Secure Programming 2021/2022
Assignment 1: SecChat

1 Introduction

For this assignment, we expect you to design and build a secure chat application in C that runs on Linux. This application consists of two programs: a client program and a server program. This document describes the requirements for these programs. We highly recommend reading the entire document before starting the design and programming of your application to ensure you meet all the requirements.

Learning objectives of this assignment include:

- learn to design a secure distributed application;
- learning to think about where and how to apply various forms of cryptography in practice;
- practice applying secure programming guidelines;
- gain more experience with explicit memory management, which is often an important source of vulnerabilities when done incorrectly (for this reason, the assignment is specifically in C);
- practice using a real-world cryptographic library.

Note that this is a substantial programming project that will require knowledge you acquire during the lectures. Do not wait for each topic to be discussed in the lectures but start off with the parts you can already do based on your prior knowledge. Do, however, make sure you check and adjust the design and the code later if it turns out it should have done been in a better way. Also, be sure to ask for help if you're stuck. You are here to learn, and we are here to help you achieve that. You can always reach us at sp@vusec.net.

2 Group work

The assignment is made in groups with a maximum of three students per group. Smaller groups are allowed, but group size is not considered for grading or for the amount of work that needs to be done. You can use the Canvas discussion board to find group members. Note that all members are fully responsible for the whole of the assignment even if the work is divided. Be sure to verify that your teammates' work is up to your standards, and set internal deadlines to ensure you have enough time to intervene if you find out it is not. If any of your group members provides an insufficient contribution to the final code, please let us know as soon as possible at sp@vusec.net.

Given that you will be collaborating on a substantial programming assignment, we strongly recommend that you use a source control system, such as for example git. This allows you to conveniently store and merge different versions of your program, and is an efficient way to send updates to each other. If you have not done so before, it is also a valuable skill to learn. However, if you use a public host (such as Github) be sure to make your repository private to prevent other students from stealing your code.

3 Compiling and running

The program must be able to compile and run on a standard installation of Ubuntu 20.04 LTS Desktop for the x86_64 architecture. You may assume that the `libssl-dev` and `libsqlite3-dev` APT packages are installed, as well as `build-essential` (including tools such as GCC and make). All code must be written in the C language (not C++), and be able to be compiled by GCC 9.3.

Running `make all` in your application's root directory must compile both the `client` and the `server` programs, creating them in the application's root directory. If the program needs any cryptographic keys and/or certificates they can also be generated, into the `serverkeys`, `clientkeys`, and/or `ttpkeys` directories. You may write a shell script or python3 script to achieve this (to be called from make), and they are allowed to use the OpenSSL command line utilities. Running `make clean` must delete any compilation output (including the cryptographic keys/certificates), as well as the `chat.db` file (if it exists).

After building, the server is started by invoking the program `server` in the application's root directory with a single argument, the TCP port number for the server to listen on. The server keeps running until explicitly terminated with CTRL+C or the `kill` command. Any number of clients can connect to the server by invoking the `client` program, also in the same directory. It takes two parameters, namely the hostname and TCP port number where the server is running. For example, the following instructions should be enough to compile everything, run a server on TCP port 1234, and connect a single local client to it:

```
make all
./server 1234 &
./client localhost 1234
```

Please test your code carefully, as failure to compile or run may result in failing the assignment, and points are deducted for compiler warnings. Be sure to use the `-Wall -Werror` compiler switches do have the compiler help point out potential issues.

4 Getting started

In the assignment tarball, you can find a directory `framework`, which contain an example implementation of the framework handling network setup and process management. This code uses the `select`-based design described in Section B. You are allowed and recommended to use this code as a starting point for your application, but this is not required.

5 Functional requirements

The client must be capable of the following tasks:

- The user can register a new account. To do so, they will have to supply a username and a password.
  - You are allowed (but not required) to set a maximum length for the username and/or the password, as long as it is no less than 8 characters.
- The application prohibits registration of a user with a previously registered username.
- The user can login. It is only possible to login to an account if one knows the password supplied at registration time.
- The user can exit. This logs out from the server and terminates the client program.
- When the client starts, it displays all public messages previously sent by anyone, and all private messages received and sent by the current user, in chronological order from old to new.
- The user can send a public message to all users.
- The user can send a private message to a specific user.
- You are allowed (but not required) to set a maximum length for messages, as long as it is no less than 140 characters.
- Each message is shown together with a timestamp, the user who sent it, and for private messages also the recipient.
- The clients only show (1) public messages and (2) private messages of which the logged in user is either the sender or the recipient.
- Each message will be shown to its recipient(s) immediately (or at least, as immediate as network latency will allow).
- The client provides a list of logged in users on request.

The server program implements a protocol to communicate with the client, supporting all the functionality needed to provide these client features. The server program takes care of all necessary storage, at least to the extent that security requirements allow. The server may limit the number of simultaneous connections, but this limit must be no less that 10.

Note that it may be necessary to implement additional user and/or message metadata, interface elements, and protocol features to ensure the security of the application as specified in Section 7. In particular, think carefully about where and how you need to use cryptography, and how to manage cryptographic keys.

5.1 Extra functionality

For most students, implementing these functionalities while simultaneously meeting all the other requirements is a significant challenge. If you are not one of those students, and you completed the project with ease, you may implement extra functionality to receive up to 1.0 point bonus on your grade. You can only get this bonus if the remainder of the assignment is good enough to receive a grade of at least 5.5.

The extra functionality to be implemented is a simple HTTP server inside the `server` program which provides web chat equivalent in functionality to the native `client` program. HTTP 1.0 should suffice, and is defined in RFC 1945. You do not need a full HTTP protocol implementation, just enough to make this particular application work in modern browsers (specifically, it must work on the latest version of the Firefox browser).

To use web chat, you start the `server` program as normal, but it now recognizes if an incoming connection is HTTP and services it appropriately (note: you are allowed to have your regular client also use an HTTP-based protocol). In this case, it serves an HTML page if the path `/` is requested that contains the web chat client. You may serve additional files as well, such as script files, style sheets, and images. The web chat client offers the same functionality as the regular client. However, it is allowed to have a small delay in message delivery, allowing you to use polling (regularly asking the server whether there is a new message). Implementing web sockets that can deliver the data immediately would solve this issue, but seems too much to ask. A common way to implement a web client would be to have a single page, with the client programmed in Javascript and communicating with the server using an `XMLHttpRequest` object.

As for the regular parts of the assignment, you may only use your own code. However, in addition you are allowed to use open-source Javascript cryptography libraries. Examples include `http://www-cs-students.stanford.edu/~tjw/jsbn/`, `https://bitwiseshiftleft.github.io/sjcl/`, and `https://www.npmjs.com/package/crypto-js`. You are also allowed to use Javascript frameworks such as jQuery. In all cases, you make clear which code is not yours and give proper credits.

6 Non-functional requirements

In addition to the functional requirements, there is a number of non-functional requirements that your program must comply with. These are listed in this section.

All permanent state is stored in a SQLite database on the server side named `chat.db`, located directly within the application's root directory. This state includes (but is not limited to) users and sent messages. The client must retrieve this state from the server when needed. Both the server and the client may use the disk to store cryptographic keys, but only in a dedicated directory named `serverkeys` or `clientkeys` respectively, located directly within the application's root directory. The programs may not access each other's keys directories. The programs may invoke the trusted third party (a script), which can access the `ttpkeys` directory. Nothing else may be stored on disk by either program. Connection-bound information (which authenticated user is on the other end, which nonce was sent, etc.) may be stored in memory. Restarting the server or any of the clients should not result in any loss of data other than the need to re-establish connections.

Network connections are only set up by the clients, and only connect to the server. There are no direct connections between clients.

You may not use external programs or libraries except where specifically allowed in the assignment. You are allowed to use the C standard library, as well as the `crypto` (part of OpenSSL), `ssl` (also part of OpenSSL), and `sqlite3` libraries (part of SQLite). One easy way to tell whether a function is in the standard library or not is whether you need the `-l` linker flag to specify the library. You are only allowed to use `-lcrypto -lssl -lsqlite3`.

Your program may not include any code written by others except where specifically allowed. In particular:

- Using code from the slides or otherwise provided by the teacher or teaching assistants is always allowed.
- Using example code from the official documentation of libraries you use is allowed, but you must explicitly specify the source of the code in a comment to avoid plagiarism.
- Using code written by students who are not members of your group is never allowed, even if you change/rewrite it.
- Using other code from the Internet, including open source projects, is generally not allowed unless specific permission is obtained from the teacher in advance. Even in that case, the source of the code must be clearly acknowledged.

Note that all submitted work will be checked for plagiarism. Any plagiarism results in failing the assignment (and the course) and will be reported to the exam board. When in doubt whether something is permissible, ask us at sp@vusec.net beforehand.

The code you hand in may be shared anonymously with other students. Do not include your name, student number, VUnet id, or any other personally identifiable information anywhere in your code. Do not reuse code that you would like to remain secret. Also, make sure that there is nothing inappropriate or potentially offensive in your code (including comments, identifiers, etc.).

7 Security requirements

7.1 Threat model

The application must satisfy a number of security properties even in the face of a strong network-based attacker seeking to violate those properties. We will name this attacker Mallory in accordance with tradition. In particular, our threat model states that Mallory can:

- Determine at which addresses all clients and the server are running.
- Read, modify, inject, and/or block data sent over any network connection between a client and the server.

- Establish a connection with any client or the server, spoofing her network address to any possible value.
- Implement a malicious client to attack either the server or other clients by sending specially crafted data.
- Implement a malicious server and get clients to connect to it instead of the intended server, to attack clients by sending specially crafted data.
- Perform these actions any number of times, possibly simultaneously.

However, Mallory has no local access to the systems running your programs. She can only access them through your client and server programs. As such, she cannot access memory, access the disk, or intercept keyboard unless she compromises your program first.

7.2 Security properties

Within the threat model specified in the previous section, your programs must be able to satisfy the following security properties:

- Mallory cannot get information about private messages for which she is not either the sender or the intended recipient.
- Mallory cannot send messages on behalf of another user.
- Mallory cannot modify messages sent by other users.
- Mallory cannot find out users' passwords, private keys, or private messages (even if the server is compromised).
- Mallory cannot use the client or server programs to achieve privilege escalation on the systems they are running on.
- Mallory cannot leak or corrupt data in the client or server programs.
- Mallory cannot crash the client or server programs.
- The programs must never expose any information from the systems they run on, beyond what is required for the program to meet the requirements in the assignments.
- The programs must be unable to modify any files except for `chat.db` and the contents of the `clientkeys` and `clientkeys` directories, or any operating system settings, even if Mallory attempts to force it to do so.

It should be noted that we only require that you protect confidentiality and integrity. Under the given threat model, it is not possible to ensure availability entirely.

7.3 Key management

In case your program requires public key management, you may assume that there is a trusted third party (TTP) available who can identify all users and link them to their public keys. This third party can only use the OpenSSL command line tools and will not run any of your programs. You may implement the TTP as a shell script or python3 script, which can operate on the files in the `ttpkeys`, `clientkeys`, and `serverkeys` directories. You may invoke the TTP from your build system and/or from your program when setting up a new user. However, do keep in mind that the TTP is really a separate server that should receive as little private information as possible. Clearly state in the documentation exactly which information the TTP receives and which actions the TTP performs.

8 User interface

The server has no user interface other than the invocation specified in Section 3. It only interacts with the client programs using TCP sockets, and not with the user who started it directly. As such, the user interface of your application is defined by the client program.

The user interacts with the client by giving commands on the console. The client reads the standard input (`stdin`) for commands and sends output to the standards output (`stdout`). The syntax for the commands is defined as follows:

```
inputline       = [WHITESPACE] command [WHITESPACE] NEWLINE
command         = exitcommand | logincommand | privmsgcommand | pubmsgcommand |
                  registercommand | userscommand
exitcommand     = "/exit"
logincommand    = "/login" WHITESPACE username WHITESPACE password
privmsgcommand  = "@" username WHITESPACE message
pubmsgcommand   = message
registercommand = "/register" WHITESPACE username WHITESPACE password
userscommand    = "/users"
username        = TOKEN
password        = TOKEN
```

Each line of input should be interpreted according to the rules for `inputline`. To read the notation, note that quotes indicate literal text, brackets indicate optional parts, and pipe characters indicate that one of the options is to be selected. `WHITESPACE` means one or more space and/or tab (`"\t") characters. \verb`NEWLINE+ refers to the newline (`"\n"`) character. `message` is a string of one or more arbitrary characters, not containing newlines, not starting or ending with whitespace, and not starting with a forward slash (`"/"`) or at sign (`"@"`). `TOKEN` refers to one or more characters that contains neither newlines nor whitespace characters.

Note that the meaning of the commands specified in the syntax above corresponds to the features required in Section 5. Any input that does not follow the syntax should be rejected. In this case, the program provides an informative error message and allows the user to specify a new command. The same applies is a command fails or is not available at the time.

Below follows an example interaction with the client, where red text is typed by the user and blue text is the response from the system (note that there will be no colors in the real output):

```
$ ./client localhost 1234
connecting to server localhost:1234
/users
error: user is not currently logged in
/login other hunter2
error: invalid credentials
/register other correcthorsebatterystaple
error: user other already exists
/register erik hunter2
registration succeeded
this is a public message I sent
2021-11-01 09:30:00 erik: this is a public message I sent
/exit
$ ./client localhost 1234
/login erik hunter2
authentication succeeded
2021-11-01 09:30:00 erik: this is a public message I sent
2021-11-01 09:30:10 other: @erik this is a private message other sent
hello world
2021-11-01 10:00:00 erik: hello world
@other thanks for your message
2021-11-01 10:00:05 erik: @other thanks for your message
```

To simplify automatic testing, make sure the program works well with input or output redirection. In particular:

- Exit the program when the standard input reaches end-of-file. This is detected when `read` returns zero, or `getchar`/`fgetc` returns `EOF`.
- Disable buffering of the output to ensure it is instantly sent to the user. If you use `printf` or similar, call `setvbuf(stdout, NULL, _IONBF, 0)` when the program starts.

9 Documentation

9.1 Code Documentation

You must include a plain text file named `README.md` that clearly documents the information described below. You may use Markdown (see `https://www.markdownguide.org/basic-syntax/` for the basic syntax), but this is not required. The contents must include at a minimum:

- How your server and clients communicate, including an overview of all possible types of interactions between server and clients, the data layout of all possible types of packets sent, and (where applicable) a description of any cryptography you apply. The description should be sufficiently detailed for a third party to be able to write a new client or server program that is able to interface with your programs, without having to read your code.

- The documentation should also make clear how you achieve the required security goals. Briefly describe each potential attack you defend against, and why that approach is effective. See Section 7 for a description of the required security properties. If there are security properties that you cannot satisfy under the given threat model, identify these cases and explain why.
- Please include references to your code wherever appropriate to make grading easier. For example, when describing a message format indicate where those messages are composed (file+function name(s)) and where they are parsed (file+function name(s)).

If you know the right design but are unable to implement it, clearly indicate in the `README.md` file which parts are not in the implementation. If you implement a feature but do not mention it in `README.md` it may be overlooked while grading.

Note that the code itself should also include comments that help understand how it works and make the code more readable. The `README.md` file covers the high-level overview of how your application as a whole works.

9.2 Group description

You must also include a plain text file named `group.txt`. This file starts off with a line for each group member, providing the following information, in the specified order, separated by pipe characters (`|`): student number, VUnet id, full name, and e-mail address. For example:

```
1234567 | eke500 | Erik van der Kouwe | vdkouwe@cs.vu.nl
7654321 | jde123 | John Doe           | j.doe@example.com
```

Use this exact format, as it will be parsed automatically using scripts. After the list of members, add an empty line followed by a brief description how each group member contributed to the project. Note that `group.txt` is the only file that may include personal information.

10 Deadlines and submission

There are three deadlines where you must submit (part of) your code. In all cases you submit the specified files on Canvas before the end of the day (23:59).

10.1 Deadline A: 16 November 2021

You must submit a basic but functional chat framework including both the client and server. You implement the networking part (sending and receiving messages) and the user interface (parsing commands and showing messages). You do not need to support authentication, listing of users, or cryptography yet. You are also not required to use the database yet. However, it would be wise to keep in mind that you will need to add these features later when writing the code. The parts that do not need to be implemented also do not need to be in the `README.md` file yet.

You hand in a single a single `.tar.gz` file, which includes at least the following files:

- The file `README.md` file described in Section 9.1;
- The file `group.txt` described in Section 9.2 (this is the only file allowed to contain any personally identifiable information);
- The source files (`*.c` and `*.h`);
- The makefile(s) (`Makefile`).

You will get feedback to indicate which problems you can expect, which you should address for deadline C. However, note that it is possible that we overlook issues at this point so be sure to also check your own code again before handing it in.

10.2 Deadline B: 23 November 2021

On this deadline you do not hand in a new version of the program, but rather your design for the use of cryptography in your application. The design includes the following:

- An overview of each type of message exchanged between client and server, including whether cryptography is applied and how.
- An overview of your approach to key distribution.
- An explanation of how this addresses the requirements in Section 7.

You hand in this document as part of your `README.md` file.

You will get feedback to indicate which problems you can expect, which you should address when implementing the cryptography for deadline C. However, note that it is possible that we overlook issues at this point so be sure to also check your own design again before handing it in.

10.3 Deadline C: 7 December 2021

This is the final application, which should support all the requirements in the assignment. Even if you do not manage to meet all requirements, prioritize at least making sure that the program works. Make sure you pass all the tests in `test.py` (see Section **??** about testing).

The files to hand in are identical to deadline A, but now also optionally includes files for the web chat feature if you choose to implement it (web-related file types such as `.html`, `.js`, `.css`, etc).

11 Grading

Most of the grade (8.0 points) is decided by the final work you hand in on deadline C. Grading is based largely (but not exclusively) on how well-designed and well-implemented security-related mechanisms in your code are. In particular, when grading we consider the following questions:

- Did you apply secure programming guidelines discussed in the lectures?
- Does your code meet all functional requirements?
- Does your code meet all non-functional requirements?
- Does your code meet all security requirements?
- Does your code meet all documentation requirements?
- Is the user interface functional?
- Is the database design appropriate?
- Is the code well-documented, properly organized, and easy to read?
- Is the code reasonably efficient and elegant, using appropriate data structures and not sending much more data over the network than needed?

In addition to this, you can also get:

- At most 1.0 point for the work handed in at deadline A. Here we primarily look for completeness, so all parts required for deadline A work, you get the full 1.0 point.
- At most 1.0 point for the work handed in at deadline B. You get the full 1.0 point if your design is completely adequate.
- At most 1.0 point for the suggested additional functionality. You get the full 1.0 point if web chat is properly implemented and completely functional.

Note that this may add up to a total of up to 11.0 points if you implement the additional functionality.

We will deduct points in case of problems. Examples include (but are not limited to) compiler warnings, errors, crashes, and incorrectly submitted code. Note that a program that does not at least pass all the tests in `test.py` may not be graded at all (see Section **??** about testing). Assignments that include plagiarized code will also not be graded.

12 Asking Questions

Frequently asked questions can be found in Section E. Please do not hesitate to contact us in case anything is unclear or you encounter problems while doing the assignment. The goal is to learn by doing, and we will gladly help you achieve that. You can contact us in the following ways:

- If you have questions about the topics discussed in the lectures, post them on the Canvas discussion forum.
- If you have questions about the assignment that do not reveal (parts of) the solution, post them on the Canvas discussion forum.

- If you have questions about the assignment that do reveal (parts of) the solution, send them to sp@vusec.net. These e-mails will be read by the teacher and all teaching assistants.
- If you need one-on-one assistance, this is available during the scheduled lab sessions over Zoom.
- If you have questions of a personal nature that you prefer not to share with the student assistants, send them directly to the teacher at vdkouwe@cs.vu.nl.

We make an effort to help you as soon as we can. When asking questions, please make sure we can do so by including all relevant context and, where applicable, your source code.

A Getting started with C

C++ evolved from C and for most practical purposes, C++ is a superset of C. This means that, as a C++ programmer who will be programming in C, you will need to know which C++ features are not available and what suitable alternatives are available. These will be provided in this section.

The following are some of the most important differences in the language itself:

- C does not support classes, which can be replaced by structs.
- C structs only contain data members (without visibility specifiers). The equivalent of a method would be to declare a function taking a pointer to the struct as its first argument (the this pointer). To make a method private, omit it from the header files and mark it `static`.
- C does not support constructors, which means that any initialization of newly allocated memory must be explicit to avoid uninitialized reads.
- C does not support destructors, which means that any cleanup of memory to be freed must be explicit to avoid memory leaks.
- C does not support the `new` operator. Instead, you can use the `malloc()` function. This requires you to compute the size yourself (using the `sizeof` operator, multiplying by the number of array elements if applicable). The result is a `void*` pointer, which is implicitly casted to any other pointer type (so be careful the type matches).
- C does not support the `delete` operator. Instead, you can use the `free()` function.
- C does not support inheritance. An equivalent of data inheritance can be achieved by including the parent struct as the first member of the child struct.
- C does not support polymorphism. Function pointers can serve as a replacement for virtual functions. The syntax can be somewhat awkward: `returntype(*varname)(paramtypes)`.

- C does not support templates. Some people use arcane macro tricks to achieve similar effects, but avoid this unless you very desperately need a template equivalent. The more common solution is to use `void*`, but keep in mind this is not type safe and often requires additional manual memory management.

- C only supports casts of the form `(type)expr` (C-style casts). There are no implicit casts (except between `void*` and other pointer types) and no safe casts.

- C does not support references. Replacing them with pointers is generally trivial, but keep in mind that (unlike references) pointers may be `NULL`.

- C does not support function overloading. Functions cannot have the same names even if the parameter lists differ.

- C has no `bool` type. You can use any integer type instead, with zero representing false and any non-zero value representing true. Be careful when mixing sizes because integer overflows could turn a true into a false. The easiest solution is to use only the values zero and one.

- In C, the correct way to specify an empty parameter list when declaring a function is `(void)` rather than `()`. The latter does compile, but means something different (the parameter list is unknown). Empty parenthesis are still the correct way to call a parameterless function.

- C does not support exceptions. While `setjmp()` and `longjmp()` provide a similar construct, they are not widely used as a replacement due to the risk of memory leaks. Instead, C functions almost always use their return value to indicate whether they succeeded or failed. Not checking for error results is a common source of vulnerabilities.

In addition to these language differences, the standard library in C is much more limited. In particular, anything in the `std` namespace is missing. This has the following impact:

- C does not support `std::string`. Instead, strings use the `char*` type and are stored in `char` arrays or memory specifically reserved by `malloc()`. The length of C strings is usually determined by storing a null byte at the end. Support functions such as for example `strcpy`, `strcat`, `strdup`, and `strcmp` are available in the `string.h` header. Be careful to avoid buffer overflows and memory leaks.

- Data structures such as `std::list`, `std::vector`, and `std::map` are not available at all. You have to implement them yourself, again avoiding buffer overflows and memory leaks. A function to sort arrays, `qsort()`, is available however.

- I/O streams are not available, and are replaced by `FILE*` for buffered I/O. See `fopen()`, `fread()`, `fwrite()`, and `fclose()` for basic file operations. `fprintf()` allows formatted output containing multiple data of types. The first argument is the format, a string containing sequences starting with `%`, which are replaced by the formatted values of the later arguments.

The percent sign is followed by a type specifier, such as `%s` for strings or `%d` for integers formatted in decimal. Note that if the format string is under user control, this is a major security risk.

- `std::cin` is replaced by `stdin`, which can be passed as a file to the buffered I/O functions. `getchar()` reads a single character from the standard input.
- `std::cout` is replaced by `stdout`, which can be passed as a file to the buffered I/O functions. `printf()` writes formatted output.
- `std::cerr` is replaced by `stderr`, which can be passed as a file to the buffered I/O functions.

Please see your operating system's man pages for details on the functions mentioned here.

B Getting started with sockets

This assignment requires that you use sockets to communicate between your client and server programs. However, given that this is not a networking course, we will provide some examples of how to correctly use sockets for this purpose. Feel free to copy this code for your assignment.

A socket is a channel between two processes. These processes may run on the same machine or on different machines connected by a network. The channel is bidirectional. If one process writes to the socket, the other process will receive this data the next time it reads from the socket. By default, socket operations are blocking. The read operation blocks until data was written on the other end, while the write operation may also block if there is so much pending unread data that the buffers are full. When the connection is closed on the other end, `read` returns zero.

To set up the socket, one process must first listen for incoming connections, after which the other process can connect to the first. These processes are named respectively the server and the client. The server chooses a specific port to listen on, while the client specifies both the network address of the other machine and the port number. The port allows one to run multiple servers on the same machine, and no two servers on the same machine can listen on the same port number. There is a specific address (localhost, encoded as `127.0.0.1` in IPv4) that allows a client to connect to a local server. For this assignment, you will use the IPv4 protocol to identify machines (as IPv6 still does not work everywhere) and the TCP protocol to transmit data (as the alternative UDP does not ensure data is received).

Each socket is identified by a file descriptor, which is a small integer that can be passed to file I/O functions to specify which socket to use. The file descriptor is returned from the `socket` function initially creating the socket.

B.1 Creating a server socket

The code below creates a server socket listening on the specified port.

```c
#include <netinet/in.h>
#include <sys/socket.h>

int create_server_socket(unsigned short port) {
  int fd, r;
  struct sockaddr_in addr;

  /* create TCP socket */
  fd = socket(AF_INET, SOCK_STREAM, 0);
  if (fd < 0) { /* handle error */ }

  /* bind socket to specified port on all interfaces */
  addr.sin_family = AF_INET;
  addr.sin_port = htons(port);
  addr.sin_addr.s_addr = htonl(INADDR_ANY);
  r = bind(fd, (struct sockaddr *) &addr, sizeof(addr));
  if (r != 0) { /* handle error */ }

  /* start listening for incoming client connections */
  r = listen(fd, 0);
  if (r != 0) { /* handle error */ }

  return fd;
}
```

Afterwards, the `accept` call on this socket blocks until there is an incoming connection, which it returns a file descriptor for.

B.2 Connecting a client socket

The code below creates a client socket and connects it to the specified server (which may be localhost) on the specified port.

```c
#include <netdb.h>
#include <netinet/in.h>
#include <string.h>
#include <sys/socket.h>

static int lookup_host_ipv4(const char *hostname, struct in_addr *addr) {
  struct hostent *host;

  /* look up hostname, find first IPv4 entry */
  host = gethostbyname(hostname);
  while (host) {
    if (host->h_addrtype == AF_INET &&
      host->h_addr_list &&
      host->h_addr_list[0]) {
      memcpy(addr, host->h_addr_list[0], sizeof(*addr));
      return 0;
    }
    host = gethostent();
  }

  /* unknown host */
  return -1;
}

int client_connect(const char *hostname, unsigned short port) {
  struct sockaddr_in addr;
  int fd, r;

  /* look up hostname */
  r = lookup_host_ipv4(hostname, &addr.sin_addr);
  if (r != 0) { /* handle error */ }

  /* create TCP socket */
  fd = socket(AF_INET, SOCK_STREAM, 0);
  if (fd < 0) { /* handle error */ }

  /* conect to server */
  addr.sin_family = AF_INET;
  addr.sin_port = htons(port);
  r = connect(fd, (struct sockaddr *) &addr, sizeof(addr));
  if (r != 0) { /* handle error */ }

  return fd;
}
```

B.3 Handling incoming connections

The server needs to be able to service multiple connections at the same time. You can achieve this by calling `fork` each time the `accept` call returns a new incoming connection. `fork` creates a new process to handle input from the client (we will call this process a worker), while the main server process continues to accept new incoming connections. Note that, at first, the worker and server processes are identical except for the return value of `fork`. This code provides an example of how to achieve this:

```
#include <stdlib.h>
#include <sys/socket.h>
#include <unistd.h>

void worker_start(int connfd);

int accept_connection(int serverfd) {
  int connfd, r;
  pid_t pid;

  /* accept incoming connection */
  connfd = accept(serverfd, NULL, NULL);
  if (connfd < 0) { /* handle error */ }
  /* create new worker process */
  pid = fork();
  if (pid == (pid_t)-1) { /* handle error */ }
  if (pid == 0) {
    worker_start(connfd);
    exit(0);
  }

  /* server continues here */
  return 0;
}
```

You should avoid using threads for this assignment. While they seem a convenient alternative to `fork` at first, the multitude of race condition bugs they can introduce make the assignment more difficult than it should be. While threads make it easier for workers to communicate between each other, very limited communication is needed for this assignment because all shared state is in the database. If you do need communication between server and workers, we recommend using the `socketpair` function before `fork` to create a communication channel. The programs can use the `select` function to wait for input on the connection socket or the socket pair before calling `read` or `accept`. The most elegant solution would work with `select` without a deadline, which avoids busy waiting (which wastes CPU cycles) and polling (which introduces unnecessary delay and also wastes CPU cycles).

It is important to note that `select` interacts poorly with buffering. For example, `select` on `STDIO_FILENO` will not detect any data remaining in the `stdin` buffer that can be read without blocking. For file streams such as `stdin`, this can be solved by calling `setvbuf` with mode `_IONBF` or by using `read` on `STDIO_FILENO` directly, bypassing buffering. Unfortunately, when using OpenSSL, buffering is often required because a full block of data must be read before it can be decrypted. See `ssl_has_data` from the lecture on cryptography for a solution.

B.4 Sending a message step-by-step

In the proposed design, delivering a message goes through a number of steps. The processes involved are shown in Figure 1. The steps are:

- User types message in stdin;
- Client sends message to worker over socket;
- Worker stores message in database (not shown in diagram);
- Worker notifies server over socket pair;
- Server notifies other workers over socket pair;
- Other workers retrieve message(s) that the client has not seen from database (not shown in diagram);
- Other workers send message(s) to their clients over socket;

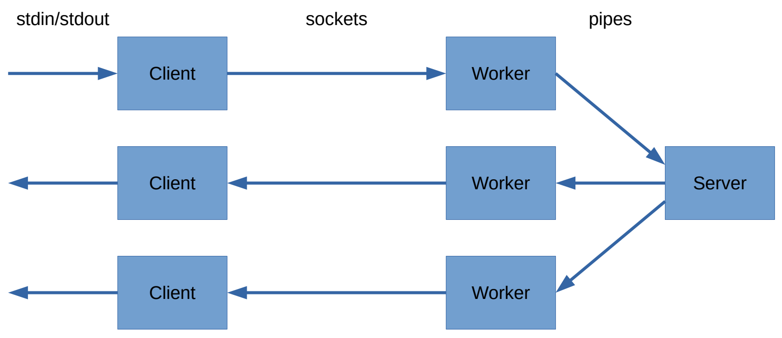

Figure 1: Steps when sending a message, boxes are processes

- Clients show message(s) on stdout.

Note that you are not required to follow this design, it is a suggestion to what we think would be the simplest solution.

C Getting started with OpenSSL

OpenSSL is a widely used library that provides cryptographic primitives and related support functions. It implements the SSL protocol (used for authentication and encryption in HTTPS) as well as various encryption algorithms (both symmetric and asymmetric), cryptographic hashes, certificates, a random number generator, and the Diffie-Hellman key exchange. It also includes offers some of the fundamentals for cryptography (such as big numbers), support functions (for example to store or load cryptographic keys on disk), and wrappers that make the algorithms easier to use (such as buffered I/O functions). A full list of functions and links to their man pages can be found at `https://www.openssl.org/docs/man1.1.1/man7/crypto.html`. The course slides op cryptography will also include some examples of using the OpenSSL API.

Some indication of how to use the available functions:

- The functions with names starting with EVP provide a general interface to do hashing/encryption/decryption/signing/validation regardless of the specific algorithm.
- Most calls require a context of a specific type to be set up for OpenSSL to store its state. For example, to sign a message you create a `EVP_MD_CTX` context using the `EVP_MD_CTX_create` function. See the examples for all the contexts that you need. Do not forget to free the context afterwards (in this case with `EVP_MD_CTX_free`).
- In many cases, there are `*Init`, `*Update`, and `*Final` functions. In these cases you start by calling Init once, then add your data in blocks using Update as many times as needed, and finally you call Final once to get the (end of the) result.

- By default, keys and certificates are stored on disk in the PEM format. The `PEM_*` functions can read these files. For example, `PEM_read_RSAPrivateKey` reads an RSA private key and provides a pointer to it in memory.
- Be sure to check for error conditions (the examples do not do this). Most functions return 1 if they complete successfully and a non-positive number on failure. `ERR_print_errors_fp(stderr)` will dump the error message(s) to stderr.

In addition to the library functions, OpenSSL offers a number of command line utilities to (amongst others) manage key generation and certificate signing. The utilities are documented on `https://www.openssl.org/docs/man1.1.1/man1/`, and some concrete examples on `https://geekflare.com/openssl-commands-certificates/`.

D Getting started with SQLite

SQLite allows you to easily set up a local database without the need to have server program running. The database is simply a local file, and SQLite is a library that interprets SQL queries on this database. SQLite is well-documented. A very short quick start guide can be found at `https://www.sqlite.org/quickstart.html` and a full overview of the C API at `https://www.sqlite.org/c3ref/funclist.html`. The course slides (SP05) also include examples of using the SQLite API.

SQLite manages access to the database file to ensure safe concurrent access. By default, it fails with `SQLITE_BUSY` when your query conflicts with a simultaneous query from another process, signalling that you have to retry later. Use the `sqlite3_busy_timeout` function on your database connection to automatically keep retrying for a given number of milliseconds, which should usually avoid this error and make programming easier.

E Frequently Asked Questions

Other students asked these questions before, so it might be worthwhile reading through them to avoid getting blocked on the same issues.

E.1 Deadlines

Deadline A: Do you need to implement users? No, it is not necessary because authentication is not required. You can show a placeholder as the name instead. The important thing is that sending/receiving messages works. Of course you will not pass the test, but this is not needed for deadline A

Deadline A: Should I use the database? You are not required to use the database for deadline A, but you are allowed to. You could use files or (shared) memory if you want for example. You can use an alternative design if you feel that is easier, but must make sure that the final program (submitted for deadline C) does comply with all requirements (including storing state only in the database).

Deadline A: What is needed? You should implement all functionality except what is excluded specifically in the assignment PDF. This means you should be able to connect to the server, parse commands, send messages, and receive messages. Note that this is more than just echoing messages, because the messages are sent to everyone and old messages are shown at the start. This means that you have to be able to send a message from one client and have it appear in the other clients. Private messages, however, are not needed because they are impossible without authentication.

Deadline B: What is needed? Deadline B is not a programming deadline, and you do not hand in a new version of your program. You only provide a design of how you plan to use cryptography to address the security requirements. Essentially you map the topics discussed in the lectures onto the requirements specified in the assignment PDF. You should provide sufficient explanation how you will do this (for example, consider key management), not just list the techniques.

E.2 Requirements

Can I use another build system? This is allowed as long as you still meet the requirements of the assignment. In particular, there must be a proxy makefile satisfying the requirements, and the required tools must be available by default on a standard system as described in the assignment. CMake is specifically allowed, and we will install it if needed. If you really need another build system not installed by default, ask first at sp@vusec.net and specify why you need that particular build system.

Can I use poll() and/or shared memory to talk between the workers? Yes, this is allowed. Note that with shared memory you will face some of the same issues as with threads, but much more contained (especially because the workers need very little communication between them). Also see the other question about `select`.

Can I use something else than select() to wait for input? You are not required to use `select`, and you are allowed to use non-blocking sockets. Some students prefer to use `poll` instead and that is fine. However, even when using non-blocking sockets you will need to use some form of wait function, because otherwise you would be doing polling or busy waiting and those are inefficient designs.

Please take a look at the Figure 1. Communication between clients and workers uses sockets. Workers can use a socket pair to communicate with the central server program, which accepts new connections and forks off new workers. All processes can use `select` in this scenario:

- clients select between the standard input and the socket;
- workers select between the connection socket and the socket pair to the central server;
- server selects between the server socket and the socket pairs to the workers.

You are allowed to use alternative designs, but you are not supposed to use non-standard libraries (unless specified otherwise in the assignment). I do think the proposed design is the easiest solution, and I know it works because I implemented it like that myself.

How am I supposed to handle messages sent when the recipient is offline? Both private and public messages can be sent when the recipient is offline, but a private message cannot be sent before the user registers. In this case, they will be shown whenever the recipient next signs in.

How am I supposed to handle the same user logging in from multiple clients? You are not required to support the same user logging in from multiple clients, but you are allowed to do so. In either case, you should ensure an attempt to do so does not result in incorrect bookkeeping of online users. If a user cannot log in because another client is already logged in under their account, they must receive a proper error message.

Is using regular expressions allowed? You can use regular expressions in the `regex.h` header, as they are part of the standard C library. Documentation can be found here: `https://pubs.opengroup.org/onlinepubs/009695399/basedefs/regex.h.html`.

What to show when the client starts? When the client starts, after the user logs in or registers a new user, it shows the full chat history. There is no limit on the number of messages shown or the time the were sent. Messages that were sent before the user registered are also shown. The only filtering you need to do is to ensure that users will not see private messages unless they are either the sender or the recipient.

Which text encoding should I use? You are not required to handle text encoding, and can send text as if it is binary data. In practice this means you will use the terminal's default, which is probably UTF-8. We only require that ASCII works correctly, which would be true in that case.

E.3 Design and Implementation

How do file descriptors and select() work? The idea is that select allows you to wait until one or more out of a list of file descriptors is ready for reading and/or writing. This is useful for example, when you are expecting data on multiple file descriptors but you do not know which will come first so you cannot just read one (which would block). A file descriptor is a small integer which identifies an open file. In UNIX, a file can be one of several things: a regular file on the disk, a directory, a pipe, a socket, a terminal, or even a hardware device. Each process has its own table of open file descriptors in the kernel, so a file descriptor in one process will not point to the same file in another process.

How to parse the input? You are not allowed to use external libraries other than what is specified in the assignment, and you should not need them either because the parsing in this assignment is very simple. The key here is that you need to change your mindset to not view a string as an abstract object, but rather as a concrete pointer to a number of bytes in memory. This is how C deals with strings, and looking at code from this perspective is a very important skill to be able to understand computer security.

C provides very little help in handling strings, but as long as it is in-place, pointer arithmetic does make it quite easy and extremely efficient. String manipulations that cannot be done in-place (such as adding characters to strings) require additional explicit memory management (for example, calls to `realloc`) and is not as easy, but should not be needed for this assignment.

A list of the string handling functions available in the C standard library, and links to their documentation, can be found here: https://pubs.opengroup.org/onlinepubs/007908799/xsh/strin

As for parsing, I would recommend not making it more difficult than it needs to be. Create a char* pointing to the first char, and just start matching using if, while, and switch statements. Simply match individual chars with ==, or longer strings with `strncmp`. Once you matched something, increment the pointer and start looking for what you expect afterwards. To give a very simple example, this removes zero or more whitespaces at the start:

```c
#include <ctype.h>
/* ... */
char *p = buf, *p_end = buf + len;
while (p < p_end && isspace(*p)) p++;
```

Here, `p` keeps track of where in the string we are and `p_end` is a pointer to the end of the buffer, which we use to prevent running over the end. The string to be parsed is in `buf`, and has length `len`.

How to communicate securely with the TTP? You do not need to worry about secure communication with the TTP. You can assume that calling the script is equivalent to visiting the TTP in person, showing your ID, and getting a USB stick with the certificate. In reality, the scripts puts the key and/or certificate in the appropriate directory (`serverkeys` or `clientkeys`).

How to read user input from stdin? There are several ways to read user input from stdin. Note that `fscanf` is a rather sloppy parser and might make it hard to satisfy the requirements. In my opinion, using `read` from `STDIN_FILENO` is the best choice because it does not buffer the input. This means you can use `select` between the client socket and stdin without needing to take special precautions due to buffering.

Why would one use fork() in the server rather than select() on all sockets? You are not required to use `fork`, though I would recommend it. A design with a single process and `select` is elegant, but requires an elaborate state machine to get it right. Otherwise, you either allow clients to block other clients, or you lose data if requests are sent piecemeal. While `fork` requires extra code for communicating between the workers, communicating with the client is easier.

Security-wise compartmentalization would also have potential benefits. Each worker will store OpenSSL state, possibly including keys, in memory. If one client can compromise their own worker to allow reading data but not changing control flow, it does not provide them benefits in listening in on the other clients.

Why would one use select()? An example can be found in the client. You can get input from the user on stdin (file descriptor `STDIN_FILENO`, which is always 0) and from the socket to the server in case a message comes in. This is a minimal example of how you could handle that:

```
int fdmax;
fd_set readfds;

/* specify which file descriptors to wait for */
FD_ZERO(&readfds);
FD_SET(STDIN_FILENO, &readfds);
FD_SET(socketfd, &readfds);

/* we need to tell select about the largest file descriptor */
fdmax = (STDIN_FILENO > sockfd) ? STDIN_FILENO : sockfd;

/* wait for at least one fd to become ready */
select(fdmax+1, &readfds, NULL, NULL, NULL);

/* handle ready file descriptors */
if (FD_ISSET(STDIN_FILENO, &readfds)) {
  handle_user_input();
}
if (FD_ISSET(socketfd, &readfds)) {
  handle_socket_input(socketfd);
}
```

The function `handle_user_input` can call read on `STDIN_FILENO` once, knowing that it will not block. The function `handle_socket_input` can call read on `socketfd` once, knowing that it will not block.

Once a pipe or socket reaches the end-of-file, it is also marked ready. In this case read returns zero. In these cases, end-of-file means that there will be no more data coming in because the other end has been closed. When using fork in the server, be sure to close any file descriptors you do not need for this reason.

Is it better to make one big SQL query or many small ones? It depends on the situation. Benefits of the former include less need for error handling, and less concurrency risks because SQLite will automatically make the single query into a transaction. Benefits of the latter allow better debuggability, and there may be some opportunity to reuse functions wrapping the queries. In the latter case you must make sure you create transactions where needed.

E.4 Grading

The test infrastructure in test.py is passing many tests (but not all, something like the first 50if you supply two programs, client and server, that terminate immediately. Passing the tests is simply a requirement that tests whether your program can compile, run without crashing, and give the expected output when you send a message. These are the minimum requirements for us to be able to grade your submission. It is not otherwise relevant for grading, as these are not the grading criteria (see Section 11 for grading criteria). However, do feel free to make your own more elaborate tests. This is a great way to improve the quality of your software.

E.5 Key Management

How to generating keys? You are allowed to have the TTP generate private keys (along with the certificate) for the client and the server. This means that you can use the command line OpenSSL invocations from the slides and the makefile of the example almost literally. This saves some work compared to doing it in C. The TTP and server can generate their keys at build time (you can invoke the TTP from the Makefile, or include the commands directly in that file), but the client cannot as you do not know which users will be created. A sensible solution is to start the TTP script whenever a new user is registered.

How to send the keys on the socket/distribute the keys? This is part of the course material and will be discussed in the lectures on cryptography.

You need to send public keys over the socket if you're going to use asymmetric cryptography. You cannot send the RSA structure directly, but need to serialize it into a data stream. There are several possible formats, and the default one for OpenSSL is PEM. This is text-based, so you can treat the serialized key simply as a string. PEM is also the format used to store keys in files, so you can read the file and send the string directly. The `cert-verify.c` example shows how to load a PEM certificate from memory.

Do keep in mind that a public key contains no information about its owner. Wrapping the public key in an X5.09 certificate solves this issue. These certificates can be stored as PEM just like the keys they contain. The `cert-verify.c` example shows how to extract a public key from a certificate.

How to use the keys directories? The client is only allowed to read and write the `clientkeys` directory. Moreover, the client may only read the key and certificate of the client currently logged in. The server is only allowed to read and write the `serverkeys` directory. The TTP is allowed to read and write all three keys directories.

You may place the CA certificate in the `clientkeys` directory at build time to allow you to check the other certificates (equivalent with how Firefox comes with CA keys preinstalled), but not the server certificate itself.

Note that the requirement that the programs are unable to write outside the proper directories means that you need to add proper validation/escaping where necessary, and obviously avoid exploits that could lead to a filename being overwritten.

What is the role of the TTP? Note that the TTP only generates key and certificates. It is not needed to verify them (the CA signature allows anyone to do that) and is not involved in the communication between client and server.