CS5430: System Security Programming Project (Spring 2025)

Phase 2: User Authentication

General Instructions. Work together in the same group you formed for phase 1.

Due Dates:
Part 1 Design Notes. 3/12 (Wednesday) 11:59pm on CMS
Part 2 Extended System Code. 3/26 (Wednesday) 11:59pm on CMS

The secure channels implemented in Phase 1 provide no assurance to the server about the identity of the human user who is submitting commands at the client during a session. Phase 2 somewhat fills that gap. The system that you develop for Phase 2 will authenticate the identity of the human user who invokes a `LOGIN` command to initiate a session. In general, we authenticate human users by checking what they know, what they have, or what they are. Phase 2 is based on what the user knows. It involves: (i) designing a password scheme for authenticating a human user who is attempting to begin a session and (ii) incorporating that scheme into the authentication protocols you delivered for Phase 1.

New Operations. Phase 2 introduces a new operation `REGISTER`. It lets a user `u` with user id `uid` register a password `pass` with the system, as follows.

```
{"uid": "fbs", "op": "REGISTER", "pass": "GoBigRed2025"}
```

A `REGISTER` operation fails if a user has already registered successfully previously.

Subsequent `LOGIN` operations from any user `u` require `u` to provide both `uid` and `pass` to the client.

```
{"uid": "fbs", "op": "LOGIN" "pass": "GoBigRed2025"}
```

A `LOGIN` operation fails if the value `pass` provided by `u` for the `LOGIN` operation does not match the password previously provided by `u` during `REGISTER`.

Users are also allowed to change their passwords using a new operation `CHANGE_PASS`. It lets the user in a current session with currently registered password `old_pass` update their password to `new_pass`, as follows.

```
{"op": "CHANGE_PASS", "old_pass": "GoBigRed2025", "new_pass":
                        "NewBuilding2025"}
```

A `CHANGE_PASS` operation fails if a user does not provide the correct currently registered password as the value for `old_pass`. Therefore, a user must have performed `LOGIN` and be executing within a session in order for a `CHANGE_PASS` operation to succeed.

Various designs are possible for this kind of authentication scheme. A good design will be informed by what is being assumed about the threat. For your system, assume that the client is more likely to be attacked successfully than the server, but that the server could be attacked and have its state (except for its private key) copied. Also assume that, over time, the same client computer might be used by a series of human users who do not trust each other.

Here are some questions to ponder for your design:

- Where are passwords and any related information stored?
- What information should be stored when a user gets registered?
- What is the appropriate cryptography for storing passwords in a secure way? Suggested reading on password hashing: https://en.wikipedia.org/wiki/Key_derivation_function#Password_hashing.
- What information is sent over the network when a password is either registered or a password is provided for an attempted `LOGIN`? In what format?
- Since a `REGISTER` command is not part of a session, what channel should be used for that communication and what security properties are required for that channel?
- Since a `CHANGE_PASS` command is always part of a session, what should happen after a `CHANGE_PASS` either succeeds or fails?

Implementation Details. Extend the `Operation` enum located in `types/request.go` to support password registration by adding `REGISTER` and `CHANGE_PASS` as possible values for the `Operation` enum. Extend the `Request` struct located in `types/request.go` to support password values `pass`, `old_pass` and `new_pass` being provided for `REGISTER`, `LOGIN`, and `CHANGE_PASS`. The three fields `pass`, `old_pass` and `new_pass` should syntactically appear in the order presented and placed after the current fields of `Request`. Remember that your program should support the expected syntax we have provided. Feel free to make other changes to the `Request` struct. Please document how you have extended `Request` so that the course staff may test your system as you intended it to be used.

You will be extending the code base that your group members submitted for Phase 1. Feel free to introduce new datatypes under the `types/` folder, but do not modify `networkdata.go`, or `response.go`.

For password hashing, you may find Go's supplementary cryptography package https://pkg.go.dev/golang.org/x/crypto to be useful. To include this package in your project first add the following line `golang.org/x/crypto` to the requirements section (this is the part of the file that begins with the keyword `require`) of your top-level `go.mod` file. Then, you can use the package in any Go file by adding the line `golang.org/x/crypto` to the import section (this is the part of the file that begins with the keyword `import`) of that file.

The Phase 2 Assignment: Two parts

Part 1. Design the protocols for password-based authentication of users. Document this design by describing the protocol, the properties it is intended to enforce, explaining why you believe the properties are enforced, and giving your answers for the above bullet-list of design questions. To describe the protocol, use the "message list diagram" format employed in lecture. We believe that these design notes can be under 5 pages.

Part 2. Program the protocol you designed in Part 2 and submit that codebase. If necessary, revise the protocol from Part 2, but explain those revisions and the reason for them.

Grading and Submissions

This programming assignment should be implemented in Golang version 1.23.0, which is available on the `ugclinux` computers. Include the line `go 1.23.0` in your `go.work` and `go.mod` files to have the Go compiler use version 1.23.0.

For Part 1: Submit a .pdf file `design.pdf` to CMS.

For Part 2: Submit a .zip file `Phase2Impl.zip` to CMS. This .zip file should include all of the files needed so that it will compile using `build.sh` and run using `run.sh` on the `ugclinux` computers, including all of the following.

- `build.sh`, a script that the grader can invoke to compile your project. This can be a modified version of the `build.sh` that we provided. It must output an executable named `main` that the grader can then invoke to test your solution.
- Source folders including `client/`, `crypto_utils/`, `network/`, `server/` and `types/` and the files `main.go`, `go.mod`, and `go.work` at the top-level folder containing your implementation of phase 2.
- `README.txt` which describes any changes you made to the `build.sh` script.
- `testRationale.txt` which describes what functionality of the program has been checked and how those checks were made. For example, you may include some testing files and run the system under file mode. Include as part of this discussion a listing of the outputs that a correct system should produce for each test.
- `revisions.txt` which describes whether there were any revisions to your protocol proposal from part 1 and explain why changes had to be made.

Grading. Project grades will be based on the following rubric

- 40% -- Part 1 describes a good design accompanied by compelling justifications.
- 30% -- Part 2 system operates correctly on tests provided by course staff to exercise functionality.
- 10% -- how well the tests provided in `testRationale.txt` exercise the functionality and security of the system.
- 10% -- the quality of the explanations in `testRationale.txt`.

- 10% -- code quality, readability, and documentation in the form of comments.

Automatic deduction of up to 50% if the program does not compile using `build.sh` or does not run using `run.sh` on the `ugclinux` computers. *Be smart: Try to rebuild your system on the `ugclinux` computers before you submit it.*