Assignment 4: Encrypted Storage

This project is due on **Tuesday, March 13** at **11:59 p.m.**. Late submissions will be penalized by 10% per day. If you have a conflict due to travel, interviews, etc., please plan accordingly and turn in your project early.

This is a group project; you will work in teams of two or three and submit one project per team. Submissions by groups of size one or four+ will receive no credit unless an exemption is given by an instructor. Please find a partner as soon as possible. If you have trouble forming a team, post to Piazza's partner search forum. The final will cover project material, so you and your partner should collaborate on each part.

The code and other answers you submit must be entirely your own work. Undergraduate students are bound by the Honor System while graduate students are bound by the Graduate School's expectation of research integrity. You may consult published references, provided that you appropriately cite them (e.g., with program comments), as you would in an academic paper.

Solutions must be submitted electronically via CS Dropbox.

Introduction

In this assignment, you will implement a secure network storage facility, building on what you have done in the first three assignments.

As before, we will give you some of the code you need, and we'll ask you to provide certain functions missing from the code we provide. You can download the code we are providing here: https://goo.gl/uCbS3e. Create a fresh directory and unzip the downloaded code into it. Then copy it into the same directory all of the *.java* files from your solution to the previous homeworks reside in. As before, you must not use any crypto libraries; the only primitives you may use are the ones we gave you, and ones you implemented from scratch yourself. If you'd like to use our reference solution for the previous homeworks you may download it here: https://goo.gl/smLxuW. Note that this library does not contain source code.

In this assignment you will implement a secure file storage system. The system is based on three components: client-side software, server-side software, and an insecure block storage device.

Specifications

The client-side software runs on the user's computer. The client-side does most of its work by sending requests to the server-side software. The client offers the following API:

```
public class StorageClientSession {
    public StorageClientSession(String serverHostname, int serverPort,
                                String serverPublicKeyFilename);
    public void createAccount(String name, String password);
    public void authenticate(String name, String password);
    public void write(int nbytes, int storageOffset, int bufOffset, byte[] buf);
    public void read(int nbytes, int storageOffset, int bufOffset, byte[] buf);
```

The constructor connects securely to a server. *createAccount* creates a new user account on the server and sets up a username and password for the account. (This only succeeds if there is not already an account on the server with the same username.) *authenticate* logs into the server. Once the user is logged in, the write and read operations will write and read the storage space that the server maintains on behalf of the logged-in user. The server maintains separate storage space for each user.

To simplify your job, you may assume that there will not be more than 16 user accounts. So, for example, you can allocate 16 "slots" for storing information about user accounts, without having to worry that the number of accounts will grow beyond 16.

The server-side software accepts connections from one or more clients and does what is necessary to satisfy the clients' requests. The server-side software normally runs for a long time, awaiting connections from clients. The design of the server software is up to you.

Data stored on the server side on behalf of clients must be persistent, which means that it is not enough to store the data in the memory of the server (although you might choose to store it in memory and also elsewhere). The server is not allowed to use any of the Java library calls relating to files or persistent storage. The only type of persistent storage that your server can use is the insecure block device that we are providing.

The third facility we are giving you is an insecure block storage device. We are giving you a complete implementation of the insecure block storage device. You may not modify this. The block device stores fixed-size blocks of data, but it does not do anything to guarantee the confidentiality or integrity of the data it holds. You should assume that the insecure block storage device is controlled by an adversary. The one exception is that the storage device has a single "super block" which does offer confidentiality and integrity. Unfortunately the super block is fairly small.

Note that the block storage device, being under the control of the adversary, can destroy any data you store in it. Because of this, your solution is not required to provide availability. If the adversary trounces data in the block device, it is okay for your client and server side code to return errors. However, you must provide confidentiality and integrity.

The block storage device supports the following API:

```
public interface BlockStore {
    public void format() throws DataIntegrityException;

    public int blockSize();
    public void writeBlock(int blockNum, buf[] buf, int bufOffset,
               int blockOffset, int nbytes) throws DataIntegrityException;
    public void readBlock(int blockNum, buf[] buf, int bufOffset,
               int blockOffset, int nbytes) throws DataIntegrityException;

    public int superBlockSize();
    public void writeSuperBlock(byte[] buf, int bufOffset,
        int blockOffset, int nbytes) throws DataIntegrityException;
    public void readSuperBlock(byte[] buf, int bufOffset,
               int blockOffset, int nbytes) throws DataIntegrityException;
}
```

The block device is (conceptually) unlimited in size. format puts the device into a known initial state, where all blocks are filled with zeroes. (The implementation doesn't actually store all of the zero-filled blocks.) *blockSize* returns the size of a data block, and *writeBlock* and *readBlock* write and read a single block, respectively. Similar facilities operate on the superblock, giving the size and allowing reading and writing of the superblock.

Don't be overly concerned about efficiency. It's okay to "waste" a constant amount of storage, but your solution should be within a small constant factor of the optimal space requirement in the case where there are many users, each using a large amount of storage. Similarly, we won't mind if you copy data more times than necessary, but your running time should be at least be within a logarithmic factor of optimal in the many users, large storage case.

Although your solution will call on code that you wrote in the previous homeworks, we will test your solution with our own reference implementation. Your solution must work correctly when we do this — this shouldn't be a problem for you as long as you respect the API requirements of the previous assignments.

Getting Started

Tips This list may grow in response to Piazza questions.

- This assignment is more difficult than you might initially think. Please get started early.
- Most of the code and most of the design work is in the server-side code. The main function of the client-side is to establish a secure connection to the server, relay requests to the server, and collect the server's answers.
- Think carefully about how you are going to use the super-block. It's the only inherently secure storage you have, so you're going to have to build the security of everything else on top of confidential, integrity-protected information that you keep in the super-block.
- The server will need to keep track of metadata such as information about user accounts and passwords, and the size and layout of users' storage. This metadata won't fit in the super-block, so you're going to have to put most or all of it in the same untrusted block device where users' data lives. One way to manage this is to create a special "magic" user whose storage area contains the metadata — but be careful to avoid circular dependences where you need to read a metadata block in order to know how to find that same metadata block.
- In writing our sample solutions, we found it useful to build our server-side storage functionality in layers. Each layer implemented the *BlockStore* interface, and assumed the layer below it also implemented the *BlockStore* interface, thereby allowing us to stack up the layers in different orders. As an example, one of our layers took an insecure *BlockStore* and built on top of it a *BlockStore* that guarantees confidentiality and integrity of all the data it holds.

GROUP For this assignment, we require a GROUP file which contains the NetID of every partner in the group. Please format it as such:

```
NetID1
NetID2
NetID3
```

README For this assignment, we will also require a README file. In the file, you should describe your setup and your threat model: What are you doing? How to you know that your solution provides the necessary confidentiality and integrity properties? This should be in addition to any documentation you would normally put in comments or a README.

Submission Checklist

Upload to CS Dropbox the files listed below. Make sure you have the proper filenames and behaviors.

GROUP

`GROUP`	A file containing the group's NetIDs.

README

`README`	A file containing your description of your implementation.

ServerAuth

`ServerAuth.java`	A file containing your implementation of the *ServerAuth* class.

BlockStoreAuthEnc

`BlockStoreAuthEnc.java`	A file containing your implementation of the *BlockStoreAuthEnc* class.