CS5430: System Security Programming Project (Spring 2026)

Phase 2: MAC Authorization for Key-Value Store

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

Due Date: 3/18 (Wednesday) 11:59pm on CMS.

The marketing department believes there would be customer demand for a key-value store that ensures children only can access "appropriate" content, where appropriateness is defined in terms of the child's age and certain beliefs held by the child's parents. To have a product for this market, Phase 2 extends the Phase 1 key-value store. In addition, to facilitate establishing assurance for the new system, management desires that changes to Phase 1 be minimized, since this would mean that the assurance argument for the Phase 1 code can carry-forward to the new system. So, as much as possible, new functionality should be implemented in new functions that you add to your codebase.

The Phase 1 server for the key-value store associates a *value* with a *key*. In addition, the Phase 1 server allows the user that creates a key *k* to specify (and alter) the set of users that are authorized to read and/or write the value associated with key *k*.

Phase 2 allows a user to transform a key into what we will call an *s-key*. The value that is associated with an s-key will be a string that is a *story*. Using an s-key *k* to access a value (i.e., a story) is restricted in two ways. First, the DAC read and write restrictions that Phase 1 enforces for accesses using *k* remain in effect. Second, restrictions to ensure *appropriate access* (as defined below) are enforced. Be clear: authorization to read and appropriate access are separate—one might hold when the other doesn't.

Appropriate access to a value is granted based on tags (*A,B*) that Phase 2 associates with s-keys and with users.

- In the tag (*A,B*) that is associated with an s-key, *A* is a positive integer giving a suggested minimum age for readers and *B* is a set of keywords enumerating the beliefs advocated in the story.

- In the tag (*A,B*) that is associated with a user, *A* is an integer approximation of that user's age and *B* is a set of beliefs this user may read about or write about.

A user *U* with tag (A_U, B_U) is allowed to access the value associated with an s-key having a tag (*A,B*) if $A \leq A_U$ and $B \subseteq B_U$ hold. So *U* is allowed to read a story associated with an s-key *k* only if *U* satisfies the DAC restrictions for reading the value associated with *k* and also the read would be an appropriate access.

Phase 2 Server Operations

All of the Phase 1 server operations continue to be supported. The execution of a few operations are modified, as detailed below. And two new operations are added.

`REGISTER.` The syntax of the `REGISTER` operation is extended. The extension associates a tag $(A, \{ b_1, b_2, \ldots b_n \})$, where $0 < A$ holds and $\{ b_1, b_2, \ldots, b_n \}$ may be empty, with the new userid *user-id* that is being created:

`op = REGISTER; uid =` *user-id*`; age =` A`; beliefs =` $b_1\ b_2\ \ldots\ b_n$ `;`

`READ.` The execution of the `READ` operation is modified, as follows, for the case where the value given for `key` is an s-key. A user *user-id* may read the value (i.e., story) associated with with an s-key *k* only if the Phase 1 DAC policy is satisfied for *k* and if reading the associated value would be an appropriate access.

`WRITE.` The syntax and the execution of a `WRITE` operation is modified, as follows, for the case where the value given for `key` is an s-key. The syntax is extended, so that triples can be optionally included that will cause a tag (A, $\{ b_1, b_2, \ldots, b_n \}$) to be associated with the key being updated.

`op = WRITE; key =` k `; val =` *string*`; age =` A`; beliefs =` $b_1\ b_2\ \ldots\ b_n$ `;`

For an existing key *k*, if triples for `age` and `beliefs` are provided then (i) *k* becomes an s-key (if it isn't already an s-key), (ii) the tag associated with the stored value is updated according to the values given for `age` and `beliefs`, and (iii) the value is updated according to the string *string* provided. A `WRITE` operation fails if *k* was not an existing key or if the user was not authorized to write to that key. In addition, a `WRITE` operation to an s-key fails if the existing tag of the s-key is T or the proposed new tag for the s-key is T, the user has used `READ` to successfully access content with tag T' since `LOGIN`, and a user with tag T reading content with tag T' would not be considered an appropriate access.

`COMBINE.` A new `COMBINE` operation

`op = COMBINE; key =` k `; val =` $k_1\, k_2 \ldots k_m$ `;`

(i) causes the already existing key *k* to become an s-key (if it isn't already an s-key), (ii) associates a value constructed by concatenating the values associated with s-keys $k_1, k_2, \ldots, k_m$ but adding an "_" (underscore) to separate those values, and (ii) constructs a suitable tag for *k*. This operation fails if (i) *k* is not an existing key or if the user is not authorized to write to key *k*, (ii) the user is not authorized to read the content associated with keys $k_1, k_2, \ldots, k_m$, or (iii) any of $k_1, k_2, \ldots, k_m$ is not an s-key.

Given these new commands, the life-cycle of an s-key can be described as follows. A key *k* is created. This key is made into an s-key using `WRITE` or `COMBINE`. The value of s-key *k* may

then be read using `READ` operations, but such a `READ` operation succeeds only if the user is authorized to read *k* and the value currently associated with *k* is appropriate for the user. The value of s-key *k* may be updated using `WRITE` (which may optionally also update the tag for *k*) or using `COMBINE`. Note, there is no operation to change an s-key back into a key—the key must be deleted and then created.

`REVTAG.` A new `REVTAG` ("review tag") operation provides a way for the principal that created a key to review the current tag for an existing key, if that key is an s-key. Execution of a `REVTAG` operation for a key *k*

`op = REVTAG; key =` *k*`;`

should display the values comprising the tag (A, { $b_1, b_2, \ldots, b_n$ }) associated with key *k*, if *k* is an s-key. The output for this command should be a triple giving a value for `status` (i.e., `OK` or `FAIL`). If `status` is `OK`, the following should also be printed:

`age =` *A*`; beliefs =` $b_1\ b_2\ \ldots\ b_n$ `;`

The Phase 2 Assignment

Implement this MAC access control and submit `Server.java` code. Derive this new system by modifying the file `Server.java` that any of your group members submitted for Phase 1. Explain design decisions and choices you made.

An example input file and the expected output can be found in `phase2.zip`, which is available for download from CMS. Beware these tests are far from exhaustive.

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. Also describe any changes you made to your Phase 1 submission in order to correct problems that we found in that.

Grading. Project grades will be based on the following rubric

- 30% -- system operates correctly on tests provided by course staff to exercise functionality.
- 20% -- the extent to which the extensions are implemented in a modular way and with the fewest changes to Phase 1 code.
- 10% -- errors in Phase 1 server were repaired in this version.
- 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.*