HW7 – Applying the SOLID Principles

Estimated time: 14-16 hours

Learning Objectives

- Gain experience with SOLID principles
- Gain more experience with testing techniques

Overview

In this homework assignment, you will build a small reusable collection of software components for analyzing network traffic to detect hacker attacks. Each of the components should

- Have only reason to change
- Be open for extension, but closed to modification
- Facilitate a custom application's compliance with the Liskov Substitution principle
- Have a simple, cohesive interface
- Follow the dependency inversion principle

In other words, the collection of components should follow the SOLID principles and thereby support reuse and extensibility.

Although you will have to implement two concrete algorithms for detecting attacks, those algorithms are NOT the focus of the assignment. Keep those algorithms as simple as possible without comprising their basic functionality.

Background

Processes communicate across networks by sending messages. We can be learned about what's happening on a network by just watching the source, destination, and timing of those messages. It's not necessary to read the content of the messages.

On the Internet, the source and destination of messages are identified by an IP Address and Port Number. The IP Address identifies the machine on which the process is running and the Port Number identifies the process. If machines were office buildings and processes people in those buildings, then the IP addresses would be like the mailing address of the building and the Port Number would be like a person's name or mail stop.

IP Addresses (in IP Version 4) consist four unsigned bytes, i.e., numbers between 0 and 255. They are often written with highest order number first and each number separated by comas, e.g.

129.123.3.42
10.15.52.1

172.168.3.4
192.168.56.24

Port numbers are unsigned 2-byte integers, and therefore range from 0 to 65535. The complete source or destination for a message contains both an address and port number, e.g.

129.123.3.42,80
10.15.52.1,443
172.168.3.4,3389
192.168.56.24,5432

There are many different programs that can watch the traffic on a network and records information about the messages in motion. For our purposes in this assigned, we'll assume that a traffic monitoring programming has recorded some basic fact about every message to a give destination address in a text file specific to that destination. Some sample text files are provided.

Requirements

Below is a list of requirements for the desired set of reusable components, which we will call the *Internet Traffic Analysis Kit* or *ITAK*. In these requirements, "user" means some other piece of software that makes use the ITAK. For example, a "user" could be a custom hacker attack detector that is looking for a specific pattern of the messages.

1. ITAK component must be able to read the records of network traffic from an input stream. It should not matter to ITAK what the input stream is or how it was opened. An ITAK user is responsible for opening an input stream passing that stream to an ITAK component.

2. An input stream will be formatted as follows:
   a. Each line represents one message on the network
   b. Each line is terminated by a line feed character (\n) or a carriage return followed by a line feed (\r\n)
   c. Each line will contain four coma-separated fields

      <timestamp>, <src address>, <src port>, <des port>

   where

<timestamp>	is a long integer that represent the time of the message in seconds since the beginning of the year. Note that many message may have the same timestamp. It is an approximation, not an exact time.
<src address>	is the IP Address of the source process
<src port>	is the Port Number of the source process
<des port>	is the Port Number of the destination process

Note that the record does not contain the destination. That is because the whole input stream is for one destination address.

3. ITAK must include an abstract component, called *Analyzer*, that represents network traffic analysis algorithms.
   a. This component must have a public method, called *run*, that takes an input stream reference as a parameter and produces a *ResultSet* (see Requirement 5) as an output.
      i. The behavior of the run must be extensible
   b. This component must include either a constructor or "set" method(s) for setting configuration parameters that will tune the algorithm. See Requirement 4 for more details on configuration parameters
   c. ITAK should include two sample analyzers that reuse the abstraction provided by *Analyzer*: one for finding *Denia-of-Service* attacks and one for find *Port-Scanning* attacks
      i. For consistency, the Denial-of-Service analyzer should be called *DenialOfServiceAnalyzer* and its *run* method should implement the algorithm explained in Appendix A.
      ii. Similarly, the Port-Scan analyzer should be called *PortScanAnalyzer* and its *run* method should implement the algorithm explained in Appendix B.
   d. *DenialOfServiceAnalyzer* and *PortScanAnalyzer* should be usable without further extension, but should still allow for extension. In other words, they should be closed for modification by open for extension.
4. The run behavior of any Analyzer, like an instance of *DenialOfServiceAnalyzer*, must be configurable. A *configuration parameter* is a key-value pair, where the key is the name the parameter and the value is the parameter setting. A *Configuration* is a set of key-value pairs.
   a. A *Configuration* should be able to hold an arbitrary number of configuration parameters
   b. The name of configuration parameter can be any string.
   c. The value of configuration parameter is also a string
   d. A Configuration should be able get the value of parameter given its name
      i. A Configuration should include a method for returning the value as a string
      ii. A Configuration should include a method for returning the value as an integer
      iii. A Configuration should include a method for returning the value as an double
5. A *DenialOfServiceAnalyzer* object should only accept Configuration objects that include, as a minimum, parameters with the following name: "Timeframe", "Likely Attack Message Count", "Possible Attack Message Count". See the algorithm in Appendix A to understand how the values of these parameters are used.

6. A *PortScanAnalyzer* object should only accept Configuration objects that include, as a minimum, parameters with the following name: "Likely Attack Port Count" and "Possible Attack Port Count". See the algorithm in Appendix B to understand how the values of these parameters are used.
7. The *run* method for any analyzer should return a result set, which is container of key-value pairs, where the keys are strings and the values are containers of strings. The contents and meaning of the key-value pairs are specific to the type of analyzer. For consistency, call the result set software component simply, *ResultSet*.
    a. *ResultSet* should include a *print* method that accepts an ostream reference and prints out each key, and then each string in the container (value) for the key. The format should be readable, but is otherwise not constrained.
    b. The *ResultSet* produced by an instance of *DenialOfServiceAnalyzer* will contain at least the following key-value pairs:

        Key: "Likely attackers"
        Value: a container of strings that represent each likely attacker's source

        Key: "Possible attackers"
        Value: a container of strings that represent possible attacker's source

        Key: "Attack Periods"
        Value: a container of strings, with strings for each likely or possible attack format as <start time>-<end time>

        Key: "Timeframe"
        Value: a container of string with one string that represents of the timeframe

    c. The *ResultSet* produced by an instance of *PortScanAnalyzer* will contain at least the following key-value pairs:

        Key: "Likely Attackers"
        Value: a container of strings that represent each likely attacker's source

        Key: "Possible Attackers"
        Value: a container of strings that represent possible attacker's source

        Key: "Port Count"
        Value: a container of string with one string that represents of the port count configuration parameter

    d. Other kinds of analyzers may produce instances of *ResultSet* with different contents.

For this assignment, you must design, implement, and test the whole project. The design will need to contain at least five classes: *Analyze*r, *DenialOfServiceAnalyzer*, *PortScanAnalyer*, *Configuration*, and *ResultSet*. The design may contain other classes, as you see fit. You don't need to show any testing classes in your design.

For the implementation, you must create a new CLion project. For consistency, call your project *ITAK*. There is no starting code for this assignment, but there are some sample data files that you can use for testing. You may re-use any from any previous assignment or example. In fact, you are strongly encouraged to do so. Hint: re-use the Utils functions for string process and string-to-integer or string-to-double conversions. You can use your own Dictionary generic where needed or use a similar component from the Standard Library. You may use the Container generic in the shared repository or the vector generic from the standard library. Basically, you reuse any public domain software. As always, you may not use another student's work. If you re-use something in you code that is not from the standard library, include a citation in your code – anything violation to this practice will be consisted plagiarism.

Set your project up like the last homework assignment. In other words, have a Testing directory with a testMain.cpp and edit the CMakeLists.txt so there are two targets: one for a main program and one for a test program. Also, at least two configurations: one to run the test target and one to run the main target. You testMain.cpp needs to run a thorough set of test cases that you implement. The main.cpp should illustrate a typical and interesting usage of the *DenialOfServiceAnalyzer* or *PortScanAnalyzer*.

Instructions

For HW7, you must do the following

1. Study the requirements and appendices (DO NOT SKIP THIS STEP!)

2. Create a ITAK project in your Git repository.

3. Add a Testing folder with a testMain.cpp and setup the CMakeLists.txt

4. Complete a design in UML according to the requirements.

5. Implement and complete test cases as your implement various components and their methods. DO NOT WAIT UNTIL THE END TO TEST. Start by implementing something that doesn't depend on anything else. Then, implement something else that either doesn't depend on anything else or only depended on things already implement and tested. Repeat this process until everything is implemented and tested.

6. Implement an interesting usage of DenialOfServiceAnalyzer or PortScanAnalyzer in main.cpp as an illustration.

7. Finally, push all your materials to your Git repository, by doing the following

   a. Open a terminal window or command prompt

   b. Change your working directory to your repo directory
      ```
      $ cd <your repo directory>
      ```

   c. Add all files to your repo
      ```
      git add .
      ```

   d. Commit a new version, with the message “Final version of HW7”
      ```
      git commit –m “Final version of HW7”
      ```

   e. Push to your remote repository
      ```
      git push
      ```

8. Submit the URL for your Git repository to Canvas

Grading Criteria

Requirements	**Max Points**
Proper setup and submission of HW7 using Git	5
A complete design (documented with a UML Class Diagram) that exhibits good localization of design decisions, encapsulation, abstraction, inheritance, and aggregation	15
An implementation that is correct with respect to the design, including an main.cpp that illustrates how to use the DosAnalyzer or PortScanAnalyzer	10
Meaningful test cases for all components in ITAK	15
The test and main target that run as expected according to the requirements	5

Penalties	**Deduction**
Incomplete or inaccessible Git repository	0 to -50
Doesn’t compile	0 to -20
Poor code readability	0 to -10
Warnings in the code that could have been easily corrected	0 to -10
Weak parameter or method declarations (things could have be declare as “const” were not)	0 to -5

Appendix A – Sample algorithm for detecting denial-of-service attacks

A denial of service attack when occurs one source IP address floods a destination (address and port) with lots of message.

Algorithm Overview

This denial-of-service detection algorithm is basically a counting summation algorithm. It consists of two parts: a data loading/process phase and an attack detection phase.

Data loading/filing phase: Read each record (line) from the input stream and account for it in an address-to-summary dictionary, where

- the key is string and represents source IP address
- the value is a timestamp-to-count dictionary, where
    - the key is the timestamp of the message
    - the value is counter

To process a record, get the source IP Address and look it up in the address-to-summary dictionary. If it is not, then add it to the dictionary and with a new timestamp-to-count dictionary as the value. Next, get the timestamp and look it up in the timestamp-to-count dictionary for the source IP address. Add 1 to value associated with that timestamp.

Since the records are reads and filed in chronological order, all the entries in the timestamp-to-count dictionary will be in chronological order. This is important for the next phase.

Attack detection phase: Find windows of time, based on a timeframe, where message counts exceed a threshold.

- Let *results* be instance of ResultSet
    - Add a key-value pair of (“Likely Attackers”, new Container<string>)
    - Add a key-value pair of (“Possible Attackers”, new Container<string>)
    - Add a key-value pair of (“Attack Periods”, new Container<string>)
    - Add a key-value pair of (“Timeframe”, new Container<string>)
- Let *timeframe* be the value of the “Timeframe” configuration parameter in the analyzer’s *Configuration* object.
    - Add a string represent of timeframe to string container associated with “Timeframe” in the *results*
- Let *likelyThreshold* be the value of the “Likely Attack Message Count” configuration parameter.
- Let *possibleThreshold* be the value of the “Possible Attack Message Count” configuration parameter.

- For each key-value pair, *x,* in the address-to-summary dictionary, do the following:
    - Loop through each key-value pair, *y*, in *a's* value, which is a timestamp-to-count dictionary for that source IP address
        - Let s be the *y's* key, which represents the start time
        - Let *c* be the *y's* value, namely the count of messages at the start time
        - Compute the total messages in a window of time starting at *s*. Specially, loop through all key-value pairs, *z*, that come after *y* but before *s + timeframe*
            - Add *z's* value to *c*
    - If *c* >= *likelyThreshold*, add the *x's* key, which is the source IP address, to the list of strings associated with "Likely Attackers" key in the *resultSet*. Also add a string that represent of window of time to the Container<string> associated with "Attack Periods" in the *resultSet*.
    - Else if *c* >= *possibleThreshold*, add *x's* key to the list of strings associated with "Possible Attackers" key in the *resultSet.* Also add a string that represent of window of time to the Container<string> associated with "Attack Periods" in the *resultSet*.

Note: This algorithm is not the most efficient, but sufficient. Also, note that if you re-use your Dictionary from HW6, you must extend it with a method to update the value of an existing key-value pair.

Appendix B – Sample algorithm for detecting port-scan attacks

A port-scan attack occurs one source IP address sends message to lots of different ports on a machine in a short period of time

Algorithm Overview

This port-scan detection algorithm is basically a counting summation algorithm. The algorithm consists of two parts: a data loading/processing phase and an attack detection phase.

Data loading/filing phase: Read each record (line) from the input stream and account for it in an address-to-summary dictionary, where

- the key is string and represents source IP address
- the value is a Container<int> that contains unique destination ports that the source IP address as tried to communication with

To process a record, get the source IP Address and look it up in the address-to-summary dictionary. If it is not, then add it to the dictionary and with a new port container as the value. Next, look up the port in the port container. If it is not there, add it.

Since the records are reads and filed in chronological order, the entries in the timestamp-to-ports dictionary will be in chronological order.

Attack detection phase: Find source IP address with messages to lots of different ports.

- Let *results* be instance of *ResultSet*
  - Add a key-value pair of (“Likely Attackers”, new Container<string>)
  - Add a key-value pair of (“Possible Attackers”, new Container<string>)
  - Add a key-value pair of (“Port Count”, new Container<string>)
- Let *likelyThreshold* be the value of the “Likely Attack Port Count” configuration parameter.
- Let *possibleThreshold* be the value of the “Possible Attack Port Count” configuration parameter.

- For each key-value pair, *x,* in the address-to-summary dictionary, do the following:

  - Let *src* be *x’s* key, which is a source IP address
  - Let *c* be the number of entries in x*’s* value, i.e. c is the number of ports that the *src* sent messages to
  - If c >= *likelyThreshold*, add *src* to the list of strings associated with “Likely Attackers” key in the *resultSet*

    - Else if c*ount* >= *possibleThreshold*, add *src* to the list of strings associated with "Possible Attackers" key in the *resultSet*.

Note: This algorithm is not the most efficient, but sufficient.