CS5430: System Security Programming Project (Spring 2025)

Phase 3: DAC Authorization for Key-Value Store

General Instructions. Work together in the same group you formed for phase 2.

Due Date: 4/27 (Sunday) 11:59pm on CMS

With discretionary access control (DAC), the principal that first 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 3, you will be implementing DAC for the key-value store. This 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 users; the objects are keys (along with an associated value and metadata).

Each key *k* is not only associated with a value but is associated with the following metadata.

- *k.writers*: the set of principals authorized to write the value associated with key `k`.
- *k.readers*: the set of principals authorized to read the value associated with key `k`.
- *k.copyfroms*: the set of principals authorized to use key `k` as a `src_key` for `COPY` operations.
- *k.copytos*: the set of principals authorized to use key `k` as a `dst_key` for `COPY` operations.
- *k.indirects*: a set of keys that augments *k.writers*, *k.readers*, *k.copyfroms* and *k.copytos*, as described below.
- *k.owner*: the principal that created key *k* and the only principal authorized to delete key *k*.

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. It is also possible to specify that a principal is authorized to copy from a key but not be authorized to read that key.

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` and `COPY` operations is determined analogously, by defining sets $W(k)$, $C_{src}(k)$ and $C_{dst}(k)$ respectively.

A principal is allowed to use `COPY` for making the value (but not the access control sets) associated with one key k also be associated with another key k', provided (i) both k and k' exists in the key-value store, and (ii) the principal requesting the operation is a member of the set intersection $C_{src}(k) \cap C_{dst}(k')$.

New Operations. In Phase 3, the `CREATE` operation for a key *k* is allowed to include (optional) arguments that will initialize some or all of the access control sets. For example, the command

```
{"op": "CREATE", "key": "gs", "val":"TA", "readers":["fbs"]}
```

would create a new key `gs` for a value `TA`. Associated with this key would be the access control set `readers` containing a single element `fbs`; the other access control sets associated with key `gs` (i.e., `writers, copytos, copyfroms`, and `indirects`) would be initialized to the Go `nil` value, the JSON default when a command does not explicitly provide values for those sets; the intended meaning of `nil` is that these access control sets should be set to the empty set. Extend the `Request` struct in `types/request.go` to support `CREATE` operations that can also accept as input the fields: `writers`, `readers`, `copytos`, `copyfroms`, and `indirects`. The type of each of these fields should be a string list (`[]string`).

The `MODACL` operation 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, copytos, copyfroms`, and `indirects`. If an access control set is not included with a `MODACL` operation request, then that set will receive the Go value `nil`, which the server should interpret to mean that no change should be made to the contents of that access control set; if an access control set with a `MODACL` operation is given the empty list `[]` then the server should interpret this to mean that the access control set should be reset to empty. So, for example

```
{"op":"MODACL", "key":"gs", "readers":[], "writers":["fbs",
                                "kb"]}
```

would change the `readers` access control set for key `gs` to empty (so no principal is authorized to read), change the `writers` access control set to authorize access by `fbs` and `kb`, and leave unchanged all the other access control sets.

The `REVACL` 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* should return the contents of all the access control sets associated with key *k* as well as the values of $R(k)$, $W(k)$, $C_{src}(k)$ and $C_{dst}(k)$. Extend the `Response` struct in `types/response.go` to support the fields: `writers`, `readers`, `copytos`, `copyfroms`, `indirects`, `r(k)`, `w(k)`, `c_src(k)` and `c_dst(k)`. You must use these exact names for the JSON fields of your extension of the `Response` struct. The type of each of these fields should be a string list (`[]string`).

The Phase 3 Assignment

Implement this DAC access control and submit that codebase. 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.**

You may extend the code base that your group submitted for Phase 2. Feel free to introduce new datatypes under the `types/` folder, but do not modify `networkdata.go`.

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.

Submit a .zip file `Phase3Impl.zip` to CMS. This .zip file should incorporate 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 3.
- `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.
- `design.txt` which describes the design decisions behind your implementation, your discussion of the performance of your code should be included here.

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 that compute R, W, C_{src} and C_{dst}.
- 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% -- 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 your system on the `ugclinux` computers before you submit it.*