CS5430: System Security Programming Project (Spring 2025)

Phase 0: Extending a secure key-value store

General Instructions. Each student should work on this assignment individually.

Due Date. 2/7/2025 at 11:59 pm (in CMS)

This semester, you will build a secure key-value store. The project will involve multiple phases. In each phase, you will add code to defend against a different class of vulnerabilities. The final product will be a system that authorizes requests based on the identity of the user making the request.

Phase 0 is intended to familiarize you with the Go programming language and with the codebase for an insecure client/server key-value store system. You will be working alone to ensure that you master all facets. In subsequent phases, you will work in a group.

The system comprises of a single server that manages a key-value store and a single client that gets requests from users, invokes the appropriate server operations, and returns any results to the user. The client and server communicate over a network that is simulated by code we provide. There will be no need for you to be concerned with multiple clients. However, designs that do not easily scale to handle multiple clients will receive lower grades than designs that can easily be scaled.

As should be clear, you are not starting with a clean slate. Rather, you will add code to a given, existing skeleton—just like the situation you will encounter as a software developer. Part of your challenge, then, will be to understand how an existing system works, why the system is structured in some given way, and how to add code without corrupting that structure.

The Software Distribution. The Phase 0 system is found in a collection of folders and files. These are distributed in `phase0.zip`, which is available for download from CMS. It contains the following:

- `build.sh`: This file compiles the entire system. Details about compilation are given below. *Do not modify this file unless you have a compelling reason, such as using an external library that requires modification to the compilation process.*

- `main.go`: This file is the entry point to the system. *In Phase 0, you will be modifying this file.*

- `client/`: File `client.go` contains the code for the client. This code makes requests to the server on behalf of users. *In Phase 0, you will be modifying this file.*

- `crypto_utils/`: File `crypto_utils.go` contains functions to perform common cryptographic operations. The file has no external dependencies, so you should be able to

compile and use it without incorporating other functions. *In Phase 0, you should not be using any of functions in this file.*

- `network/`: File `network.go` contains code to simulate a network that connects the client to the server. *You should never modify this file.*

- `server/`: File `server.go` contains code for the server. This code processes requests sent by the client, takes appropriate action based on the request, and reports the result back to the client. *In Phase 0, you will be modifying this file.*
- `types/`: The folder contains three Go files:
    - `networkdata.go` defines the type for messages that can be sent over the network,
    - `request.go` defines the type for requests sent by the client, and
    - `response.go` defines the type for responses sent by the server.

  You will most likely modify `request.go` and `response.go`. You should not modify `networkdata.go`.

System Operation. In the code we provide (and in the code you submit), the client, network, and server each executes as a separate Go routine. Our synchronization code ensures that a client never sends a new request if there a request is awaiting a response from the server.

The client puts requests into the client's Go channel `Requests` that the network simulator reads; the network then transfers that request to the server's Go channel `Requests`. Once the server is done processing a request, the server puts a response into the server's Go channel `Responses`; the network simulator then transfers that response to the client's Go channel `Responses`.

For the client to send a message to the server and receive a response, it uses the method `sendAndReceive` in `client.go`. For the server to receive a message from the client and send a response, it uses the method `receiveThenSend` in `server.go`. The method `receiveThenSend` invokes another method, called `process`. *You should not modify the methods `sendAndReceive` and `receiveThenSend` or the Go channels of the client and the server. However, you may modify `process` to change the functionality of the server.*

Code that we are providing will execute to initialize certain data structures before your code starts. All initialization code of a Go file must be included in the method named `init`. The server and the client are each assigned a unique name during initialization. The client's name is stored in a variable called `name` in `client.go` and the server's name is stored in a variable called `name` in `server.go`. *You may extend our initialization code, but do not modify the initialization code already provided*.

The detailed system operation is best understood by first studying the datatypes defined in the folder `types/`. In `types/networkdata.go` the Go struct `NetworkData` defines the format of messages sent over the network. `NetworkData` consists of two fields: Field `Name` should be populated with the name of the entity (either the client or the server) that is sending the

message. Field `Payload` should be populated with a byte array containing the data to be sent over the network. *Do not modify the `NetworkData` struct.*

The client and server we provide employ a fixed format for the client to make requests and for the server to return responses. These messages are Go structs that adhere to the JSON format. For more detail on JSON in Go, including examples and the procedures for serialization and deserialization of JSON-adhering structs, see: https://gobyexample.com/json.

The type of a request is a Go struct named `Request`; it can be found in `types/request.go`. A request has three fields:

- `Key`: Corresponds to the keys being stored in the key-value store. Any non-empty string can be a valid key in our key-value store.

- `Val`: Corresponds to the values being stored in the key-value store. Any arbitrary Go value (as indicated by the data type of values: `interface{}`) can be a value in our key-value store.

- `Op`: Corresponds to the operations that can be requested by the client. The `Operation` enum type in `types/request.go` is the list of every operation the system is supporting.

The type of a response is a Go struct named `Response`; it can be found in `types/response.go`. A response has two fields:

- `Status`: Describes the outcome of an operation that the server attempted to perform. The `Code` enum type in `types/response.go` lists the two possible outcomes (`OK` for successful operations and `FAIL` for unsuccessful operations) for the operations currently supported by the server. In adding support for other operations, you may need to add new labels for possible outcomes.

- `Val`: Corresponds to a value the server is returning to satisfy an operation the client requested.

Certain operations and/or their outcomes may cause `nil` or default values to be assigned to certain fields of `Request` or `Response`. For example, with a key-deletion request, the client should have provided a value for a `Key` to be deleted and should have set `Op` to `DELETE` in `Request`. But a value for `Val` need not be supplied when making a key-deletion request. And, in response to the key-deletion request, the server sets the `Status` field of `Response` as appropriate, but does not provide a value for `Val`.

After familiarizing yourself with the datatypes in the folder `types/`, look at the source code in the folders `client/` and `server/`. The code there has been structured to make the system's operation easy to understand.

Compiling the System. To create an executable for the full system (i.e., the client, the server, and the network), run the command `./build.sh` from the command line. After running this command, an executable named `main` will appear in the current directory. Re-run this command each time you would like to re-compile the system.

You may notice that a file `json.go` is created in the folder `types/` after the first time you compile the system. This file is generated by the build script to assist in the serialization and deserialization of the `Request` and `Response` datatypes. You should not need to understand its functionality or how it was generated. Throughout the code base, you may find files that contain comments of the form `//go:generate ….` These comments instruct the Go compiler to generate some files (such as the `json.go`) during compilation. *Do not remove these comments from the files.*

Running the System. Executable `main` can be run in two modes.

- **Interactive mode**: Type `./main -i` into the command line to enter interactive mode. After the prompt >> appears on the screen, you may enter a command. Each command you enter must be a valid JSON string, corresponding to the `Request` struct. For example, the command to read the value associated with the key "CS5430" would be

  ```
  >> {"key": "CS5430", "op": "READ"}
  ```

  followed by a newline. After entering that command, you should see two lines of output:

  ```
  Input: {"key":"CS540","val":null,op":"CREATE"}

  Output: {"status":"OK","val":"security"}
  ```

  The first output line is identified by the prefix `Input:`. This line is displaying how the input was parsed. The second output line is identified by the prefix `Output:`. This line is showing the response that the client received from the server, after requesting the operation. In the above example, we see that the operation succeeded because the `status` of the operation is listed as `OK`.

  In this example, we only provided a partial JSON string as input, since the JSON object in the read request makes no reference to the field `val`. The system will populate missing fields. This can be seen in the example above. The JSON string that was input did not make any reference to the field `val`, yet we see displayed on the `Input` line that `val` has a value—it is the default (`null`) for that field.

  To exit interactive mode, input an `EOF` (end of file) signal. In Unix (and Unix-like systems, such as MacOS) this is done by pressing *Control-D* on your keyboard.

- **File mode**: Type `./main -f <`*`path/to/file_name`*`>` into your command line to have the client obtain a sequence of commands by reading the file

`<path/to/file_name>`. Each line in *file_name* must be a valid JSON string, corresponding to the `Request` struct. Program `main` will read commands from the file and execute them, one by one, until an `EOF` (end of file) signal is read. The output will be stored in the file `<path/to/file_name_output>`.

The Phase 0 Assignment: Three Parts.

Part 0: Familiarize yourself with Go. There is an official Go tutorial from the Go team at https://go.dev/doc/tutorial/getting-started. We urge you to complete this tutorial.

For further information about programming in Go, you may want to look at the following:

- https://gobyexample.com
- https://golangbot.com/learn-golang-series/
- https://medium.com/a-journey-with-go
- https://dave.cheney.net
- https://quii.gitbook.io/learn-go-with-tests
- https://www.youtube.com/playlist?list=PL64wiCrrxh4Jisi7OcCJIUpguV_f5jGnZ
- https://www.youtube.com/playlist?list=PLoILbKo9rG3skRCj37Kn5Zj803hhiuRK6

We will spend a little time in class on Go programming, but that presentation will not suffice for completing this project.

Part 1: Extend the existing key-value store system with a `COPY` operation. This operation

```
{"src_key": "CS5430", "op": "COPY", "dst_key": "fbs"}
```

requests that the value associated with `src_key` be copied to `dst_key` provided both `src_key` and `dst_key` already exist in the key-value store—that is, both keys have previously been created. (Check the server source code and you will see that a key is created using the `CREATE` operation and that a key is deleted using the `DELETE` operation.) If a `COPY` operation succeeds then `Status` in the response from the server should be set to `OK`; otherwise, `Status` should be set to `FAIL`.
Hint: You will need to extend the struct `Request` to support a `COPY` command. The serialization and deserialization process for JSON structs will automatically support the parsing of your extended `Request` type.

Part 2: For the key-value system ultimately to authorize operations based on who is making the request, the system will have to be associating a principal with each request. The first step is to extend the existing system to support rudimentary `LOGIN` and `LOGOUT` operations. Assume that every user `u` has a user id (say) `uid` that is some string. A user `u` at a client begins a session by issuing

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

specifying that the operation is set to `LOGIN` and providing `fbs` as the value for `uid`. The same user `u` ends the current session by issuing

```
{"op": "LOGOUT"}
```

specifying that the operation is set to. At most one session may be active at a client at any time. At most one session may be active at the server at any time. During a session, every `Request` submitted should include the `uid` of the user `u`, and every `Response` sent by the server should echo the `uid` found in the `Request`. The `uid` of the user should be inserted into the `Request` automatically by the client, overwriting whatever `uid` was provided in the input to the client.

Grading and Submissions

Use 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.

Download from CMS the zip file `Phase0.zip` to start working on the project. When you are done, submit to CMS a zip file `Phase0Impl.zip` that contains the following.

- `build.sh`, which is a script that the grader can invoke to compile your project. This can be a modified version of the `build.sh` that we provided. Your `build.sh` must output an executable named `main` that the grader can then invoke to test your solution.

- The folders `client/`, `network/`, `server/` and `types/` and the file `main.go` at the top-level folder containing your implementations for parts 1 and 2.

- `README.txt` which describes any changes you made to the `build.sh` script.

- `testRationale.txt` which describes what functionality of your extensions have been checked and how those checks your happening. 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.

Grading. Project grades will be based on the following rubric

- 50% -- system operates correctly on our test cases
- 20% -- how well your tests in `testRationale.txt` exercise the extended functionality of the system and don't compromise the initial functiomality.
- 10% -- the quality of the explanations in `testRationale.txt`.
- 20% -- 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.*