CS5430: System Security Programming Project (Spring 2025)

Phase 1: Secure Channels

General Instructions. Work together in a group of 2 or 3 students from the class.

Due Dates:
Part 1 Design Document. 2/21 (Friday) 11:59pm on CMS
Part 2 Extended System Code. 3/5 (Wednesday) 11:59pm on CMS

The key-value store in Phase 0 uses a network to link the client and the server. That system can be subverted by a Dolev-Yao attacker who might insert, modify, or monitor the request messages sent by the client or the response messages sent by the server. Phase 1 is focused on eliminating this class of vulnerabilities. You will (i) design protocols to resist Dolev-Yao attackers and (ii) implement those protocols by extending the Phase 0 code. So, Phase 1 is concerned with creating a secure channel by using the network that we provide. This secure channel will enable authenticated, secret, and integrity-protected communication between the client and the server and vice versa.

The extensions made when building your Phase 0 deliverable resulted in the interactions between the client and server being partitioned into sessions, with at most one session at any time. Each session starts with a `LOGIN` operation and ends with a `LOGOUT` operation. Prudent security engineering would have each session use a separate secure channel, involving fresh secrets. So, compromising the secure channel implemented for a session would not help an attacker to compromise the traffic that was sent during a previous session or a later session. The secure channel associated with a session would be established in response to a user `u` submitting a `LOGIN` operation and the secure channel should be terminated in response to `u` submitting a `LOGOUT` request.

Cryptographic Building Blocks

To implement secure channels, you will want to employ cryptography. Code to perform common cryptographic functions was included in the file `crypto_utils/crypto_utils.go` that you downloaded with the rest of the system. Use these routines as your building blocks.

In systems that use public-key cryptography, some means is required to provision the principals with the public keys for other principals. A certification authority is often employed. Our project ignores this part of the picture and, instead, does some pre-provisioning, as follows.

- The server has two local variables named `publicKey` and `privateKey` that are populated with appropriate values during initialization. In addition, during initialization, the server saves the byte string of `publicKey` in a file `SERVER_PUBLICKEY`.

- During network initialization, a method `ObtainServerPublicKey()` in the client is called. This method populates the client's `serverPublicKey` variable from the file `SERVER_PUBLICKEY`. *This method should not be modified.*

If your implementation of secure channels requires other public/private key pairs or requires symmetric keys, then your code should invoke the appropriate routines from `crypto_utils.go` to generate the keys and distribute them. A good rule of thumb: A private key that was generated by a principal should not be communicated to any other principal. *Your system should not store any keys or any other information in the file system, to preserve the fiction that the client and server are running on physically separate machines.*

There are many ways to use cryptography for protecting confidentiality, integrity, and authenticating the endpoints of a channel. Different cryptographic functions have different costs in terms of execution times, key sizes, and encrypted message lengths. Be mindful of those costs when designing your protocols.

The Phase 1 Task

Phase 1 is split into two parts, with a different delivery date for each part.

> **Part 1**. Design the protocols to implement a channel that enables authenticated, secret, and integrity-protected communication between the client and the server and *vice versa*. Document this design by describing the protocol, the properties it is intended to enforce, and explaining why you believe the properties are enforced. To describe the protocol, use the "message list diagram" format employed in lecture. This document should be only a few pages long.
>
> **Part 2**. Implement the protocol you designed in Part 1 and submit that codebase. If necessary, revise the protocol design submitted in Part 1, explaining those revisions and the reasons they were necessary.

For Part 2, you may use the code that one of your group members submitted for Phase 0 or you may create a new code base by merging the code bases submitted by your group's members.

Feel free to introduce new datatypes under the `types/` folder when building Part 2—but do not modify `networkdata.go`, `request.go` or `response.go`. You may have to define a new struct that supports authentication of requests and responses in a way that is easily serialized and deserialized.

The course staff will test the security of your system by using a <u>different</u> implementation of network simulator `network.go`. That alternative implementation gives testers the control needed for launching Dolev-Yao attacks. We encourage you to try implementing your own version of such a network simulator, so that you can test your secure channels under Dolev-Yao attacks. *Be warned: If the system you submit does not work correctly with the originally distributed implementation of `network.go` then your system will not work correctly in the grading environment.*

Grading and Submissions

This programming assignment should be implemented in Go 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 `Phase1Impl.zip` to CMS. This zip file should incorporate the following.

- `build.sh`, a script that the grader can invoke to compile your project. This script 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 file `main.go` at the top-level folder containing your implementation of phase 1.

- `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

- 5% -- Part 1 gives an appropriate list of properties to be supported
- 15% -- Part 1 write-up is clear and explains why properties follow
- 20% -- Part 1 protocols do not have vulnerabilities
- 20% -- Part 2 system operates correctly on tests provided by course staff to exercise functionality
- 15% -- how well your tests in `testRationale.txt` exercise the functionality of the system.
- 15% -- the quality of the explanations in `testRationale.txt`.
- 10% -- code quality, readability, and documentation in the form of comments.

Submissions will receive an automatic deduction of up to 50% if the program does not compile using `build.sh` or does not run either in interactive or file mode on the `ugclinux` computers. *Be smart: Try your system on the `ugclinux` computers before you submit it.*