CS5430: System Security Programming Project (Spring 2026)

Phase 1: DAC Authorization for Key-Value Store

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

Due Date: 2/27 (Friday) 11:59pm on CMS

With discretionary access control (DAC), the principal that creates an object is the principal that has authority to delete that object and to define which principals can perform various operations involving that object. In this phase, you will be implementing DAC for the key-value store. DAC functionality will enable enforcing restrictions on which users are allowed to perform various operations on each key.

Access Control Semantics for Key-Value Store. The details of the DAC authorization scheme that the key-value store should support are sketched below. The principals are all possible users (existing or not); the objects are all possible keys (existing or not).

In Phase 1, each key *k* will be associated with the following metadata.

- *k.writers*: those principals directly authorized to write the value associated with key `k`.
- *k.readers*: those principals directly authorized to read the value associated with key `k`.
- *k.indirects*: a set of keys that augments *k.writers* and *k.readers* as described below.

If a key *k* has no associated value—because it has not yet been created or because it was subsequently deleted—then each of these access control sets is empty.

Notice, there are no restrictions on the contents of the *access control sets* above. So, it is possible to specify that a principal is authorized to write to a key but not read it. Also, there is no requirement that the elements of access control sets already be present in the registry that the `REGISTER` command populates or in the set of keys that have been created but not deleted.

The $k.indirects$ set allows the authorization for a key k to depend on authorizations for other keys—the keys that are elements of $k.indirects$. In particular, the set $R(k)$ of principals that are authorized to read a key k is defined as follows:

$$R(k) := k.readers \cup \bigcup_{k' \in k.indirects} R(k')$$

Authorization for `WRITE` is determined analogously, but defining the set $W(k)$.

Supporting DAC authorization means that `READ` and `WRITE` commands may terminate with `status = FAIL` because the requested operation is not authorized.

New Operations. In this phase, the `CREATE` operation for a key *k* should be extended to include (optional) arguments that will initialize some or all the access control sets. The additional operand names are `readers`, `writers`, and `indirects`. And an operand value that is the finite set `{a, b, c}` is specified by giving the elements separated by whitespace. So, for example. the input

```
op = CREATE; key = ans; val = 23; readers = fbs gs;
```

would create a new key `ans` having initial value 23. Associated with this key would be the access control set `readers` containing set `{fbs, gs}` and the other access control sets associated with this key would be initialized to empty sets. The output for this command remains to be a single triple giving a value for `status`.

A new `MODACL` ("modify ACL") operation in this phase provides a way for the principal that created a key to change some, or all, of the access control sets associated with that key. This command gives new lists of principals for some or all of `readers, writers,` and `indirects`. If an access control set is not included with a `MODACL` operation request, then no change should be made to the contents of that access control set; if an access control set is given as just whitespace then the server should interpret this to mean that the access control set should be reset to empty. So, for example

```
op = MODACL; key = ans; readers = fbs; writers = gs;
```

would change the `readers` access control set for key `ans` to `{fbs}` (so principal `gs` was removed), change the `writers` access control set to authorize access by `gs`, and leave unchanged all the other access control sets. The output for this command should be a single triple giving a value for `status` (i.e., `OK` or `FAIL`).

A new `REVACL` ("review ACL") operation provides a way for the principal that created a key to review the authorizations that are allowed by the current values of the access control sets for that given key. Execution of a `REVACL` operation for a key *k*

```
op = REVCAL; key = k;
```

should display the contents of all the access control sets associated with key *k* as well as the values of $R(k)$ *and* $W(k)$. The output for this command should be a triple giving a value for `status` (i.e., `OK` or `FAIL`). If `status` is `OK`, then five lists should also be printed:

```
indirects = ...; readers = ...; rk = ...; wk =...; writers =
...;
```

where k is replaced by the key.

The Phase 1 Assignment

Download `phase1.zip`, which is available on CMS. This is version of the system does not contain a file `Server.java` in the folder `server-module`. The version also has an updated

parser that now accepts an *opr-value* consisting of only whitespace in a triple: "*opr-name = opr-value;*". This *opr-value* will be parsed as the empty string. You may need to update your Phase 0 server code to accommodate this change. In addition, the revised parser prints output a well-defined deterministic order.

Implement this DAC access control and submit `Server.java` code. You may derive this by modifying and/or combining the `Server.java` that any of your group members submitted for Phase 0. Explain any design decisions and choices you made. For example, explain where and how you store the access control sets and what data structures will you use? **Your submission will be assessed, in part, on how performant your code is in making authorization decisions.**

An example input file and the expected output can be found in phase1.zip, which is available for download from CMS.

Grading and Submissions

This programming assignment should be implemented in Java version 25, which is available on the `ugclinux` computers.

Submit file `Server.java`, containing the code for an extended version of the server, and `Tests.zip`, containing test data (in one or more files) that demonstrate correct operation of your system, to CMS. The .zip file `Tests.zip` should incorporate the following.

- `testRationale.txt` that explains how well your tests exercise the extended functionality of the system and don't compromise the initial functionality.
- `design.txt` which describes the design decisions behind your implementation, including a discussion of the expected performance of your code.

Grading. Project grades will be based on the following rubric

- 40% -- system operates correctly on tests provided by course staff to exercise functionality.
- 20% -- the performance/efficiency of the portions of your code.
- 20% -- 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% -- the quality of the explanations in `design.txt`.

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 your system on the `ugclinux` computers before you submit it.*