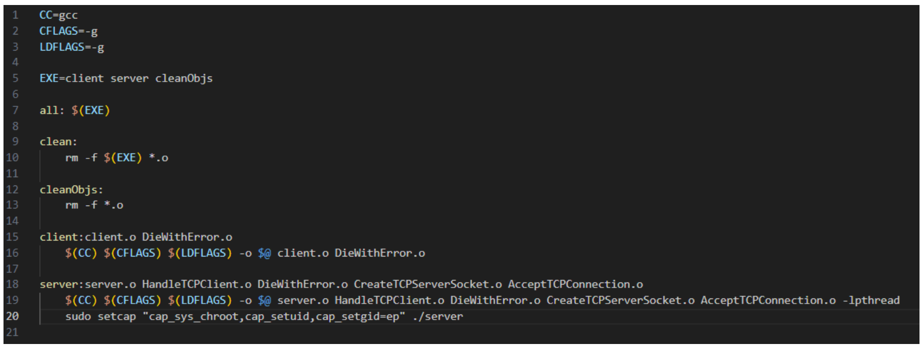
- chroot_setup.sh : a custom bash script that is needed to initialize the pseudo-root folder, otherwise when using chroot() no binaries will be found (Figure 5).



```bash
#!/bin/bash

#Root path
ROOT_PATH="/tmp/local"

#Set the allowed binaries here
APPS="/bin/bash /usr/bin/rm /usr/bin/su /bin/cp /bin/mv /usr/bin/rm /bin/ls /usr/bin/mkdir /usr/bin/cat /usr/bin/sha256sum"

PATH=/usr/bin:/bin

chmod 777 $ROOT_PATH
cd $ROOT_PATH
mkdir -p dev
mkdir -p bin
mkdir -p lib64
mkdir -p etc
mkdir -p usr/bin
mkdir -p usr/lib64

if [ -e "/lib64/libnss_files.so.2" ]
then
 cp -p /lib64/libnss_files.so.2 ${ROOT_PATH}/lib64/libnss_files.so.2
fi

if [ -e "/lib/x86_64-linux-gnu/libnss_files.so.2" ]
then
  mkdir -p ${ROOT_PATH}/lib/x86_64-linux-gnu
  cp -p /lib/x86_64-linux-gnu/libnss_files.so.2 ${ROOT_PATH}/lib/x86_64-linux-gnu/libnss_files.so.2
fi

for prog in $APPS
do
  cp $prog ${ROOT_PATH}${prog}
  if ldd $prog > /dev/null
  then
    LIBS=`ldd $prog | grep '/lib' | sed 's/\t/ /g' | sed 's/ /\n/g' | grep "/lib"`
    for l in $LIBS
    do
      mkdir -p ./`dirname $l` > /dev/null 2>&1
      cp $l ./$l  > /dev/null 2>&1
    done
  fi
done
```

Figura 5: Setup script for pseudo-root folder

- Makefile : this is used to build all the modules, and to set the capabilities on the server executable (Figure 6).



```makefile
CC=gcc
CFLAGS=-g
LDFLAGS=-g

EXE=client server cleanObjs

all: $(EXE)

clean:
	rm -f $(EXE) *.o

cleanObjs:
	rm -f *.o

client:client.o DieWithError.o
	$(CC) $(CFLAGS) $(LDFLAGS) -o $@ client.o DieWithError.o

server:server.o HandleTCPClient.o DieWithError.o CreateTCPServerSocket.o AcceptTCPConnection.o
	$(CC) $(CFLAGS) $(LDFLAGS) -o $@ server.o HandleTCPClient.o DieWithError.o CreateTCPServerSocket.o AcceptTCPConnection.o -lpthread
	sudo setcap "cap_sys_chroot,cap_setuid,cap_setgid=ep" ./server
```

Figura 6: Makefile

5 Commands and syntax

The execution of commands is done using execve, then the output and exit status are collected and used to make the response to the client. In most cases, STDOUT and STDERR of the execution are directly reported to the client, so the error handling and output is delegated to the shell.
The following commands are implemented directly on the server:

- **copy <src> <dest>** : copy file <src> into file <dest>. On success returns "copy successful", or the error otherwise.
- **move <src> <dest>** : move file <src> into file <dest>. On success returns "move successful", or the error otherwise.
- **delete <file>** : delete file <file>. On success returns "delete successful", or the error otherwise.

- **list** : list all files in the current directory, performing a "ls -l". On success returns the output, or the error otherwise.
- **create_dir <dirname>** : create the directory <dirname>. On success returns "create_dir successful", or the error otherwise.
- **delete_dir <dirname>** : delete the directory <dirname>, only if it's empty. On success returns "delete_dir successful", or the error otherwise.
- **cd <path>** : change the current directory to <path>. On success returns a message with the new current path, or the error otherwise.
- **run <cmd>** : run command <cmd> if it is in the allowed list in the configuration file. On success returns a message with the output, or the error otherwise.
- **run cmd > file** : run command <cmd> if it is in the allowed list in the configuration file, and write the output on client file <file>. On success returns a message, or the error otherwise.
- **run "cmd1 | cmd2 > file"** : run command <cmd1>, pass its output to <cmd2> and write the final output on server file <file>. The pipe is not limited to one. Every command must be on the allowed list in the configuration file. On success returns a message, or the error otherwise. Can return syntax error.
- **run "cmd1 | cmd2" > file** : run command <cmd1>, pass its output to <cmd2> and write the final output on client file <file>. The pipe is not limited to one. Every command must be on the allowed list in the configuration file. On success returns a message, or the error otherwise. Can return syntax error.

6 Known bugs and conclusions

The implementation of this application have surely security issues. For example there may be possible buffer overflows. Another problem is that using chroot() does not ensure that the user cannot escape it, for example creating and/or moving executables, or maybe even with cd. The <run> command parser I wrote has probably some issues, I didn't test for every possible malformed syntax, but in the end every command (not only <run>) is executing with execve() and the error handling is mostly left to the shell itself.
This project was a challenge to me, not having programmed that much in C language, especially for the strings handling, but I'm pretty satisfied of the final result. Certainly, it was an interesting project which brought me much more experience than I had before.