CS5430: System Security Programming Project (Spring 2026)

Phase 0: Extending a key-value store

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

Due Date. 2/6/2026 at 11:59 pm (in CMS)

This semester, students will be extending a key-value store server and client. The project involves multiple phases. In each phase, you will add code that provides new security functionality and/or that defends against a different class of vulnerabilities.

The system is intended to be compiled using Java 25. Phase 0 will familiarize you with the codebase for an insecure client/server key-value store system. You will be working alone to ensure that each student is well prepared to participate in subsequent phases. In the remaining phases, students will work in a group of 2 or 3.

The architecture of the system is straightforward. We use Java's module system to separate each component of the system into a module.[1] Here is a high-level view: The system consists of a module called `app.module` that contains a single class `App` that gets input from the standard input (a keyboard or a file) and invokes the `Parser` class, located in the `parser.module`, to parse the input request string into a map that is passed to the client. The `Client`, located in the module called `client.module`, invokes the `Parser` class to serialize the map into a byte array that is then passed to the network. The network is simulated by the `Network` class, located in the `network.module`. The network delivers the request to the `Server` class, located in `server.module`, where the request is processed and a response is produced. Responses are returned to the network, which returns them to the client, which in turn returns them to `App`. Finally, `App` invokes parser to write the response to the standard output. Notice, that the network only accepts and returns byte arrays.

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 bash script compiles the entire system. *Do not modify this file. That means you cannot use functionality that is not provided in the standard Java library.*

- `clean.sh`: This bash script deletes all files and folders produced by `build.sh`.

- `run.sh`: This bash script runs the compiled system produced by `build.sh`.

[1] An official guide for Java modules can be found here: https://dev.java/learn/modules/. A useful unofficial guide can be found here: https://www.baeldung.com/java-modularity.

- `src/app-module/`: Defines the `app-module` module. File `app/App.java` contains the code for reading from the standard input and writing to the standard output. This code relays request to the client on behalf of users and prints responses the client receives. *In Phase 0, you are not allowed to modify any file in this folder or add new files to this folder.*

- `src/client-module/`: Defines the `client-module` module. File `client/Client.java` contains the code for the client. This code makes requests to the server on behalf of users. *In Phase 0, you are not allowed to modify any file in this folder or add new files to this folder.*

- `src/network-module/`: Defines the `network-module` module. File `network/Network.java` contains the code for the network. This code simulates a network that connects the client to the server. *In Phase 0, you are not allowed to modify any file in this folder or add new files to this folder.*

- `src/parser-module/`: Defines the `parser-module` module. File `parser/Parser.java` contains the code for the parser. This code performs parsing operations by converting `String` to `Map<String, String[]>`, and vice versa. Serialization is provided; it converts the `Map<String, String[]>` datatype to the `byte[]` datatype, and vice versa. *You should never modify the contents of this module or add new files to this folder.*

- `src/server-module/`: Defines the `server-module` module. File `server/Server.java` 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 <u>will</u> be modifying this file.*

- `baseline.tests:` A collection of test cases that can be used to exercise the system before any of the Phase 0 extensions have been made.

- `baseline.tests.out:` The expected output from running the system with the input `baseline.tests.`

- `phase0.tests:` A (non-exhaustive!) collection of test cases that can be used to exercise the system with the Phase 0 extensions in place.

- `phase0.tests.out:` The expected output from running the system with the input `phase0.tests.`

System Operation. Familiarize yourself with the semantics of the operations that are implemented by reading the source code and by observing the effects of running the sample input file that we have provided. The syntax for starting the system and running it with that input file is:

```
./run.sh < baseline.tests
```

Another way to gain familiarity with how the system operates is by experimenting with inputs that you provide.

You should have already noticed that input lines and output lines are lists of triples having the form

> *opr-name* = *opr-value*;

where *opr-name* is a predefined keyword consisting of only lower-case letters, *opr-value* is a string of characters (that may include arbitrary whitespace). There may be arbitrary whitespace surrounding the = sign, and the triple must end with a semicolon. All strings are case-sensitive and the non-whitespace characters are restricted to alphanumeric characters.

Any input line should specify the name of an operation by including the triple

> … `op` = *cmdName*; …

The triples that precede or follow should specify values for the operands expected by operation *cmdName*. Files `baseline.tests` and `phase0.tests` specify operands in a particular order, but by consulting the source code you will see that this order does not need to be followed. However, each input line should describe a single operation—descriptions of operations may not span multiple lines.

An output line specifies the results of an operation by giving the triple

> … `status` = *statusName*; …

possibly preceded or followed by other triples. Those other triples would specify other values, as appropriate for the operation that was requested. For now, the only values for *statusName* are `OK` or `FAIL`. Files `baseline.tests.out` and `phase0.tests.out` specify triples in a particular order, but by consulting the source code you will see that this order is not determined. Each output line, however, will describe the results of a single operation.

Notice, that in processing each input line, the line is listed, followed by some form of output. This output of a command invocation could be

- the input line repeated, which means there is either a syntax error in the input line or some sort of failure regarding serialization/deserialization, or

- output based on the state of the client and the server.

The Phase 0 Assignment

The system we are providing does not distinguish between different users. You will now remedy that shortcoming with your Phase 0 extensions. We assume that each system user will be identified by a `String`, which we refer to as that user's *user-id*.

- The server should be extended to implement and maintain a *registry* that holds the user-id for each user that is authorized to access the system.

- The server should be extended to keep track of an *active-user* that equals the user-id for the user that is currently submitting inputs to the client.

The intended semantics for the registry and active-user are supported by the commands `REGISTER`, `LOGIN`, and `LOGOUT` you will be adding to the system. The expected syntax and operation of these new commands are as follows.

The `REGISTER` command

```
op = REGISTER;  uid = user-id;
```

adds *user-id* to the registry (independent of whether there currently is an active-user). If *user-id* is already in the registry then the command should fail. The output for this command should be a single triple giving a value for `status`.

The `LOGIN` command

```
op = LOGIN;  uid = user-id;
```

changes the active-user to *user-id* provided *user-id* is present in the registry. If *user-id* is not in the registry or if there is already an active-user, then the command should fail. The output for this command should be a single triple giving a value for `status`.

The `LOGOUT` command

```
op = LOGOUT;
```

changes the active-user to *no-user*, signifying that there is no active user. If there was no active user when the command was invoked, then the command should fail. The output for this command should be a single triple giving a value for `status`.

There are also changes to the semantics of the `CREATE` and `DELETE` commands in order to implement a form of ownership of keys. The operation of all other commands is unchanged, though.

The *owner* of a key is the user-id of a user that is authorized to do certain things that other users are not.

The `CREATE` command

```
op = CREATE; key = k;  val = v;
```

is extended to also set the owner of key *k* to be the active-user. The command should fail if it is invoked when there is no active user. The output for this command continues to be a single triple giving a value for `status`.

Finally, the `DELETE` command

```
op = DELETE;  key = k
```

is extended so that it only succeeds if the key *k* exists in the system and the owner of key *k* is the active-user.

Grading and Submissions

Use Java 25, which is available on the `ugclinux` computers. Submit file `Server.java` and zip file `tests.zip` to CMS for grading. We will use your implementation of `Server.java` to compile a system with the other files that we originally provided to you.

Grading. Project grades will be based on the following rubric

- 50% -- system operates correctly on our test cases

- 50% -- file `tests.zip` containing test data (in one or more files) that demonstrate correct operation of your system, along with file `testRationale.txt` that explains how well your tests exercise the extended functionality of the system and don't compromise the initial functionality.
  - 30% -- how well your tests exercise the extended functionality of the system and don't compromise the initial functionality.
  - 20% -- the quality of the explanations in `testRationale.txt`.

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.