University of Maryland
CMSC414 — Computer and Network Security
Professor Jonathan Katz

Semester Project

Part 1 (design): Due before class, Nov. 20
Part 2 (implementation): Due 11:59 PM, Dec. 12

The toy company Jolly Kid is making a giant overhaul of how its company operates and how its employees work. To facilitate on-site research, the toy company is implementing a policy that allows toy researchers with young kids to work from home. The researchers can then work and watch their kids at the same time. Not only that, the kids (who are too young to go to school), will play with the toys being developed and hence provide valuable feedback. The researchers who work at home will then submit the observation of the interaction between the child and the toy to the office for evaluation.

You, a senior computer/network security expert, were contracted by Jolly Kid to develop such a system. A high-level description of the system that Jolly Kid wants is below; however, the specifics will rely on your knowledge and experience.

1 Server Specification

There will be a server that mediates a secure (centralized) data repository. A user with a valid account will be able to log onto the server and, according to the user's level of privilege, retrieve or add data to the repository. The server shall comply with the following requirements:

1. **(State-Saving)** Routine system maintenance is unavoidable. This means that the Server should record its state when it is stopped running. When the Server is restarted again, it will resume from the last known state.

2. **(Directory System)** A user with a valid account shall have access to the Server. But they must use a client program. This Client will be freely distributed to the employees of Jolly Kid. Once a user logs onto the Server, he will be in his own home directory. The home directory for a user is the user's login name. For example, Bob's home directory is labelled "/Bob". (The "root" directory is labelled "/".) By default, users can own, read, and write their home directories. The server itself has own, read, and append rights to the root directory. To avoid creating confusion to some directory-challenged researchers, only directories one level deep need be supported (i.e., "/Bob/temp/" need not be supported.) In addition, only absolute directory names need to be supported (see command `cd` below for more detail).

3. **(Access Control System)** Jolly Kid is very concerned with confidentiality and requires both discretionary access control (DAC) and mandatory access control (MAC). Mandatory access control will be enforced by assigning each file or directory, and also

each user, one of the following security levels: **unclassified**, **classified**, **secret**, or **top-secret** (listed from lowest to highest level of privilege). Jolly Kid is concerned with users of lower privilege reading documents on a higher security level, and the Server must prevent this kind of access. (The Bell-Lapadula model has been suggested as one possible approach.) *Within each level of clearance*, discretionary access control is enforced by allowing the users to grant various rights on their files. The rights that must be supported are **read (r)**, **append (a)**, and **own (o)**. (A user can only access data if it is allowed by *both* the MAC and DAC mechanisms.)

4. **(Authentication)** The Server should allow valid users to log in. Jolly Kid wants a password-based scheme for authentication. But the details of the authentication are up to you, and Jolly Kid relies on your expertise to come up with a secure solution. However, if you believe that a alternate scheme (not based on passwords) is better, you may implement such a solution *in addition* to a password-based solution. You will receive a bonus (i.e., *extra credit*) for doing so.

5. **(Server-End Functionalities)** Jolly Kid claims that the Server will be placed in a secure dungeon guarded by corporate-minded trolls. Only trusted personnel will have access to the Server terminal. The Server should have a means for displaying critical system data and messages. In particular, the Server should have a debug mode. In debug mode, the server would display debugging messages in a specific format as described in the Appendix.

   In addition to the DEBUG mode, the Server should be able to accept keyboard commands defined as follows:

   (a) `exit` — commands the Server to shut down. If there is a live connection to a Client, display a message on the Client's side and terminate the Server program. Before termination, the Server should save state.

   (b) `states` — displays the current states of the system. This should include the list of current user names, current directories/files on the server, and any relevant authentication information. The data should be organized and concise.

6. **(Helpful Messages)** The Server should relay helpful messages to a Client for each requested action. For example, if a user does not have privilege to access a file in a certain way, then an error message should be displayed to the user's Client. Client-side error messages may be different from the messages being displayed on the Server side in debug mode. The messages should help the user understand what went wrong, but should not reveal "too much" information.

2 Client Specification

The Client also has some specifications that must be met. When run, the Client should allow a user to type various commands. Supported commands must include:

1. `cd` ⟨ `dirName` ⟩ — changes the current working directory of the user to `dirName`. You only need to support absolute pathnames (i.e., you can expect that `dirName` will be

an absolute pathname, not "..".) This means that if user Bob types `cd /Cathy` from his home directory "/Bob" then Bob will be working in the directory "/Cathy".

2. `create` ⟨ `filename` ⟩ — should create an (empty) file by that name in the current working directory. This should only be allowed if the user has "append" rights for the current directory.

   When a file is created, the default rights should be as follows: the creator of the file can own, read, and append the file. A user who owns the current directory can own, read, and append the file. All other users have no rights on the file. Make sure to check that a file by that name does not already exist in the current directory.

3. `delete` ⟨ `filename` ⟩ should allow user to delete a file from the current directory if the user owns it.

4. `upload` ⟨ `filename` ⟩ should upload a file from the (local) machine of the client to the current working directory on the remote server. Otherwise, this is treated like a `create` request (except the file is not empty).

5. `append` ⟨ `filename` ⟩ ⟨ `string` ⟩. If the user executing this command has append rights to the file, then `string` should be appended to the file.

6. `more` ⟨ `filename` ⟩ — should allow the user to read the file specified, if they have read access to the file. You may assume the files on the Server are ASCII text files.

7. `ls` — should list the contents (filenames) in the current working directory, only if the user has "read" rights for that directory.

8. `add o|r|a user name` — should give the specified user the specified rights on the specified file. (For example, typing `add oa phil filename` will give `phil` "own" and "append" rights for the file `filename`.) Here, `name` can be either a directory (specified using an absolute pathname) or a file. Only the owner of a file/directory can execute this command.

9. `sub o|r|a user name` — similar to the previous command, except that the specified rights are revoked. If you decide to give any warning messages when this command is executed, explain your decision.

10. `script` ⟨ `filename` ⟩ — should make Client read from the filename one line at a time, and execute each line as a valid command (sequentially). The filename should contain valid commands — one per line. Obviously, the file should reside on the Client side. (This command is also mostly for grading purposes.)

11. Please notice that it is the client's responsibility to make sure that the command entered by the user is a valid command. Also, the client should support a DEBUG mode as described in the Server specifications section. In debug mode, the client must display debugging messages in a specific format, as described in the Appendix.

3 Network Security Considerations

You should assume a passive (eavesdropping) adversary who can see *all* communication between Clients and Servers. Note that valid users might potentially be adversaries, so you should assume that an adversary can also establish an account, connect to the server, and send commands. However, you may assume that man-in-the-middle attacks are not a concern. Clearly, the authentication process needs to be secure. But you should also make sure that data is kept confidential from users and eavesdroppers that shouldn't have access to the data, and also that users who don't have the rights to modify files should not be able to do so. This will require more than just secure authentication...

4 Design Considerations

Some minimal issues that you should think about and address *before* implementation are listed below. In particular, each question should be amply addressed in your 1st deliverable.

- You should state how a user account is established. This may involve some out-of-band mechanism (go see system administrator, present ID, etc.). You may use any mechanism you can think of, but it should be secure.
- What other assumptions are you making when designing the system? Are they reasonable assumptions? Why?
- Does your system allow any covert channels? Where can they arise from your system? How do you prevent it from happening? Or what did you do to purge their existence?
- Does your system allow multiple owners to a file? What if there is a conflict in granting rights?
- What is the label of a file once it is being created by a user?
- How does your system address the question of attenuation of privileges?
- What kind of access policy do you choose to implement MAC?
- How do you implement access control? What type of data structure did you choose? Can you justify the space usage/access time it requires?
- What type of authentication scheme do you choose to implement? Why? Why do you think it is secure (and to what extend)?
- Can a malicious user use the Client to cause buffer overflow via any of the legitimate commands? How do you prevent it from happening?
- How do you implement the `sub` command?
- How does your system discourage on-line or off-line password-guessing?
- Where do you store username/password pairs? How do you store then? Is it secure? Defend your reasoning.

5 Deliverables

Jolly Kid requires 2 deliverables. The first will be an overall system design, and the second will be the implementation. In the first deliverable, you should outline what your system will accomplish. You should discuss all of the issues above. You should be specific in discussing how your authentication will work, how your access control will work, any cryptographic mechanisms you choose to use, and where you use it. You should discuss whether your system requires any support files to be stored in either the Server side or the Client side. You should justify their usefulness. You should describe how to implement the Server to save state upon exit. You should also defend the security of your system and impress the Jolly Kid executives. This system design should meet all the requirements Jolly Kid has set forth. The more detailed and complete this description is, the more pleased the executives would be. You should discuss why you opt to use one specific mechanism instead of another and in the interest of what (e.g. security, efficiency, etc.).

The second deliverable will consist of the actual programs. There will be 2 core programs: a Server and a Client. They should meet the specifications described in your first deliverable. You should submit a working server program and a working client program. (It will be better to submit a program that implements a subset of the above, rather than submitting a non-working program.)

6 Submission Guidelines

1. **(Design)** The 1st deliverable should be a well-prepared documentation that outlines what you will do to achieve the required functionalities. The documentation should be submitted as a printed hard copy.

2. **(Implementation)** The 2nd deliverable should consist of 2 directories. One contains the Server, and the other one contains the Client. They should be named, respectively, Server, Client. So when we unzip and untar your submission, we should see these 2 directories. Of course, any system-dependent files will be in its corresponding directories. The main routine that runs Server should be named Server.java. The main routine that runs Client should be named Client.java.

   To launch the Server, the grader would type

   ```
   java Server port-number [acfile psswdfile] [-DEBUG]
   ```

   Where arguments in "[ ]" are optional. "acfile" is the name of the access file from which the Server obtains the access control information (users, privileges, objects, etc.). "psswdfile" is the name of the password file from which the Server obtains authentication information (user/password pairs). "[acfile psswdfile]" means that they either both are supplied in the command line or both are not. "-DEBUG" sets the Server to be running in debug mode. For detailed description, please see the section on *Server Specification*. Here is how an access control file would look like:

(Users)	*(Clearance)*	TVguide	PCShopper	JetManual	MagicScroll	lyrics	temp
	(Label)	U	U	S	TS	C	U
Homer	U	or	ora	n	r	n	r
Artanis	TS	r	n	roa	n	ora	n
cLOUD	S	n	n	n	ra	n	n
Spike	S	n	n	r	r	r	n
ed	C	ra	ar	ra	n	a	n

Each subject has a clearance level. Each object has a security label. The entries denote the DAC access of each subject to each object. (r: read. a: append. o: own. n: none). In the file form, the entries in *()* are assumed to be empty. Moreover, the vertical and horizontal lines do not exist (this applies to the password file as well).

A password file would look like:

username	*password*
Homer	doh-nuts
Artanis	ThisIsNot3D
cLOUD	dJ3xjmm2
Spike	xIMycow
ed	EinTheDog

Notice that both the access control file and the password is in clear text. These files are always used in initiating a Server. However, for security reasons, you may not want to store one/both of them as clear text on your system (You must justify your choices). This also means that if without initialization, your system will need to deal with files other than clear text.

To start the Client, the grader would type

```
java Client CPortNum ServerMachine SPortNum username password
```

where CPortNum is the client port number and SPortNum is the server port number. If you implemented another authentication scheme besides passwords, you may add more arguments at the end as needed. In this case, you must document clearly on how we can authenticate ourselves to the Server as valid users, otherwise we cannot grade it.

3. **(Makefile)** You should also submit a Makefile. At a minimum, we should be able to type `make all` to compile all of your programs and to type `make clean` to remove all .class files as well as any temporary files created by the Server or the Client.