Project 9: Packet Sniffer

Introduction to the Project

Name 1: Ishan Rai ID Number: 2019B3A70504P
Name 2: Shrey Bansal ID Number: 2019B3A70592P

Here is a general overview of the different classes of which the project is composed:-

1. Abstract Traffic Class
2. Attack Traffic Class
3. Normal Traffic Class
4. Victim Host Class

The goal of the Project - Sniff all the packets on a port, and extract all the information from the packets including source IP, destination IP, source port, destination port. If the number of packets exceeds 1000, then it is classified as an attack through the victim host class.

Normal Traffic Class

The Normal Traffic class is one of the subclasses of the traffic class, through this class we generate traffic towards the Victim Host. The total number of packets through this class = 5. For showing that data travels in the packet, we have added a string in each of the packets which can be imputed to the packet through the console and is visible on the console of the Victim Host class along with all the other info such as source port, destination port, etc.

Attack Traffic Class

Attack Traffic class is one of the subclasses of the traffic class, through this class we generate traffic towards the Victim Host. Total number of packets through this class = 1050. For showing that data travels in the packet, we have added a string in each of the packets which are randomly generated through a random String Generator function and are visible on the console of the Victim Host class along with all the other info such as source port, destination port, etc.

Victim Host Class

This is the class where the port scanning is done and it presents the required data along with the input string to the console. As soon as the number of packets exceeds 1000, with each new package, a message of An Attack is Happening is printed to the string. It not only accepts the packets but also scans the packet, which is done through the method scan port.

Design Principles

1. **Encapsulate what varies**
2. **Favor composition over Inheritance**

   Rather than explicitly mentioning the exact number of packets in both the Attack Traffic and the Normal Traffic classes, we can instead make an interface (an instant of which is present in the superclass of these 2 classes -> Traffic Class i.e, **has- A relationship**) Let's name this interface **TrafficGenerator,** which has an abstract method returning the number of packets requested. This interface will be extended by 2 concrete classes, one the **AtttackTrafficGenerator** and the other the **NormalTrafficGenerator.** Both of these methods would provide an implementation for the method requesting the number of Packets.

   Now both of these concrete classes can be instantiated in the AttackTraffic class and the NormalTraffic class respectively, taking the reference of the interface, **TrafficGenerator,** thus getting the number of packets to be generated

   Another possible scenario where we can use composition is for the Port Scanning Class, wherein rather than inheriting the Victim Host class and then Scanning all the packets, we could provide an instance of the Victim Host class in the Port Scanning class and then instantiate it within the port scanning class's constructor. This would help in providing **more flexibility** to the code.

   Advantage- Using this principle we can make changes in the number of packets without making changes to our main code. Code becomes more readable and easier to interpret for a new eye.

   Disadvantage - Since the number of packets is constant for both of the concrete classes - **AttackTraffic and NormalTraffic,** it is probably better to just declare public static final variables at the beginning of the code (thereby declaring them as constants). If these variables are named appropriately the code is still readable. Moreover, this is a modular approach as we would just have to alter that constant. This is indeed what we have actually done.

3. **Program to an interface not implementation**
4. **Depend on abstraction, do not depend on concrete classes**

The abstract traffic class can be used to encapsulate the common behavior of the NormalTraffic class and the AttackTraffic Class. This behavior would then be inherited by both the concrete Traffic classes.

Advantage - This also helps us in maintaining the code for future use in case we want to add new functionalities to both the AttackTraffic and the NormalTraffic class.

5. **Strive for loose coupling between objects that interact**

   This design principle can be used very efficiently in the Packet Sniffer program as there are various classes interacting with each other. For example, we can see that the Traffic classes (AttackTraffic and NormalTraffic) are interacting continuously with the VictimHost class. Now, it would be a nightmare to test this code without loose coupling. This is because the loose coupling between these classes helps us in reducing the dependencies between these. If there's an error in one of them, it won't necessarily affect the other. We have tried to achieve this in our code as far as possible. The only dependency between the traffic classes and the Victim host class is the Port Number they share (different constant variables but the same value which is necessary for interaction). The variables, methods, and everything else behaves independently of each other.

6. **Classes should be open for extension and closed for modification**

   Since the traffic class is an abstract class, the classes NormalTraffic and AttackTraffic are open for extension. This could be achieved by implementing the extensions in the Traffic class instead of modifying the source code for the NormalTraffic and the AttackTraffic classes. For making this more concrete we can make the static methods final as well in the concrete traffic classes.

Use of Design Patterns:-

Strategy Design Pattern

Strategy Design pattern can be used in our project by encapsulating the part of the code that varies and putting it in an interface whose instance is provided in the Abstract Traffic class. (Composition).

For example, the number of packets in both the classes differ, so as explained above we can make an interface TrafficGenerator which contains the method generateTraffic() and provide it's 2 different implementations in the 2 concrete classes AttackTrafficGenerator and NormalTrafficGenerator. Then an instance for these concrete classes can be passed in the

constructor of the AttackTraffic and the NormalTraffic class respectively, thus instantiating the TrafficGenerator interface, as the abstract traffic class would contain a reference of the Traffic Generator and return the number of packets to be generated.

The main advantage of this pattern is that we have shifted the part of the code which can change continuously away from the main part where all the implementation is taking place, this allows us to change, add new values for the number of Packets or even create new concrete classes for the TrafficGenerator class without making any changes to the Traffic and it's subclasses. Thus increasing the functionality and the flexibility of the code.

Observer Design Pattern

We can use the **Observer design pattern** in this program as we have to constantly observe the request made by the client. As the user inputs the request, we can notify the client class. This is because the **observer pattern defines the one to many relationships, so that when any one of the objects changes state (In this case, the Attack Traffic or the Normal Traffic Class), all of its dependencies (In this case, the Victim Host class) are notified immediately. So whenever a new string is generated and a packet is being sent from one of the Traffic classes, we can notify the VictimHost class.**

Link to the video presentation: Click here for access to the project