CS5430: System Security (Fall 2023) Programming Project

Phase 1: An authenticated key-value store

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

This programming assignment should be implemented in Java version 11, which is available on the `ugclinux` computers.

Due Date. October 16, 2023 at 11:59 pm (in CMS)

Project Overview

An *secure authenticated key-value store* processes only those requests that have been issued for users who have previously registered; requests issued for other users are ignored. The system you will build for Phase 1 is intended to implement this functionality by extending the key-value store you implemented for Phase 0.

The system you build should be structured in terms of the following Java objects.

- **client**: This Java object (named `Phase1App.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 `Phase1StubImpl.java`) implements an interface named `Phase1Stub.java`. The client invokes methods defined by this interface in order to (i) register at the server, and (ii) perform key-value store operations at the server. The client stub invokes methods provided by the network in order to cause the invocation of a methods that the server is providing. *We provide a skeleton for `Phase1StubImpl.java`, and you add code to this skeleton in order to implement the required functionality of the client stub methods.*
- **network simulator**: This Java object (named `NetworkImpl.java`) simulates a network that connects the client stub to the server. `NetworkImpl.java` implements an interface named `Network.java`. *The code we provide implements a network simulator that delivers requests sent to the server by the client stub and delivers responses sent to the client stub by the server.*
- **server**: This Java object (named `Phase1ServerImpl.java`) authenticates each requested operation to ascertain that it was issued for a registered user. If a request authenticates then the server should perform the designated operation. `Phase1ServerImpl.java` implements an interface named `Phase1Server.java`. *We provide a skeleton for `Phase1ServerImpl.java`, and you add code to this skeleton in order to implement the desired functionality of the server methods.*

In the code we provide (and in the code you submit), each of these objects executes in a separate Java virtual machine, and Java's RMI (remote method invocation) is used to invoke their methods. For technical reasons, each of the objects could throw a `RemoteException`, so this exception is listed in the interfaces we give below for the various objects. Your code should not change any part of the interfaces that we are providing, and your code should not "throw" this exception.

Details are given below, but here is a high level description of how to use RMI and the network simulator that we are providing. To convey invocation requests from the client stub to the methods that the server implements, use the methods `handleAuthenticatedRegister` and `handleAuthenticatedDo` that are implemented by instances of the network simulator. In the client stub, an instance of the network simulator can be found in the field that is named `network` in `Phase1StubImpl.java`; this field will have been initialized before your code starts executing. Upon calling these methods, the network simulator will make the appropriate call to the indicated method that your server is implementing. Network simulator methods `handleAuthenticatedRegister` and `handleAuthenticatedDo` each will terminate (allowing execution of the client stub to resume) after the invoked server method returns a value.

As should be clear, you are not starting with a clean slate. Rather, the project requires that you add code to a given, existing skeleton. Like it or not, that's the situation you will encounter as a software developer. And it can be challenging to understand how an existing system works, why it is structured in some given way, and how to add code without corrupting that structure.

The threat. Your system should defend against Dolev-Yao attackers in the network who are seeking to subvert the integrity of the authenticated key-value store. The good news: No user is concerned with keeping values secret, so you need not protect the confidentiality of requests that clients issue and you need not protect the confidentiality of values in the authenticated key-value store.

To defend against Dolev-Yao attacks that subvert integrity of messages, use digital signatures on all messages that traverse the network. To generate and verify these signatures, use the methods in `Signature.java`. Use `KeyGenerator.java` to generate a key-pair called `pair` (of type `KeyPair` as defined in `Java.security.KeyPair`), as follows.

```
KeyPairGenerator keyPairGen =
      KeyPairGenerator.getInstance("DSA");
keyPairGen.initialize(2048);
KeyPair pair = keyPairGen.generateKeyPair();
```

(Note, Java uses the terminology of public key and private key for what we are calling a verification key and a signing key, respectively).

Initialization code that we are providing and that gets executed before your code starts will provide a signing key and a verification key that can be used for the server. These keys were generated using the code snippet given above. The keys are made available, as follows.

- The server's signing key will have been stored in a field called `signingKey` in `Phase1ServerImpl.java`. This key is 2048 bits, stored as a base64-encoded byte array.

- The server's verification key will have been stored in a field called `serverVerificationKey` in `Phase1StubImpl.java object`. This key is 2048 bits, stored as a base64-encoded byte array.

When testing the security of your system, the course staff will use a <u>different</u> implementation of `Network.java`. That implementation gives the necessary control for generating various forms of Dolev-Yao attacks. We encourage you either to extend `NetworkImpl.java` or to replace that network simulator with a different Java object that will give you the capability to perform various Dolev-Yao attacks. Be warned: if the system you submit does not work correctly with the vanilla `NetworkImpl.java` then your system will not work correctly in the grading environment.

Grading and Submissions

What we are providing. Download from CMS the zip file `Phase1.zip`. It contains:

- `phase1.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:
  ```
  jar -xf phase1.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 fill-in that skeleton code to produce a secure authenticated key-value store.
- `build.sh` which is a script that compiles the contents of `phase1_impl` by linking it with `phase1.jar` and outputs a Jar file named `phase1_impl.jar`.
- `run.sh` which is a script that executes your program by using the main method located in `phase1_impl.jar` as the entry point. Note that `run.sh` also performs the necessary RMI setup before executing `Phase1App.java`.

You should submit a zip file `Phase1Impl.zip` that contains the following.

- `phase1.jar`, which should be the exact same `phase1.jar` file that we provided to you. (We will check its MD5 hash).
- `src` which is a folder containing your implementations for all of the Java interfaces and abstract classes to produce a secure authenticated key-value store.
- `build.sh`, which is a script that the grader can invoke to compile your project. This can be a modified version of the `build.sh` that we provided to you, but it must link with the above `phase1.jar.`
- `run.sh`, which is a script that the grader can invoke to execute your project using `Phase1App.java` as the entry point. This can be a modified version of the `run.sh` that we provided to you.
- `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 `Phase1App.java`. Include as part of this discussion a listing of the outputs that a correct system should produce for the test program in `Phase1App.java` .

Grading. Project grades will be based on the following rubric

- 20% -- system operates correctly on tests provided in `Phase1App.java`
- 20% -- how well the tests provided in `Phase1App.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 `Phase1App.java` and are all of the right things being checked.
- 10% -- system operates correctly on tests provided by course staff to exercise functionality
- 30% -- system operates correctly on tests provided by course staff to exercise resilience against attacks
- 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.*

Interfaces to be Implemented

Detailed functional requirements are given below for each of the Java objects comprising your system.

client stub. The client stub interface provides methods that the client uses to request operations in the key-value store. The comments below describe how your implementation of each method should behave.

```
public abstract boolean registerUser(String userId) throws
RemoteException;
// Invoked on behalf of userId to initiate a session that allows
userId thereafter to issue authenticated operations.  Returns
true if the server registered the user in response to this
request;  returns false, otherwise.

public abstract boolean create(String userId, K key, V initVal, M
initMetaVal) throws RemoteException;
// Invoked on behalf of userId to perform create(K key, V
initVal, M initMetaVal) at the key-value store.  Returns true if
the server performed the requested operation;  returns false,
otherwise.

public abstract boolean writeVal(String userId, K key, V newVal)
throws IllegalArgumentException, RemoteException;
// Invoked on behalf of userId to perform writeVal(K key, V
newVal) at the key-value store.  Returns true if the server
performed the requested operation; returns false, otherwise.  If
```

```
the server indicates that writeVal was invoked when key does not
already exist then throw IllegalArgmentException.

public abstract boolean writeMetaVal(String userId, K key, M
newMetaVal) throws IllegalArgumentException, RemoteException;
// Invoked on behalf of userId to perform writeMetaVal(K key, M
newMetaVal) at the key-value store.  Returns true if the server
performed the requested operation; returns false, otherwise.  If
the server indicates that writeMetaVal was invoked when key does
not already exist then throw IllegalArgmentException.

public abstract V readVal(String userId, K key) throws
NoSuchElementException, RemoteException;
// Invoked on behalf of userId to perform readVal(K key) at the
key-value store.  Returns the value associated with key K if the
server performed the requested operation; returns null,
otherwise.  If the server indicates that readVal was invoked when
key K does not already exist then throw IllegalArgmentException.

public abstract M readMetaVal(String userId, K key) throws
NoSuchElementException, RemoteException;
// Invoked on behalf of userId to perform readMetaVal(K key) at
the key-value store.  Returns the metavalue associated with key K
if the server performed the requested operation; returns null,
otherwise. If the server indicates that readMetaVal was invoked
when key does not already exist then throw
IllegalArgmentException.

public abstract boolean delete(String userId, K key) throws
RemoteException;
// Invoked on behalf of userId to perform delete(K key) at the
key-value store.  Returns true if the server performed the
requested operation;  returns false, otherwise.
```

Register requests to the server are packaged as instances of an `AbstractRegisterRequest`, which is defined as follows. It would be the argument to the method `handleAuthenticatedRegister` of `NetworkImpl.java.`

```
public abstract class AbstractAuthenticatedRegisterRequest
implements Serializable {
  public String userId;
  public byte[] verificationKey;
  public byte[] digitalSignature;
}
```

Similarly, a request by the client stub for the server to perform any other operations is packaged as an instance of an `AbstractAuthenticatedDoRequest`, which is defined as follows. It would be the argument to the method `handleAuthenticatedDo` of `NetworkImpl.java.`

```
public abstract class AbstractAuthenticatedDoRequest<K extends
Serializable, V extends Serializable, M extends Serializable>
implements Serializable {
  public String userId;
  public DoOperation<K, V, M> doOperation;
  public byte[] digitalSignature;
}
```

where a `doOperation` is defined as

```
      public class DoOperation<K extends Serializable, V extends
      Serializable, M extends Serializable> implements Serializable {
        public K key;
        public V val;
        public M metaVal;
        public Operation operation;

     public enum Operation {
          CREATE, DELETE, READVAL, READMETAVAL, WRITEVAL,
WRITEMETAVAL;
     }}
```

You are being provided with the concrete Java classes `AuthenticatedRegisterRequest.java` and `AuthenticatedDoRequest.java` that each extends the abstract Java classes `AbstractAuthenticatedRegisterReqiest.java` and `AbstractAuthenticatedDoRequest.java` respectively. Feel free to edit these files in order to add additional fields and functionality to the concrete classes.

---

server. The code you add to `Phase1ServerImpl.java` should generate responses that are packaged as follows. You are being provided with the concrete Java classes `AuthenticatedRegisterResponse.java` and `AuthenticatedDoResponds.java` that each extends the abstract Java classes `AbstractAuthenticatedRegisterResponse.java` and `AbstractAuthenticatedDoResponse.java` respectively. Feel free to edit these files in order to add additional fields and functionality to these concrete classes.

For an invocation of the server's `AuthenticatedRegister` method, the response that is the value returned from the server's method should be an instance of

```
public abstract class AbstractAuthenticatedRegisterResponse
implements Serializable {
  public Status status;
  public byte[] digitalSignature;

  public enum Status {
    OK, UserAlreadyExists, AuthenticationFailure;
  }
}
```

It also would be the return type of the `handleAuthenticatedRegister` method of `NetworkImpl.java`.

For an invocation of the server's `AuthenticatedDo` method, the response that is the value returned from the server's method should be an instance of

```
public abstract class AbstractAuthenticatedDoResponse<K extends
Serializable, V extends Serializable, M extends Serializable>
implements Serializable {
  public DoOperationOutcome<K, V, M> outcome;
  public byte[] digitalSignature;
}
```

where a `DoOperationOutcome`is defined as

```
public class DoOperationOutcome<K extends Serializable, V extends
Serializable, M extends Serializable> implements Serializable {
  public K key;
  public V val;
  public M metaVal;
  public Outcome outcome;

  public enum Outcome {
    AUTHENTICATION_FAILURE, SUCCESS, NOSUCHELEMENT,
    ILLEGALARGUMENT;
  }}
```

`AbstractAuthenticatedDoResponse` also would be the return type of the `handleAuthenticatedDo` method of `NetworkImpl.java`.

The `phase1Server.java` methods enable the client stub to perform operations in the key-value store. These methods implement the following interface. The comments below describe how your implementation of each method should behave.

```
public AbstractAuthenticatedRegisterResponse
authenticatedRegister(AbstractAuthenticatedRegisterRequest
request)
      throws RemoteException;
//Invoked in response to a client stub register request on
behalf of user userId to register verificationKey.  Return
UserAlreadyExists if this userId is already registered with a
different verification key;  return AuthenticationFailure if the
register request cannot be authenticated; otherwise return OK.
```

```
public AbstractAuthenticatedDoResponse<K, V, M>
authenticatedDo(AbstractAuthenticatedDoRequest<K, V, M>
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; otherwise,
return one of SUCCESS, NOSUCHELEMENT, or ILLEGALARGUMENT
according to the semantics of the key-value store operations
defined in Phase 0.
```