General Instructions. Work together in a group of 2 or 3 students from the class. You may work with the students from your Phase 1 group or with a different set of students.

This programming assignment should be implemented in Java version 11, which is available on the ugc-linux computers.

Due Date. November 8, 2023 at 11:59 pm (in CMS)

Project Overview

Authorization for a key-value store restricts which principals are able to perform the various specific operations associated with each key. With DAC (Discretionary Access Control), the principal that first creates a key is the principal that defines which principals are authorized to perform the various operations associated with that key.

The system you build for phase 2 will implement a new client interface and a new server interface that together support DAC for a key-value store that is shared among a collection of users who all execute within a single client. We discuss below how these interfaces should be implemented by using the client and server you implemented for Phase 1.

Access Control Semantics. Each userId corresponds to a principal. The authorization model to be implemented defines authorization in terms of three sets that are associated with each key $k$:

- $k$.writers: a set of userId’s that are authorized to write a value to be associated with key $k$.
- $k$.readers: a set of userId’s that are authorized to read the value associated with key $k$.
- $k$.indirects: a set of keys that are used to augment the sets of writers and readers for key $k$.

Set $k$.indirects allows the authorization for accessing a key $k$ to be based on authorizations for other keys — specifically, those keys that are elements of $k$.indirects. In particular, the set of userId’s that can read (say) a key $k$ is given by the set $R(k)$, defined as follows:

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

Authorization for write is defined analogously in terms of a set $W(k)$.

In addition to the three sets defined above, each key $k$ also has an associated owner, which is the userId that created key $k$. The owner of a key $k$ is the only userId that is authorized to read and/or write the sets $k$.writers, $k$.readers, and $k$.indirects associated with key $k$. Notice, it is possible to have a key where these sets imply that the owner cannot read or write that key.
All of the information used to implement the authorization model for a key $k$ should be stored as part of the meta-values associated with key $k$. Also, authorization for a request should be checked only if that request has been authenticated.

**Client Stub Interface.** The client stub for phase 2 extends the client stub from phase 1 by repurposing the phase 1 operations for reading and writing meta-values. In phase 2, these operations are used only for reading and writing the writers, readers, and indirects sets. The skeleton code you are given for phase 2 defines a ClientACLobject to be used for this purpose. The body of this object definition is empty, so you can define a suitable implementation.

**Server Interface.** The server you built for phase 1 allowed an arbitrary object type to be used as the meta-value associated with each key. In phase 2, this arbitrary object should specialized to a ServerACLobject, which we have defined in the skeleton code. The skeleton code does not give an implementation for ServerACLobject—you must create something appropriate. Note that your ClientACLobject and a ServerACLobject need not be the same, depending on your design.

In order to enforce the DAC security policy, the semantics of certain phase 1 server operations are changed in phase 2. Here is an informal description for the changed operations.

```java
public AbstractAuthenticatedDoResponse<K, V, ServerACLobject>
    authenticatedDo(AbstractAuthenticatedDoRequest<K, V, ServerACLobject> request)
    throws RemoteException;

// Invoked in response to a client stub request to perform
CREATE, DELETE, READVAL, READMETAVAL, WRITEVAL, or
WRITEMETAVAL. The operation to invoke is specified by the
field operation, which is a field of the doOperation field of
the request.
Return AUTHENTICATION_FAILURE if the request cannot be
authenticated as being on behalf of user userId; return
AUTHORIZATION_FAILURE if the request does not satisfy the
authorization requirements given below; otherwise, return one of
SUCCESS, NOSUCHELEMENT, or ILLEGALARGUMENT according to the
semantics of the key-value store operations defined in Phase 0.
CREATE: No authorization privileges are required if the key does
not already exist. If the key does exist, then the invoker must
be the owner of that key.
DELETE: The invoker must be the owner of the key.
READVAL: The invoker must be authorized to read that key.
READMETAVAL: The invoker must be the owner of the key.
WRITEVAL: The invoker must be authorized to write that key.
WRITEMETAVAL: The invoker must be the owner of the key.
```
We have updated the outcome field of `DoOperationOutcome` so that it now includes `AUTHORIZATION_FAILURE` as a possible value for this field.

**System To Be Built.** As before, we provide a code skeleton, as follows. And, as before, you modify this code skeleton in order to implement a system that enforces DAC authorization.

- **client:** This Java object (named `Phase2App.java`) makes requests to the key-value store. For our purposes, the client is providing a sequence of tests that exercise the rest of the system. *You write the code for this object.*

- **client stub:** This Java object (named `Phase2StubImpl.java`) implements an interface named `Phase2Stub.java`. The client invokes methods defined by this interface in order to (i) register at the server, (ii) set authorization, and (iii) perform key-value store operations at the server. The client stub invokes methods provided by the network in order to cause the invocation of methods that the server is providing. *We provide a skeleton for `Phase2StubImpl.java`, and you add code to this skeleton in order to implement the required functionality of the client stub methods.*

- **network simulator:** As with phase 1, this Java object (named `NetworkImpl.java`) simulates a network that connects the client stub to the server. *You should not make modifications to this code.*

- **server:** This Java object (named `Phase2ServerImpl.java`) authenticates and authorizes each requested operation to ascertain that it was issued for a registered user that has appropriate privileges. `Phase2ServerImpl.java` implements an interface named `Phase2Server.java`. *We provide a skeleton for `Phase2ServerImpl.java`, and you add code to this skeleton in order to implement the desired functionality of the server methods.*

You may use the code for any group’s Phase 1 in building Phase 2. The skeleton we are providing has a place where you can copy this code, so that it is incorporated into your Phase 2.

**Grading and Submissions**

Download from CMS the zip file `Phase2.zip`. It contains:

- **phase2.jar** which contains the definitions for the interfaces and abstract classes that are discussed below. You may inspect the contents of this jar file by executing:
  
  ```bash
  jar -xf phase2.jar
  ```

- **src** which is a folder containing skeleton implementations for all of the Java interfaces and abstract classes that are described below. Your assignment is to add to that skeleton code and produce a secure authenticated key-value store.

- **build.sh** which is a script that compiles the contents of `phase2_impl` by linking it with `phase2.jar` and outputs a Jar file named `phase2_impl.jar`. 
• run.sh which is a script that executes your program by using the main method located in phase2_impl.jar as the entry point. Note that run.sh also performs the necessary RMI setup before executing Phase2App.java.
• README.txt which describes any changes you made to the build.sh or run.sh scripts that we provided.
• testRationale.txt which describes the functionality and/or defense that is being checked by each step of the test program in your Phase2App.java. Include as part of this discussion a listing of the outputs that a correct system should produce for the test program in Phase2App.java.

Grading. Project grades will be based on the following grading rubric

• 20% -- system operates correctly on tests you provided in Phase2App.java
• 20% -- how well the tests provided in Phase2App.java exercise the functionality and security of the system.
• 10% -- the quality of the explanations in testRationale.txt: Does testRationale.txt explain the checks in Phase2App.java and are all of the right things being checked.
• 40% -- system operates correctly on tests provided by course staff to exercise functionality
• 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.