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

Problem Set 4

Due by Dec. 8, 11:59 PM

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

1 Overview

You are to implement two programs: a client and a server. These will be submitted in two directories named (appropriately) "Client" and "Server", one containing the client (and any necessary files) and one containing the server (and any necessary files). The main routine that runs the server should be named Server.java, and the main routine that runs the client should be named Client.java.

The server will be launched using the command:

```
java Server port-number [init-file] [-DEBUG],
```

where the -DEBUG flag is optional. In the above, `init-file` is the name of a file — stored at the server — which contains information about the employees who are allowed to log-in remotely. The format of this file can be whatever you like. However, you must also have a program `init` which creates an appropriate `init-file` based on an ASCII-text file containing information about the employees. This program will be called as follows:

```
java init [infile] [init-file],
```

where `init-file` is the same as mentioned earlier, and `infile` is an ASCII file containing information about employees, one per line, in the following form: [username] [clearance] [password]. In this file, usernames and passwords are alphanumeric strings of length at most 32 characters and the clearance is one of "unclassified", "classified", "top-secret" (from least to highest clearance).

(To summarize: `infile` is a file containing information about the various employees, set up by a sys-admin. Calling `java init infile init-file` creates a file `init-file` which represents what is actually stored on the server. When the Server is run for the first time, it only looks at `init-file`.)

The client is run using the command:

```
java Client CPortNum SPortNum username password,
```

where `CPortNum` is the client port number and `SPortNum` is the server port number.

You should also make sure to include 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.

2 Server Specification

The server should satisfy the following requirements:

State. Routine system maintenance is unavoidable. The Server should record its state (i.e., the files in the system) when it is shut down. When the Server is restarted it should resume from the last known state, if present.

Directory structure. When a user logs in to the Server using the Client program (and his correct password), he will be in his home directory which is the user's login name; for example, Bob's home directory is labeled "/Bob". (The "root" directory is labeled "/".) You do not need to support sub-directories below user-level directories.

Once a user has logged in, they may read and write to files as described in the section detailing the client specification. These commands will be issued at the client side and transmitted to the server who will then process them accordingly.

Responses to client commands. The Server should relay useful diagnostic 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 (cf. below). The messages should help the user understand what went wrong, but should not reveal "too much" information.

Access control. Jolly Kid is concerned with confidentiality and has decided to use mandatory access control and, in particular, the Bell-La Padula security model. Any file in a given user's directory is classified according to the security clearance of that user. Thus, for example, the file "/Bob/foo" is considered "classified" if Bob's security clearance is "classified".

Authentication. The client software allows users to log-in to the server, but clearly this should be done in some secure fashion. (See further below for exact security requirements.) Jolly Kid has decided to use a password-based scheme, but the details of the authentication mechanism are entirely up to you. *Note:* since Jolly Kid has an educated workforce, you may assume that all employees use only high-entropy passwords, and in particular you do not have to worry about off-line dictionary attacks. The server only needs to support a single client logging in at a time.

Server-side diagnostics. When run in debug mode (i.e., using the -DEBUG flag), the server should display diagnostic messages as described in the appendix of this document.

Additional commands. The Server should accept the following local commands (i.e., not issued by a client but typed directly at the server console):

- `exit` — commands the Server to shut down. If there is a live connection, display a message on the Client side. Before termination, the Server should save state.

- `states` — displays the current state of the system. This should display which user (if any) is currently connected as well as the current list of directories/files on the server.

3 Client Specification

Once a user has logged in to the server using the Client software, they should then be able to issue the following commands:

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. Make sure to check that a file by that name does not already exist in the current directory.
3. `delete` ⟨ `filename` ⟩ allows the user to delete a file from the current directory.
4. `upload` ⟨ `filename` ⟩ should upload the file `filename` from the (local) machine of the client to the current working directory.
5. `append` ⟨ `filename` ⟩ ⟨ `string` ⟩. The text `string` is appended to the specified file.
6. `more` ⟨ `filename` ⟩ — allows the user to read the file specified. You may assume the files on the Server are ASCII text files.
7. `ls` — should list the contents (filenames) in the current working directory.
8. `script` ⟨ `filename` ⟩ — allows Client to read from the (local) file `filename` one line at a time, and execute each line as a valid command (sequentially). The filename should contain valid commands — one per line. (This command will be used for grading purposes, and may also simplify the process of entering commands.)

Access control. Commands `create`, `delete`, `upload`, and `append` are only allowed if the user has "write" access to the current working directory. Commands `more` and `ls` are only allowed if the user has "read" access to the current working directory.

4 Network Security Considerations

Jolly Kid does not want any non-employees to learn about the new toys they are developing. The protocol you develop should be secure against an adversary who is assumed able to (1) run client impersonation attacks (and, in particular, is assumed to know the usernames of all employees in the system); (2) eavesdrop on *all* communication between the Client and Server; and (3) try to mount a "session hijacking" attack in which the adversary waits for a user to connect to the server and then tries to send commands of its own. You must also ensure that the protocol is secure against malicious employees (e.g., an employee with clearance “unclassified” trying to read a file marked “top-secret”.) 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…

You may assume that man-in-the-middle attacks and server impersonation attacks are not a concern. Also, as mentioned earlier, you may assume that only high-entropy passwords are used and so you do not need to protect against off-line dictionary attacks. (However, if you can do so then so much the better! And if you have a nice solution that protects against off-line attacks, you may get extra credit…)

5 Deliverables

In addition to any software you turn in, you should also include a document describing your solution. At a minimum, this document should contain (1) a description of the authentication protocol you use; (2) a discussion of how your system protects against eavesdropping, client-impersonation, and session-hijacking attacks; and (3) any design decisions you made in the course of developing your protocol. (Give as much detail as necessary for the TAs to understand what you did: they are not going to be reading your code to figure out what authentication mechanism you implemented.) In addition, you may want to touch upon some or all of the following issues:

- Why did you choose the authentication protocol you did? Did you consider any other options? Did these other options offer any advantages?
- Does your authentication protocol happen to be secure against server impersonation and/or man-in-the-middle attacks anyway (even though not required)?
- 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?
- How do you implement access control? What were the design decisions here (if any)? Could you easily extend your system to allow for discretionary access control?
- Can a malicious user running the Client software cause a buffer overflow at the Server? How do you prevent this from happening?
- Does your system discourage on-line or off-line password-guessing?
- What is the format of your `init-file`? Does this offer any security advantages?

6 Appendix: Debug Message Formats

When running in DEBUG mode, the server should return the following after every command issued by the client:

```
username command status
```

where `username` and `command` are obvious, and `status` is as follows for each command:

1. `cd` ⟨ `dirName` ⟩ — return “success” if the command is executed with no error. Return “invalid directory name” if the directory does not exist. Return “permission denied” if the user does not have permission to access the directory.

2. `create` ⟨ `filename` ⟩ — return “success” if the command is executed with no error. Return “file already exists” if a file with that name already exists in the current directory. Return “permission denied” if the user does not have permission to invoke this command.

3. `delete` ⟨ `filename` ⟩ — return “success” or “permission denied” as above. Return “file does not exist” if the specified file does not exist in the current directory.

4. `upload` ⟨ `filename` ⟩ — return “success” or “permission denied” as above. Return “file already exists” if a file with that name already exists in the current directory.

5. `append` ⟨ `filename` ⟩ ⟨ `string` ⟩ — return “success” or “permission denied” as above. Return “file does not exist” if the specified file does not exist in the current directory.

6. `more` ⟨ `filename` ⟩ — return “success”, “file does not exist”, or “permission denied” as above.

7. `ls` — return “success” or “permission denied” as above.