CS361 Assignment 1

Due: Monday, June 12 by the end of the day. This means that your submission must be dated on or before that date, so you have until midnight. I'm giving more time on this assignment than will be typical. Don't put it off, because it may overlap with the next assignment.

Each assignment is graded on a 10 point scale.

Deliverables: You will be submitting your code electronically using Canvas. Your program must be able to run on the UT Linux machines, so if you develop it elsewhere, make sure it runs on Linux.

Don't forget that you'll be zipping your Java implementation files and README.txt file together and submitting a single zip file for the main program. Don't forget to name your zip files according to the instructiions in General instructions.

Grading README file: Your zip file must contain a README.txt file. We're using a program to grade your README.txt file, so make sure that you follow the directions precisely. Any other names or formats of REAMDE file won't be accepted, e.g. readme.txt, readme, README, README.c, README.java. The README.txt file is worth 2 (of the 10 points). If you use the incorrect REAMDE.txt file, you will have to email the TA to handle this issue.

Here's a sample README file for Assignment 1: README.txt. You're strongly advised to follow this model.

For the README.txt file, the maximum point is 2.

1. First Five Lines [0.25 point] have to match the format that has been explained in General instructions..
2. [Both Part 1 and Part 2] Good description of your code [0.25 point]

   Write a few sentences (at least 100 words) to explain the high-level idea of your program along with short descriptions of the key/important function in your code.
3. [Both Part 1 and Part 2] Tell us how much you've finished [0.25 point]

   (1) Finished all requirements (2) Only finished parts of the requirements.

   If you're in the case (2), you have to list the unfinished things and write down the reasons for each, e.g., you ran out of time or you encountered some bugs. If grader thought you explanation is very good and detailed, you will get some credit (0.5 point) even if you didn't finish some of the program requirements.
4. [Part 1 only] Create at least one more test case [0.5 point]

   First, include one sentence how many test cases you created. If you have >=2 test cases, you will get 0.5 point. Second, attach all your own test cases and the output from them at the end of README.txt file. (If the test cases or outputs are too large, you can give separate *.txt files by indicating the file names in your README.txt). Third, you don't need to create any bizaare or tricky test cases. Also, a single line cannot be regarded as a test case. It is enough that each test case has 5~20 lines. One of the test cases has to be the example on this website.
5. [Part 2 only] Machine information [0.25 points]

   What is your machine type? What is the clock speed? Is that a Linux, Windows or Mac?
6. [Part 2 only, Results summary] [0.5 points]

You need to provide a table that contains at least 2 test cases in README.txt file. Here is an example of README.txt for assignment 1. Two files have to be ``Pride and Prejudice" and ``Metamorphosis".

The grading scheme of this README.txt file has very little to do with the correctness of your program. Even if your program doesn't work well, include a well-formatted README.txt file for partial credit.

Grading Program: For the program itself, for each part, you can earn a maximum of 4 points. In total, it is 8 points.

1. You will get [1 points] if the program can be compiled. Before submission, make sure that your code can be compiled and executed on the Linux Machine (in our CS Lab). Even if your code is working on your own machine (Windows, ...), but not working on CS Lab machine, you won't get any points!

2. You will get [2 points] If the program can be complied and run.

3. You will get [4 points] If your output is reasonable.

The Assignment

This assignment has two main parts:
1. implement a simple BLP secure system;
2. extend the system and implement a covert channel.

Part 1: A Simple Secure System

For Part 1:
1. The primary Java file name should be `SecureSystem.java`. Students are welcome to organize their assignments into multiple files and should submit those files as well. However, the main method must be in `SecureSystem.java` and we should be able to compile your program with `javac *.java`.

2. The program should be executed via the command: `java SecureSystem instructionList`

3. Include a README file as specified in General instructions.

In this first part, you will implement in Java a simple "secure" system following the Bell and LaPadula (BLP) security rules--simple security, the *-property, and strong tranquility.

Many years ago, the Anderson Report suggested the following architecture for secure access control systems: Segregate the subjects and objects into separate domains. All accesses by subjects to objects must be mediated by a trusted system component called a "reference monitor." The reference monitor has the following two properties:

1. it should mediate all accesses by subjects to objects;
2. it must be non-bypassable;
3. it should be simple enough to be verified by inspection.

You should implement your system following this model.

Create a system with subjects and objects implemented as separate classes: Subjects and ObjectManager, respectively. You can think of ObjectManager as a simple file system. Each object has fields: NAME and VALUE (an integer, initially 0). Each subject has fields: NAME and TEMP. TEMP holds the value most recently received from a READ operation (also initially 0). Subjects and objects do not store their own security levels. For Part 1, you will pre-populate the system with a collection of subjects and objects. That is, it will not be possible to create subjects/objects on the fly.

The reference monitor (RM) maintains all information about the levels of subjects and objects. When you create a subject or object, inform the RM about its level and the level subsequently never changes (strong tranquility).

Subjects can request to perform READ or WRITE operations on objects. These are submitted to the RM which decides whether to allow the operation, according to the BLP rules.

For a READ, the subject reads the current value of the object, and saves that value into its TEMP variable (a subsequent READ will smash it). If the operation is not allowed, a zero is returned to the subject.

When a subject does a WRITE, the object's value is updated, if the operation is allowed by the RM. Otherwise, no change is made to the object.

The input to your system is a file of *commands*. For Part 1 of this assignment, legal commands are of the form:

```
READ subject_name object_name
WRITE subject_name object_name value
```

All fields of the instruction are strings except the value, which is an integer.

For Part 1, you should read successive lines from an input file and parse each. *Commands are not case-sensitive, even object and subject names*. Arbitrary whitespace is allowed in commands, though you can assume that each command is on a single line.

Suggestion: Define an InstructionObject class and parse each input line into an object of the class. It is the InstructionObject that you pass to the RM for the security check. Any instruction that fails to parse could then generate a constant BadInstruction object of that class. The reference monitor will always reject a BadInstruction object.

Be sure to deal with the possibility of errors in the instructions (neither READ nor WRITE, wrong number/type of arguments, references undefined name, etc.). For example, suppose that "hal" and "lyle" are the names of defined subjects and that "lobj" and "hobj" are the names of defined objects. Given the following list of instructions, the first three and the last three are bad and the others are syntactically legal.

```
write hal hobj
read lal hobj
read hal
write lyle lobj 10
read hal lobj
write lyle hobj 20
write hal lobj 200
read hal hobj
read lyle lobj
read lyle hobj
foo lyle lobj
Hi lyle, This is hal
The missile launch code is 1234567
```

Notice that, at this stage, you're not checking security attributes or definedness of the subjects/objects (yet), merely the syntactic well-formedness of the commands.

Now, let's add some security to our system. Start by giving both subjects and objects associated security labels (levels). These labels are maintained by the RM and can't be changed after they are created (strong tranquility).

In the secure version of our system, whenever a subject requests to perform an action (READ or WRITE), the parsed InstructionObject is submitted to the RM, which decides whether to perform the action or not

based on the BLP properties (Simple Security and the *-Property). If the instruction is both syntactically legal and is allowed by the BLP rules, the reference monitor tells the ObjectManager class to perform the appropriate action; otherwise, no objects are accessed.

Assuming an instruction is not Bad, *the reference monitor always returns an integer value to the subject: the value of the object read, if the command was a legal READ; and 0 otherwise*. If the subject is performing a READ, it stores this value into its TEMP variable. Think of the reference monitor as a firewall around the ObjectManager. Note that the ObjectManager itself doesn't know or care about labels or security; it just performs simple accesses.

The top-level class (SecureSystem) manages subjects and the reference monitor, and also serves as the command interpreter. It reads successive instructions from the instruction list, parses them, and submits them to the RM, which asks the ObjectManager to perform them (or not). The value returned by the reference monitor is passed to the subject executing the instruction.

Your task in Part 1 is to implement all this, subject to the following constraints.

1. The ObjectManager should be local to the ReferenceMonitor. That ensures that you can't access objects except via the ReferenceMonitor interface.
2. SecurityLevel should be a class with a defined "dominates" relation. You can assume that levels are linearly ordered. That is, you don't need to worry about need-to-know categories. Within that class define two constant levels HIGH and LOW such that HIGH dominates LOW.

To see how your system is behaving, you should write a debugging method printState in the SecureSystem class that prints out current values from the state: the value of objects and the TEMP value for subjects. See the example below for what the output of this should look like. For this assignment you can assume that there are only two of each and you know their names. (Note: printing this kind of information is not something that a typical system user should be able to do.)

The main function in your SecureSystem class should perform the following tasks:

1. Create two new subjects: lyle of security level LOW, and hal of security level HIGH. Store these subjects into the state and inform the reference monitor about them.

2. Create two new objects: lobj of security level LOW, and hobj of security level HIGH. Store these in the ObjectManager, telling the ReferenceMonitor about their levels. The initial value of each should be 0.

3. Read successive instructions from the input file and execute them, following the Bell and LaPadula constraints on reading and writing. The ReferenceMonitor class checks access requests and perform the appropriate update (if any) on the state, following the instruction semantics outlined above.

4. After each instruction, call printState to display the state change, if any, from the instruction execution.

As an example of the execution, given the instruction list above, my implementation gives the following output. *Note: none of this output would ordinarily be visible to hal and lyle*.

```
> java SecureSystem instructionList

Bad Instruction
The current state is:
   lobj: 0
   hobj: 0
   lyle read: 0
   hal read: 0

Bad Instruction
```

```
The current state is:
   lobj: 0
   hobj: 0
   lyle read: 0
   hal read: 0

Bad Instruction
The current state is:
   lobj: 0
   hobj: 0
   lyle read: 0
   hal read: 0

lyle writes 10 to lobj
The current state is:
   lobj: 10
   hobj: 0
   lyle read: 0
   hal read: 0

hal reads lobj
The current state is:
   lobj: 10
   hobj: 0
   lyle read: 0
   hal read: 10

lyle writes 20 to hobj
The current state is:
   lobj: 10
   hobj: 20
   lyle read: 0
   hal read: 10

hal writes 200 to lobj
The current state is:
   lobj: 10
   hobj: 20
   lyle read: 0
   hal read: 10

hal reads hobj
The current state is:
   lobj: 10
   hobj: 20
   lyle read: 0
   hal read: 20

lyle reads lobj
The current state is:
   lobj: 10
   hobj: 20
   lyle read: 10
   hal read: 20

lyle reads hobj
The current state is:
   lobj: 10
   hobj: 20
   lyle read: 0
   hal read: 20

Bad Instruction
The current state is:
   lobj: 10
```

```
   hobj: 20
   lyle read: 0
   hal read: 20

Bad Instruction
The current state is:
   lobj: 10
   hobj: 20
   lyle read: 0
   hal read: 20

Bad Instruction
The current state is:
   lobj: 10
   hobj: 20
   lyle read: 0
   hal read: 20
```

I would suggest that you try to mimic this format exactly. It will make it much easier for the TA to grade with a script.

Part 2: Adding a Covert Channel

In Part 2, your task is to update your secure system from Part 1 and add three new operations designed to introduce a covert channel into the system. You will implement the channel and use it to signal information from high level user hal to a low level user lyle. Finally, you will measure and report the bandwidth of the channel.

For Part 2:

1. The primary file name should be `CovertChannel.java`. Students are welcome to organize their assignments into multiple files and must submit those files as well. However, the main method must be in `CovertChannel.java` and the TA should be able to compile your program with `javac *.java`. As usual, these will be zipped with your README.txt file for the submission.

2. The program should be executed via one of the following commands:

   ```
   java CovertChannel v inputfilename
   java CovertChannel inputfilename
   ```

   Your output file should be named inputfilename.out. That is, take the supplied input filename and append ".out" to it. If the call has the "v" (for "verbose") option, you should also output to a file called "log" the instructions that you generate. (See below.)

   Update your secure system from Assignment 1 to add three new instructions:

   ```
   CREATE  subject_name object_name
   DESTROY subject_name object_name
   RUN     subject_name
   ```

   Again, instructions are NOT case-sensitive in any part.

   The semantics of CREATE is that a new object is added to the state with SecurityLevel equal to the level of the creating subject. It is given an initial value of 0. If there already exists an object with that name *at any level*, the operation is a no-op.

   DESTROY will eliminate the designated object from the state, assuming that the object exists and the subject has WRITE access to the object according to the *-property of BLP. Otherwise, the

operation is a no-op.

RUN allows the named subject to execute some arbitrary private code. *It has no access to any of the state objects, and so should be irrelevant to the security of the system*. It models whatever processing the subject may do with the value it has just read. You should code this as a method within your SecureSubject class. For this assignment, the point of RUN is to allow Lyle to do whatever processing is necessary to input the value from Hal, add it to a byte he's creating, and if the byte is complete, write it to the output. *Since Hal will never need to execute RUN, you can make its functionality specific to Lyle*.

As with Part 1, you will pre-define two subjects Hal and Lyle, but no objects initially. You will generate the instructions necessary to pass information from Hal to Lyle using the covert channel we described in class. (See below.) However, generate the instructions to be executed on the fly rather than reading them from an external file. That means that your program generates instructions and executes them as needed. There is no file of instructions. (If you generated them, wrote them to a file, and then read them back in your performance would suffer quite badly from the extra IO.)

The execution of the instructions you generate should implement a covert channel passing information one bit at a time from Hal to Lyle. You can assume that Hal and Lyle are the only two subjects and have those names. Also note that you can eliminate most parsing and syntax checking since you should generate only syntactically correct instructions.

If your program is running with the "v" parameter, log the instructions that you generate one per line to a file called "log". This will allow us to ensure that you are actually using the covert channel to transfer the information. For your timing runs, don't use the "v" parameter.

Implement the following covert channel. For this channel, to send a 0 bit Hal executes:

```
CREATE HAL OBJ
```

To send a 1 bit, Hal does nothing. Lyle senses the bit (the presence or absence of the object) by executing:

```
CREATE LYLE OBJ
WRITE LYLE OBJ 1
READ LYLE OBJ
DESTROY LYLE OBJ
RUN LYLE
```

If Lyle sees a value of 1 returned from the READ, then his CREATE succeeded, and Hal has not previously created the object (sending a 1 bit). If Lyle sees a value of 0 returned from the READ, then Hal has previously created the object, Lyle's CREATE has failed, and Hal has sent a 0 bit. (Notice that the value of 1 in the WRITE could have been any non-zero value.)

Lyle's RUN statement allows him to do whatever he has to do to record the bit into his internal state, add it to the byte he's creating, and output the byte if he's received the 8th bit for that byte. You definitely need RUN for Lyle. *You're not allowed do this stuff in the top-level routine*.

The goal is to send an arbitrary stream of bits (actually a complete ASCII file) over this channel. I advise getting your program running by sending some small fixed number of bits and making sure they arrive correctly. However, the version you will submit will take a filename parameter on the command line. Your program will read the file contents (probably one byte at a time using a ByteArrayInputStream), convert each byte to 8 bits, send them through the covert channel, reconstruct the byte on the receiving side and write it out to a file. The idea is to transfer the contents of an arbitrary ASCII file over the channel.

As an alternative to using a ByteArrayInputStream, you can read a line in the file into a String and access the bits from there. *Don't read the entire file into a String*. That's not scalable.

Your program may work for non-ASCII file types, but doesn't have to. In a realistic system where the subjects were running as concurrent threads, that would be very useful because Lyle would have to know when he can stop receiving. If you only worry about ASCII files, you can assume that only ASCII characters appear in the file. That means you can use a non-ASCII character to signal the end of the input. E.g., if Lyle receives a null byte 00000000, then he knows he can stop receiving. But for this version, your main program probably knows when Hal is out of bits to send and just stops generating instructions for Lyle. That is, you probably don't have to worry about sending an end byte from Hal to Lyle. *I think that's true, but I'm not absolutely certain!*

Ideally, reading bytes from the file, breaking them into bits, etc. should be done by the subjects themselves using their RUN operation. But just use your main function to handle Hal's portion of that kind of thing. That is, you can treat the entire program as a big covert channel engine, feeding Hal one bit at a time, as long as you implement the covert channel according to the directions above and don't cheat.

The output file should be essentially identical to the input file. Make sure to check that you preserve the first and last characters in the file. I suggest using the Linux diff utility to compare the files when debugging.

Note that line endings are represented differently in different systems. Early teletype printers used the CR and LF control characters, respectively, to move the print head left and then scroll down one line. But line termination is treated differently by different modern operating system: Macintosh uses a CR (ASCII 13); Linux uses a LF (ASCII 10); Windows uses both. Most FTP programs substitute appropriate line endings if transferring in ``text mode.'' If in ``binary mode'' no translations occur. This shouldn't matter to you unless you transferred your input file onto a different system in binary mode. In that case, your program might not recognize the end of lines correctly. In previous semesters, some students' programs didn't handle the end of line characters correctly so a file with line breaks ended up as one long line.

Finally, time your program execution on several different input files and compute the channel's average bandwidth in bits / microsecond. Your program should compute and write to standard output the timing and bandwidth for a given run. Also, record that information for several runs, compute the average, and include that information along with the machine type and clock speed in the README.txt file for your program. You won't be graded on the bandwidth, but we'll collect the results and see which group's program was the most efficient.

For a ballpark, one group in a recent class reported the following bandwidths for different documents:

Document	*Size*	*Bandwidth*
Pride and Prejudice	**717,571 bytes**	**600 bits/ms**
Metamorphosis	**141,411 bytes**	**1126 bits/ms**
Test	**45 bytes**	**40 bits/ms**