Homework 4: Monitoring and Prioritizing Security Events

Due **Wednesday November 23 2:00pm** via InstructAssist (*we are giving you until Wednesday since we know many students will be traveling on Tuesday for Thanksgiving*).

This week we pull together several concepts we've covered so far to implement the core of a security-monitoring system.

[*Credits:* The infrastructure for this assignment was developed by course SAs Alex Pauwels and Andrew Rottier]

Monitoring what users do on a system is a key part of modern security management. A user could try to access a file that they shouldn't need to look at, or someone might have several failed login attempts. Either one of these cases (or others) should alert someone to check for a potential security breach. Unfortunately, alerts can come in faster than the information technology team can process them. Security monitoring systems therefore also *prioritize* the alerts based on the severity of the detected issue.

At a high level, you have three tasks on this assignment:

- Write classes and methods that look for specific patterns in the actions that users took on a system, and generate objects that report alerts when suspicious patterns occur. We will give you a `LinkedList` of user actions of various kinds (all generated by our starter code). The questions below explain what alerts and patterns should look like.
- Store alerts in an instance of Java's `PriorityQueue` class, so that highest-priority alerts are readily accessible.
- Think about using tests to explore data structures when you don't have access to the internals of those data structures.

We are giving you a lot of infrastructure code this week. In part, this lets us do a (hopefully) more interesting exercise than what you could write on your own. In part, this gives you practice reading existing code and adding to an existing project.

*Unless we tell you otherwise for a specific file, we do **not** expect you to read or understand all the code we provided*. Some of it uses material we haven't discussed yet. You should only need to look at the contents of the following files:

- EventLog.java (not strictly needed)
- Examples.java
- IPattern.java
- SecurityMonitor.java
- SuspiciousWebPattern.java

In addition, the `NameCheck.java` file is the usual tester that you have your class names and constructors as expected by the autograder (we added this to the starter zip on

11/16 at 10pm – if you downloaded before that, you can get the file directly here).

Here are the starter files. We'll explain what you need from these as we go.

As of this week, you should put each class in its own separate file. This is proper Java coding practice.

1 Creating an Alert class and Prioritizing Alerts

- Create a class called `Alert`, which will track security warnings. This class needs three fields: the username of the person with suspicious behavior (a string), the severity rating (an integer from 1 to 10), and the type of the event (which will be an integer between 1 and 6 – details on what these integers mean are provided below).

- Our monitor is going to need to be able to detect when one security alert is more critical than another (so that staff handle high-priority cases first). Several built-in classes in Java benefit from knowing when one object should be considered "smaller" or "larger" than another (such as when using a sorting routine). Java therefore has a default method name called `compareTo` that indicates the "order" between objects.

  Annotate your `Alert` class to implement a built-in interface called `Comparable`. The interface requires a method `compareTo` that takes one input of type `Object` and returns an integer. The sign of the returned integer indicates which of `this` object or the given object should be considered "smaller" (and, in our case, given priority in the queue).

  Specifically, your `compareTo` method should return:

  - 0 if the two alerts are equally severe
  - -1 if this alert is more severe than the given alert
  - 1 if this alert is less severe than the given alert

  `compareTo`, like the `equals` method we discusssed last week, is a method name that gets standardized across classes and is expected to handle comparing any object to any other. As we showed in the notes for `equals`, we need to tell Java explicitly that we expect to only compare alerts to one another. You do this by starting off your `compareTo` method as follows (the `Alert other ...`) line is how we tell Java to trust us that `otherObj` will be an `Alert`.

```
public int compareTo(Object otherObj) {
    Alert other = (Alert) otherObj;
    // insert the code to do the comparison here
}
```

- When we get to testing, you will also find it handy to have a `equals` method in your `Alert` class. Go ahead and add that now. Remember that this method takes an object and returns boolean, indicating whether this and the given object should be considered equal. Here, we will treat alerts as equal if they have the same values for all three fields.

- Create some tests for `compareTo` and `equals` on `Alerts`. We've given you a starter `Examples` class – you can put your tests in there. We will check these tests against broken implementations.

2 Creating Monitoring Patterns

An *event* is an action that a user takes on a system, such as opening a file, saving a file, logging in, or trying to access a web page. Each event contains the username that took the action and a timestamp for when the action occurred. **You do not need to create classes for events – they are provided in the starter files.**

An `EventLog` contains a `LinkedList` of events that have occurred on a system (over a period of hours or days). Given an `EventLog`, a *pattern* traverses the log to look for suspicious sequences of events. Checking a pattern generates a list of `Alerts`, one for each suspicious case.

The starter code provides an interface called `IPattern`. That interface requires a method named `run` that takes an `EventLog` and returns a `LinkedList` of `Alerts` generated by that pattern. We have given you one pattern (so you can see an example). You will write two more.

2.1 A Sample Pattern (provided, not something you have to write)

One typical monitoring pattern checks whether users are trying to access other computers (over the internet) that are known to be malicious. Our event logs contain events showing when a user tried to connect to a site on the internet (using computer addresses, not conventional URLs). The monitoring patterns check for users who are trying to connect to malicious (aka "blacklisted" sites).

The starter file `SuspiciousWebPattern.java` defines a pattern for this situation. There is a variable `blacklist` containing addresses (strings) to avoid. For each known username, the pattern iterates through the `EventLog` checking how many times that username tried to access a malicious site. If a user did try to access a malicious site, the pattern creates an alert; the severity is the number of attempted accesses (maxing out at 10).

2.2 Patterns for You to Write

You will now create two more patterns, each one in its own class.

1. When users create very large files, it can indicate that they are trying to save a lot of data that they shouldn't have access to. Our monitor therefore wants to track creation of files larger than 1GB.

   Create a class called `LargeFilePattern` that implements `IPattern`. The `run` method for this pattern creates an `Alert` for any username that has created more than one large file (where large has size greater than one million). The severity for each user is the number of large files they have created.

   You can detect whether an event has saved a large file with the following code (which assumes you have a variable `event` that is a single entry from a `EventLog`, as in the example pattern):

```
(event.getType() == AbsEvent.FILE_SAVED) &&
(((FileSaved)event).getSize() > 1000000)
```

2. Failed attempts to login to a system also raise security warnings, especially if they occur many times for the same user in a short timeframe.

   Create a class called `FailedLoginPattern` that implements `IPattern`. The `run` method for this pattern creates an `Alert` for any username that has more than three failed logins within 5 minutes. Use the number of failed logins in the entire log as the severity within the alert, maxing out at 10.

   [**Correction added 11/19, 1:50pm:** Someone noticed that the `NameCheck` file and the handout used different names for this class. I have corrected the name to match the file, so it should be `FailedLoginPattern` (without an "s" after "Login").]

   [**Clarification added 11/18, 4:45pm:** The original wording was not clear on whether the "number of failed logins" was just within the 5-minute window, or the overall log file. Use "the entire log", since there could otherwise by multiple 5-minute windows with several logins.]

   You can detect whether an event is a failed login attempt with the following code (which assumes you have a variable `event` that is a single entry from a `EventLog`, as in the example pattern):

   ```
   (event.getType() == AbsEvent.LOGIN) &&
   (!((Login)event).wasSuccessful()))
   ```

   (Here, the exclamation point is the logical `not` operator).

   You can determine whether two events are more than 5 minutes apart with the following code (which assumes your events are named `event1` and `event2`):

   ```
   long diff = event1.getTimestamp().getTime() -
               event2.getTimestamp().getTime();
       if (diff / 60000.0 <= 5.0) { ... }
   ```

   [**Update 11/21:** The original comparison used > instead of <=. It should have been <=.]

   You may assume that all of the `EventLogs` have the events listed in the order of their timestamps.

3 Putting it All Together

Now it's time to put the two pieces together. We have given you a skeletal `SecurityMonitor` class that defines a priority queue on alerts and provides some methods for accessing information about the queues. The monitor's constructor takes a list of patterns as input.

Your job is now to run every pattern on an event log and put any generated alerts into the priority queue. We have given you the start of a method `runLogFile` that takes a filename (containing a log) and a list of usernames (that are in the log) and reads the log in from the file. You need to finish this function to run the patterns and populate the priority queue.

The `Examples` class in the starter file shows you how to call this function on a specific log. **We will give you the log files to use for testing** (see below).

3.1 Java Tips

- PriorityQueues have a method `addAll` that takes a list of elements and inserts them all into the queue.
- As you are trying to see what your code is doing, it can be very handy to have Java print out messages showing you the contents of variables in your program. If you look in the `runLogFile`, you will see an example of how we do this using a method called `System.out.println`. This method takes a string (which we can construct using + as string-append) and prints it to the console while the program is running.

4 Testing

Testing this seems difficult unless you know how to create your own log files (which we aren't expecting you to do). We will provide several small and larger files that you can use in testing. You should write test cases on the priority queue contents after running the patterns on our sample files. There is a sample test in the `Examples` class in the starter files. You should include other tests that similarly use the priority queue access methods in `SecurityMonitor`. Our broken implementations could break either pattern implementations or the `runLogFile` code that you need to fill in.

Other test files

Here is a list of test files that you can use, along with the collection of alerts that should get created for each one (you may create them in different orders – that's fine). These are all in the Hwk4StarterFile.zip as of 9pm on 11/19. You can also save these files into your directory (for those who already downloaded the starter files).

- saveLog.txt should yield one alert:

```
user = kathi; severity = 2; type = 4 (AbsEvent.FILE_SAVED)
```

- saveLog2.txt should yield two alerts:

```
user = kathi; severity = 2; type = 4 (AbsEvent.FILE_SAVED)
user = simmon; severity = 3; type = 4 (AbsEvent.FILE_SAVED)
```

- loginLog.txt should yield one alert:

```
user = root; severity = 4; type = 6 (AbsEvent.LOGIN)
```

- loginLog2.txt should yield no alerts.
- webrequestlog.txt should yield one alert:

```
user = kathi; severity = 1; type = 5 (AbsEvent.WEB_REQUEST)
```

- largeLog1.txt should yield 3 alerts:

```
user = simmon; severity = 4; type = 4 (AbsEvent.FILE_SAVED)
user = jordan; severity = 3; type = 4 (AbsEvent.FILE_SAVED)
user = jordan; severity = 10; type = 6  (AbsEvent.LOGIN)
```

- eventlog1.txt should yield 4 alerts:

```
user = kathi; severity = 2; type = 5 (AbsEvent.WEB_REQUEST)
user = simmon; severity = 1; type = 5 (AbsEvent.WEB_REQUEST)
user = jordan; severity = 2; type = 5 (AbsEvent.WEB_REQUEST)
user = simmon; severity = 10; type = 6  (AbsEvent.LOGIN)
```

- eventlog2.txt should yield 4 alerts:

```
user = root; severity = 2; type = 5 (AbsEvent.WEB_REQUEST)
user = kathi; severity = 1; type = 5 (AbsEvent.WEB_REQUEST)
user = jordan; severity = 1; type = 5 (AbsEvent.WEB_REQUEST)
user = jordan; severity = 10; type = 6 (AbsEvent.LOGIN)
```

- largestLog.txt should yield 3 alerts:

```
user = kathi; severity = 1; type = 5 (AbsEvent.WEB_REQUEST)
user = kathi; severity = 2; type = 4 (AbsEvent.FILE_SAVED)
user = jordan; severity = 10; type = 6 (AbsEvent.LOGIN)
```

Don't include tests on your own log files! If you created some log files of your own, don't include tests on them in your submitted `Examples` class, because our reference solution won't have your log files. If you comment out just the `@Test` line before a test that uses your own file, it won't get picked up by autograding (so you can leave it in the file, just take off the annotation).

In terms of grading, we will do thoroughness checking on your tests for the `compareTo` and `equals` methods in the `Alert` class, as well as your queue-based tests on two large log files (which will be posted soon).

This week we will assess a 10% deduction for work that we have to grade manually due to compilation errors that remain unresolved after the resubmission deadline.

5 Submission Guidelines

Unzip the starter files. Add any files you needed to create to that directory, then zip up and submit the results.