

## 4.1 Introduction :

After introducing our approach and explained it in detail in the previous chapter, we will need to validate. To implement this approach methodology we have developed a detector under the Netbeans environment using the Java programming language

So this chapter is structured as follows: first we will present the development environment (java, Winpcap, Jpcap) then we will see the Nmap scanner used for the tests. Finally we present our developed detector application.

## 4.2 Platform :

### 4.2.1 Java :

**Java** is a computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. It is intended to let application developers, WORA, meaning that code that runs on one platform does not need to be recompiled to run on another. Java applications are typically compiled to bytecode (class file) that can run on any JVM regardless of computer architecture. Java is, as of 2014, one of the most popular programming languages in use, particularly for client-server web applications, with a reported 9 million developers. Java was originally developed by James Gosling at Sun Microsystems (which has since merged into Oracle Corporation) and released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++, but it has fewer low-level facilities than either of them.

The original and reference implementation Java compilers, virtual machines, and class libraries were developed by Sun from 1991 and first released in 1995. As of May 2007, in compliance with the specifications of the Java Community Process, Sun relicensed most of its Java technologies under the GNU General Public License. Others have also developed alternative implementations of these Sun technologies, such as the GNU Compiler for Java (bytecode compiler), GNU Classpath (standard libraries), and IcedTea-Web (browser plugin for applets).[32]

### 4.2.2 NetBeans :

**NetBeans** is an IDE for developing primarily with Java, but also with other languages, in particular PHP, C/C++, and HTML5. It is also an application platform framework for Java desktop applications and others.

The NetBeans IDE is written in Java and can run on Windows, OS X, Linux, Solaris and other platforms supporting a compatible JVM.

The NetBeans Platform allows applications to be developed from a set of modular software components called *modules*. Applications based on the NetBeans Platform (including the NetBeans IDE itself) can be extended by third party developers.

The NetBeans Team actively support the product and seek future suggestions from the wider community. Every release is preceded by a time for Community testing and feedback.

[33]

### 4.3 Winpcap:

Winpcap is the packet capture and filtering engine of many open source and commercial network tools, including protocol analyzers, network monitors, network intrusion detection systems, sniffers, traffic generators and network testers. Some of these tools, like (Wireshark, Nmap, Snort, ntop) are known and used throughout the networking community. [34].

#### 4.3.1 Winpcap structure:

To capture data from the network, a capture application must directly interact with the network adapter. Therefore the operating system must offer a set of primitives for capture with a view to communicate with the adapter. The purpose of these primitives is to capture network packets transparently from the point of view of the user and to transfer them to the calling application. This part of the kernel should be quick and efficient in order to capture all the packets in real time without losing information. The logical architecture of Winpcap is a hierarchical structure on 3 levels (from the network adapter to an application): [35]

- **The *Packet Capture Driver* :**

Is the lowest software level of the capture structure. It is a part of the kernel and interacts with the network adapter to obtain captured packets. It supplies the application a set of functions used to read /write data from the network at data-link level.

- ***Packet.dll* :**

Works at user level, but it is detached from the capture application. It is a dynamic link library that separates the application from capture driver providing a system-independent capture interface. It allows user's application to be executed on different Windows

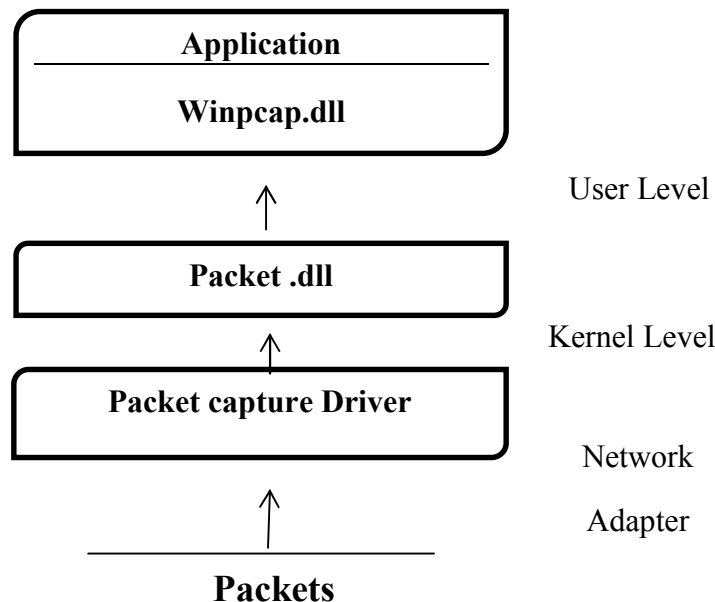
operating systems without being recompiled. It represents a set of API functions used to access the capture driver directly.

• **WinPCap.dll :**

(The third level) is a *static* library that is used by the packet captures part of the application. It uses the services exported by Packet.dll, and provides the applications a higher level and a powerful capture interface. It is statically linked; it means it is part of the application that uses it. The user interface is the highest part of the capture application. It manages the interaction with the user and displays the result of a capture

#### 4.3.2 WinPcap consists of:

- x86 and x86-64 drivers for the Windows NT family (Windows NT 4.0, Windows 2000, Windows XP, Windows Server 2003, Windows Vista, Windows 7, etc.), which use NDIS to read packets directly from a network adapter;
- Implementations of a lower-level library for the listed operating systems, to communicate with those drivers;
- A port of libpcap that uses the API offered by the low-level library implementations



**Figure 4.1** Logical Structure of Winpcap

#### 4.3.3 The working of a WinPCap:

The Winpcap action is based on packets capture at the network adapter level. Therefore, any application using WinPCap must follow the steps summarized below:

- a. Specify the network adapter which will be used for capture.
- b. Initialize winpcap (mention the network adapter if the capture is on-line)
- c. Offer a pattern for the captured data (a TCP/IP structure for example), so it can be done a cast at this pattern and obtain significant information (complex applications offer patterns for majority protocols).

## **4.4 Jpcap :**

Is based on libpcap/Winpcap, and is implemented in C and Java JPCAP is a Java library for capturing and sending network packets. Using JPCAP, we can develop applications to capture packets from a network interface and visualize/analyze them in Java.

JPCAP can capture Ethernet, IPv4, IPv6, ARP/RARP, TCP, UDP, and ICMPv4 packets.

JPCAP is a set of Java classes that provide an interface and system for network packet capture and also a protocol library and tool for visualizing network traffic is included. JPCAP hides the low-level details of network packet capture by abstracting many network packet types and protocols into Java classes.[36]

### **4.4.1 Function of Jpcap:**

Jpcap is an open source library for capturing and sending network packets from Java applications. It provides facilities to:

- Capture raw packets live from the wire.
- Save captured packets to an offline file, and read captured packets from an offline file
- Filter the packets according to user-specified rules before dispatching them to the application.
- Send raw packets to the network Jpcap is based on libpcap/winpcap, and is implemented in C and Java and discover patterns of intrusions. This is the initial stages of present research; much remains to be done including the following tasks:
  - Implement a support environment for system builders to iteratively drive the integrated process of pattern discovering, system feature selection, and construction and evaluation of detection models.
  - Investigate the methods and benefits of combining multiple simple detection models. It is needed to use multiple audit data streams for experiments.
  - Implement a prototype agent-based intrusion detection system.
  - Evaluate present approach using extensive audit data sets.

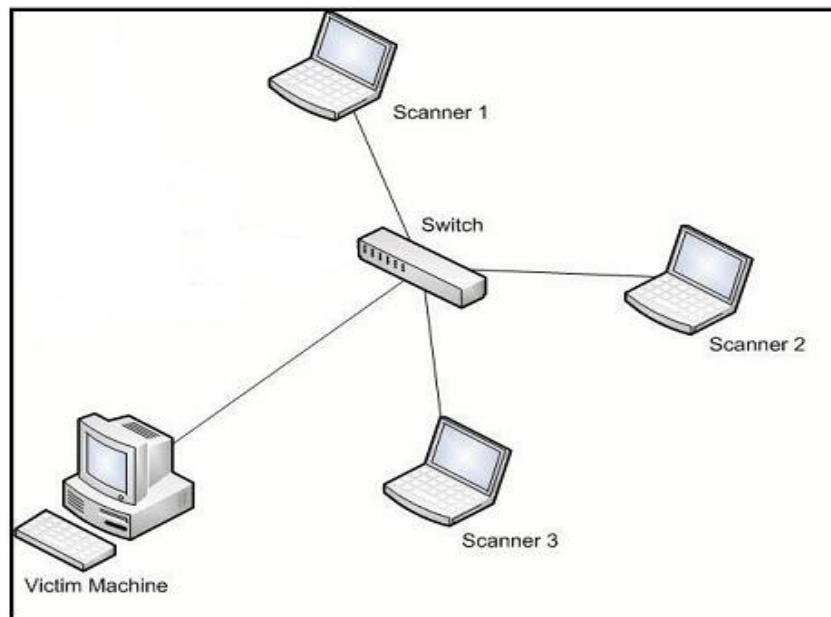
## 4.5 Experiments:

We implemented our proposed method and tested it on a small local area network. A testbed was built comprised of a victim machine, a Layer 2 switch, and three other machines as shown in Figure 4.1. We assume that the victim machine provides services and by such is a server.

To test our detection technique, we use a very popular scanning tool called Nmap to perform the three types of scans (the connect scan, the half-connect scan and the FIN scan), where the time interval between the sent probes can be set by the scanner. The Nmap was run on the three scanner machines and the time interval between the scanning probes was set to be 0.4 seconds, 4 minutes and 6 minutes for scanner1, scanner2 and scanner3 respectively. These intervals were chosen to represent both normal port scanning (scanner1) and slow port scanning (scanner 2 and 3). In fact, Nmap uses an interval between the probes equal to 0.4 seconds as a default value when normal port scanning is performed.

To implement our detection method, we developed a Java code that captures traffic at the network interface of the victim server using the Jpcap library. The time window (T) is set to be 3 minutes, so that based on the traffic of each IP in three minutes, we calculate the three features ( $N_{closed}$ ,  $N_{Half\_connect}$ ,  $N_{Fin}$ ) and we divide the IPs based on these features into scanner IPs, suspicious IPs. A suspicious IP is removed from the suspicious list after (K=5) consecutive time windows.

Our detection method was able to identify that scanner 1 is performing port scanning in the first time window and a notification was sent to the administrator of detecting port scanning coming from the IP of the scanner1. Scanners 2 and 3 who were performing slow port scans (where the interval is equal to 5 and 6 minutes) were classified as suspicious IPs in the first time window and were added to the suspicious list. By observing the behavior of the scanners in the following 5 time windows, our detection method notices that their behavior is again abnormal. Actually, our approach detected the malicious scanning behavior in the second and third time windows for Scanner 2 and Scanner 3 respectively and alarmed the administrator of detecting slow port scanning. Our experiment proves that our method is able to correctly detect both normal and slow port scanning without giving false alarms.



**Figure 4.2** Testbed Topology

The victim machine represents a server that offers services, one of the three scanners is performing normal port scanning and the other two are performing slow port scanning on the victim server.

## 4.6 Nmap:

To scan the ports we use the tool Nmap.

### 4.6.1 Find hosts:

Use this option when you need to determine hosts and routers in a network scan. It will traceroute and ping all hosts defined in the target with the command:  
`nmap -sn 192.168.43.*`

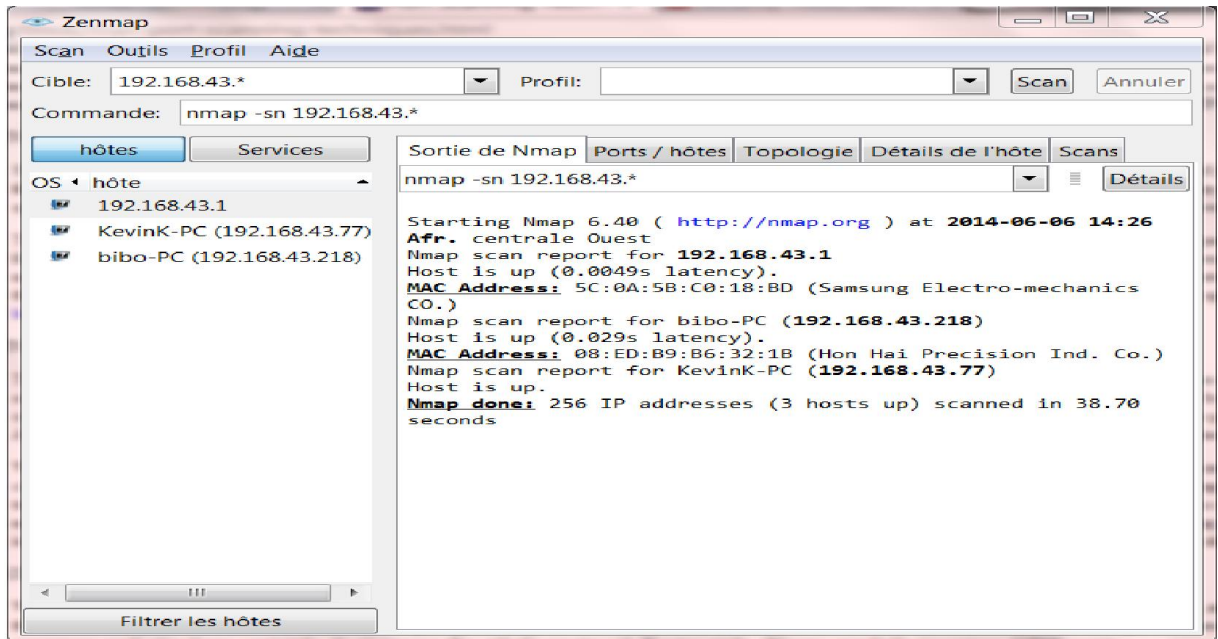


Figure 4.3 list of hosts with Nmap

## 4.6.2 Port scanning techniques:

### 4.6.2.1 TCP Syn Scan (-sS):

We use the command: `nmap -sS 192.168.43.77`

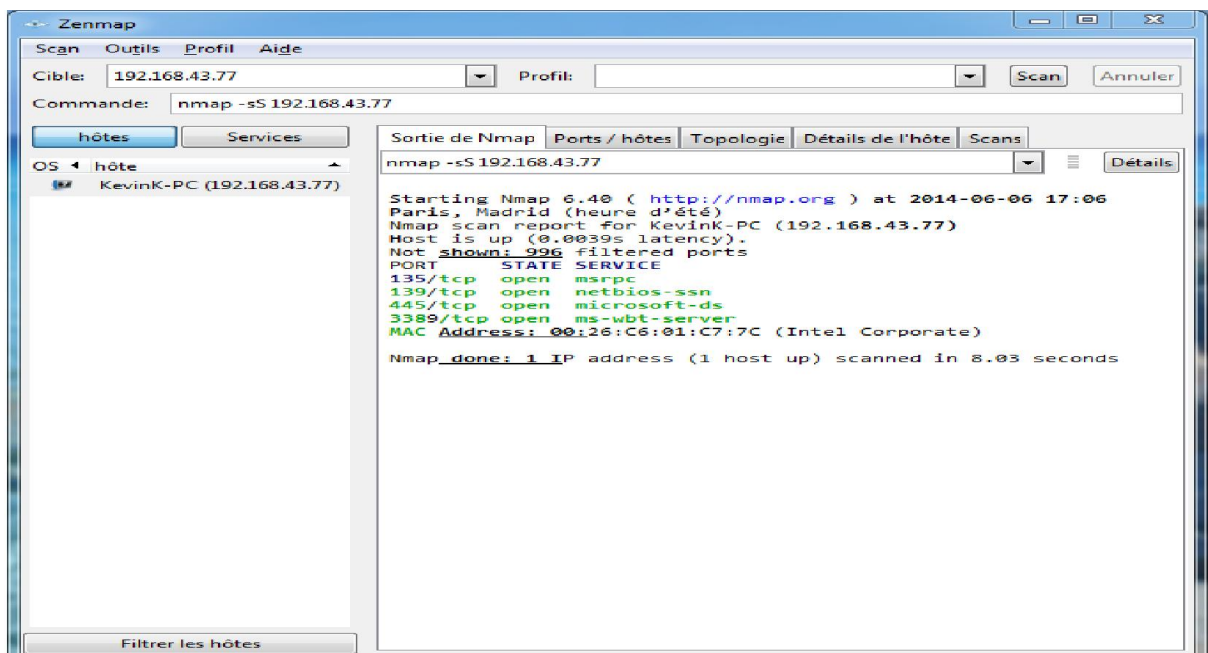


Figure 4.4 Tcp SYN scan

#### 4.6.2.2 Tcp connect scan(-sT):

The command: `nmap -sT 192.168.43.77`

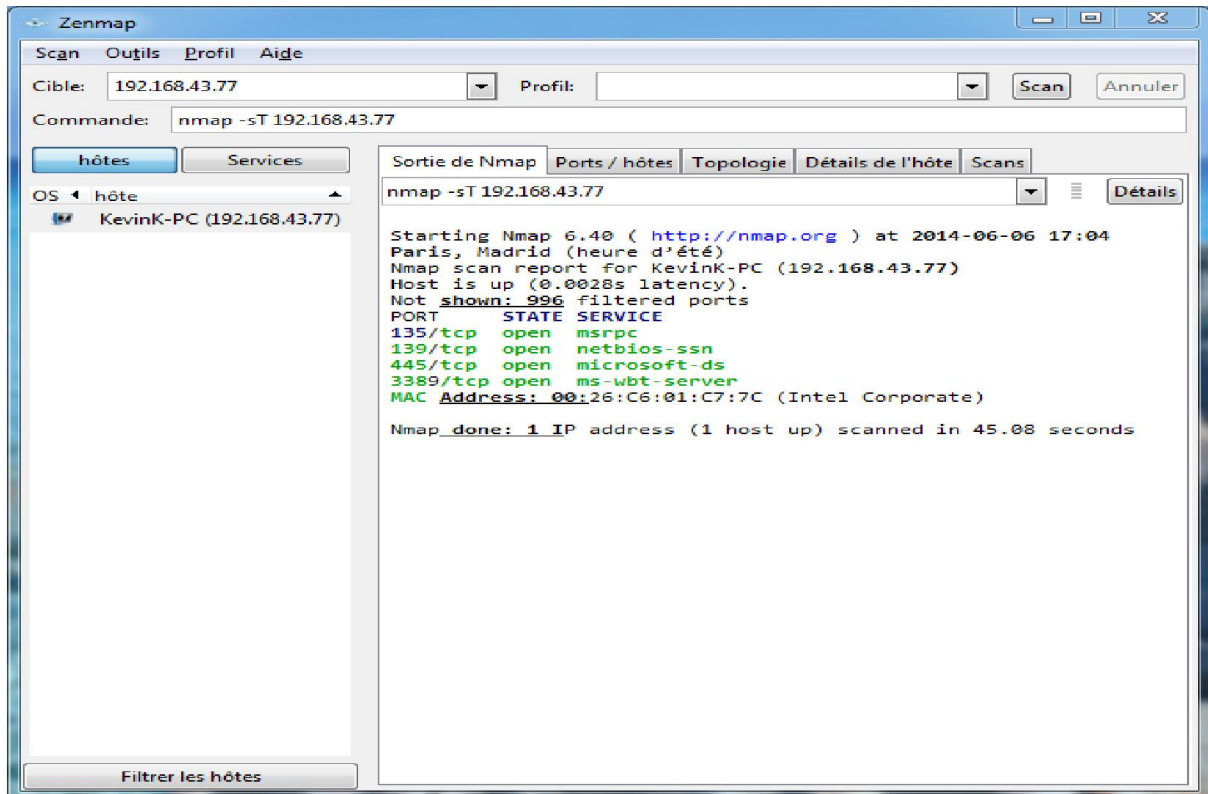


Figure 4.5 TCP connect scan

#### 4.6.2.3 Fin scan (-sF):

The command: `nmap -sF 192.168.43.77`

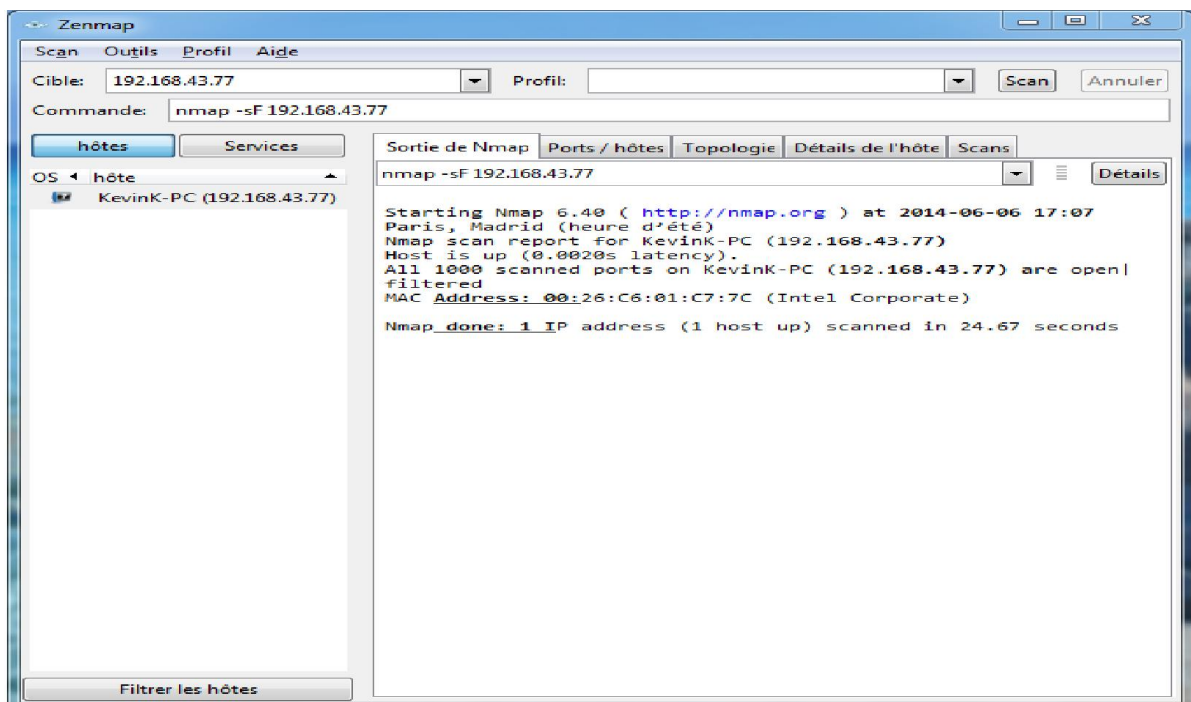


Figure 4.6 FIN scan

#### 4.7 Packet Capturing using Jpcap:

- **Obtain the list of network interfaces:**

When you want to capture packets from a network, the first thing you have to do is to obtain the list of network interfaces on your machine. To do so, Jpcap provides `JpcapCaptor.getDeviceList()` method. It returns an array of `NetworkInterface` objects. [36]

The following sample code obtains the list of network interfaces and prints out their information.

```
//Obtain the list of network interfaces
NetworkInterface[] devices = JpcapCaptor.getDeviceList();
```

A `NetworkInterface` object contains some information about the corresponding network interface, such as its name, description, IP and MAC addresses, and datalink name and description.

```
//Obtain the list of network interfaces
NetworkInterface[] devices = JpcapCaptor.getDeviceList();

//for each network interface
for (int i = 0; i < devices.length; i++) {
//print out its name and description
    System.out.println(i+": "+devices[i].name + "(" +
devices[i].description+"");

//print out its datalink name and description
    System.out.println(" datalink: "+devices[i].datalink_name +
 "(" + devices[i].datalink_description+"");

//print out its MAC address
    System.out.print(" MAC address:");
```

```

    for (byte b : devices[i].mac_address)
        System.out.print(Integer.toHexString(b&0xff) + ":");
    System.out.println();

    //print out its IP address, subnet mask and broadcast address
    for (NetworkInterfaceAddress a : devices[i].addresses)
        System.out.println(" address:"+a.address + " " + a.subnet
+ " " + a.broadcast);
}

```

#### • Open a network interface:

Once you obtain the list of network interfaces and choose which network interface to capture packets from, you can open the interface by using `JpcapCaptor.openDevice()` method. The following piece of code illustrates how to open a network interface.

```

NetworkInterface[] devices = JpcapCaptor.getDeviceList();
int index=...;
// set index of the interface that you want to open.
//Open an interface with openDevice(NetworkInterface intrface,
int snaplen, boolean promisc, int to_ms)
JpcapCaptor captor=JpcapCaptor.openDevice(device[index],
65535, false, 20);

```

When calling the `JpcapCaptor.openDevice()` method, you can specify the following parameters:

| Name:                            | Purpose                                                                          |
|----------------------------------|----------------------------------------------------------------------------------|
| <i>NetworkInterface intrface</i> | Network interface that you want to open                                          |
| <i>int snaplen</i>               | Max number of bytes to capture at once.                                          |
| <i>boolean promisc</i>           | True if you want to open the interface in promiscuous mode, and otherwise false. |
|                                  | In promiscuous mode, you can capture packets every packet                        |

from the wire, i.e., even if its source or destination MAC address is not same as the MAC address of the interface you are opening.

In non-promiscuous mode, you can only capture packets sent and received by your host.

*int to\_ms*

Set a capture timeout value in milliseconds.

`JpcapCaptor.openDevice()` returns an instance of `JpcapCaptor`. You can then call several methods of the `JpcapCaptor` class to actually capture packets from the network interface.

#### • Capture packets from the network interface:

Once you obtain an instance of `JpcapCaptor`, you can capture packets from the interface. There are two major approaches to capture packets using a `JpcapCaptor` instance : using a `callback` method, and capturing packets one-by-one.

##### Using a callback method

In this approach, you implement a callback method to process captured packets, and then pass the callback method to `Jpcap` so that `Jpcap` calls it back every time it captures a packet. Let's see how you can do this approach in detail.

First, you implement a callback method by defining a new class which implements the `PacketReceiver` interface. The `PacketReceiver` interface defines a `receivePacket()` method, so you need to implement a `receivePacket()` method in your class.

The following class implements a `receivePacket()` method which simply prints out a captured packet.

```
class PacketPrinter implements PacketReceiver {  
    //this method is called every time Jpcap captures a packet  
    public void receivePacket(Packet packet) {
```

```
//just print out a captured packet
    System.out.println(packet);
}
}
```

Then, you can call either `JpcapCaptor.processPacket()` or `JpcapCaptor.loopPacket()` methods to start capturing using the callback method. When calling `processPacket()` or `loopPacket()` method, you can also specify the number of packets to capture before the method returns. You can specify -1 to continue capturing packets infinitely.

```
JpcapCaptor captor=JpcapCaptor.openDevice(device[index],
65535, false, 20);

//call processPacket() to let Jpcap call
PacketPrinter.receivePacket() for every packet capture.
captor.processPacket(10,new PacketPrinter());
captor.close();
```

The two methods for callback, `processPacket()` and `loopPacket()`, are very similar. Usually you might want to use `processPacket()` because it supports timeout and non\_blocking mode, while `loopPacket()` doesn't.

#### Capturing packets one-by-one

Using a callback method is a little bit tricky because you don't know when the callback method is called by Jpcap. If you don't want to use a callback method, you can also capture packets using the `JpcapCaptor.getPacket()` method.

`getPacket()` method simply returns a captured packet. You can (or have to) call `getPacket()` method multiple times to capture consecutive packets.

The following sample code also prints out captured packets.

```
JpcapCaptor captor=JpcapCaptor.openDevice(device[index],
65535, false, 20);
```

```
for(int i=0;i<10;i++){  
    //capture a single packet and print it out  
    System.out.println(captor.getPacket());  
}  
captor.close();
```

- **Set capturing filter**

In Jpcap, you can set a filter so that Jpcap doesn't capture unwanted packets. For example, if you only want to capture TCP/IPv4 packets, you can set a filter as following:

```
JpcapCaptor captor=JpcapCaptor.openDevice(device[index],  
65535, false, 20);  
//set a filter to only capture TCP/IPv4 packets  
captor.setFilter("ip and tcp", true);
```

The filter expression "ip and tcp" means to to "keep only the packets that are both IPv4 and TCP and deliver them to the application".

By properly setting a filter, you can reduce the number of packets to examine, and thus can improve the performance of your application.

- **Save captured packets into a file**

You can save captured packets into a binary file so that you can later retrieve them using Jpcap or other applications which supports reading a tcpdump format file.

To save captured packets, you first need to open a file by calling `JpcapWriter.openDumpFile()` method with an instance of `JpcapCaptor` which was used to capture packets, and a `String` filename.

```
JpcapCaptor captor=JpcapCaptor.openDevice(device[index],  
65535, false, 20);  
//open a file to save captured packets  
JpcapWriter  
writer=JpcapWriter.openDumpFile(captor, "yourfilename");
```

Once you obtained an instance of `JpcapWriter` through `openDumpFile()` method, you can save captured packets using `JpcapWriter.writePacket()` method. After you saved all the packets you want to save, you need to call `JpcapWriter.close()` method to close the opened file.

The following sample code, combined with the above code, captures and saves first 100 packets captured.

```
for(int i=0;i<10;i++){
//capture a single packet
    Packet packet=captor.getPacket();
//save it into the opened file
    writer.writePacket(packet);
}
writer.close();
```

- **Read saved packets from a file:**

In Jpcap, you can read the packets you saved using `JpcapWriter` by opening the file using `JpcapCaptor.openFile()` method. Similar to `JpcapCaptor.openDevice()` method, `JpcapCaptor.openFile()` method also returns an instance of `JpcapCaptor` class. So you can use the same ways described in Capture packets from the network interface section to read packets from the file. For example, you can read and print out saved packets from a file as follows:

```
//open a file to read saved packets
JpcapCaptor captor=JpcapCaptor.openFile("yourfilename");

while(true){
    //read a packet from the opened file
    Packet packet=captor.getPacket();
    //if some error occurred or EOF has reached, break the
loop
    if(packet==null || packet==Packet.EOF) break;
    //otherwise, print out the packet
```

```

    System.out.println(packet);
}
captor.close();

```

### • Send packets through a network interface

You can also send packets to the network using Jpcap. To send a packet, you need to obtain an instance of JpcapSender by calling either `JpcapSender.openDevice()` or `JpcapCaptor.getJpcapSenderInstance()` methods.

Once you obtain an instance of JpcapSender, you can pass an instance of Packet class to `JpcapSender.sendPacket()` method.

The following sample code sends a TCP/IPv4/Ethernet packet onto a network interface.

```

//open a network interface to send a packet to
JpcapSender sender=JpcapSender.openDevice(devices[index]);

//create a TCP packet with specified port numbers, flags,
and other parameters
TCPPacket p=new
TCPPacket(12,34,56,78,false,false,false,false,true,true,true,true,10,10);

//specify IPv4 header parameters
p.setIPv4Parameter(0,false,false,false,0,false,false,false,0,1010101,100,IPPacket.IPPROTO_TCP,
InetAddress.getByName("www.microsoft.com"),InetAddress.get
ByName("www.google.com"));

//set the data field of the packet
p.data=("data").getBytes();

//create an Ethernet packet (frame)

```

```

EthernetPacket ether=new EthernetPacket();
//set frame type as IP
ether.frameType=EthernetPacket.ETHERTYPE_IP;
//set source and destination MAC addresses
ether.src_mac=new
byte[] {(byte)0, (byte)1, (byte)2, (byte)3, (byte)4, (byte)5};
ether.dst_mac=new
byte[] {(byte)0, (byte)6, (byte)7, (byte)8, (byte)9, (byte)10};

//set the datalink frame of the packet p as ether
p.datalink=ether;
//send the packet p
sender.sendPacket(p);
sender.close();

```

## 4.8 Detection (Port Scan/Slow Port Scan):

### 4.8.1 Detection Port Scan:

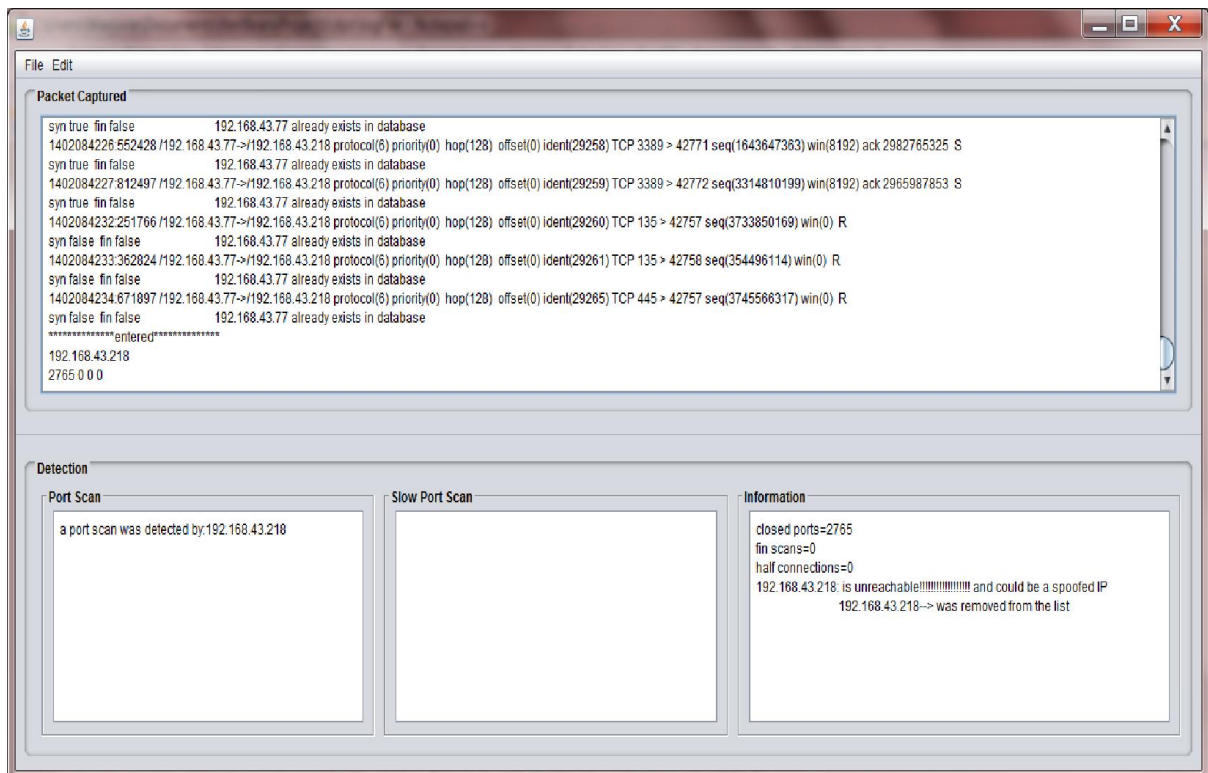
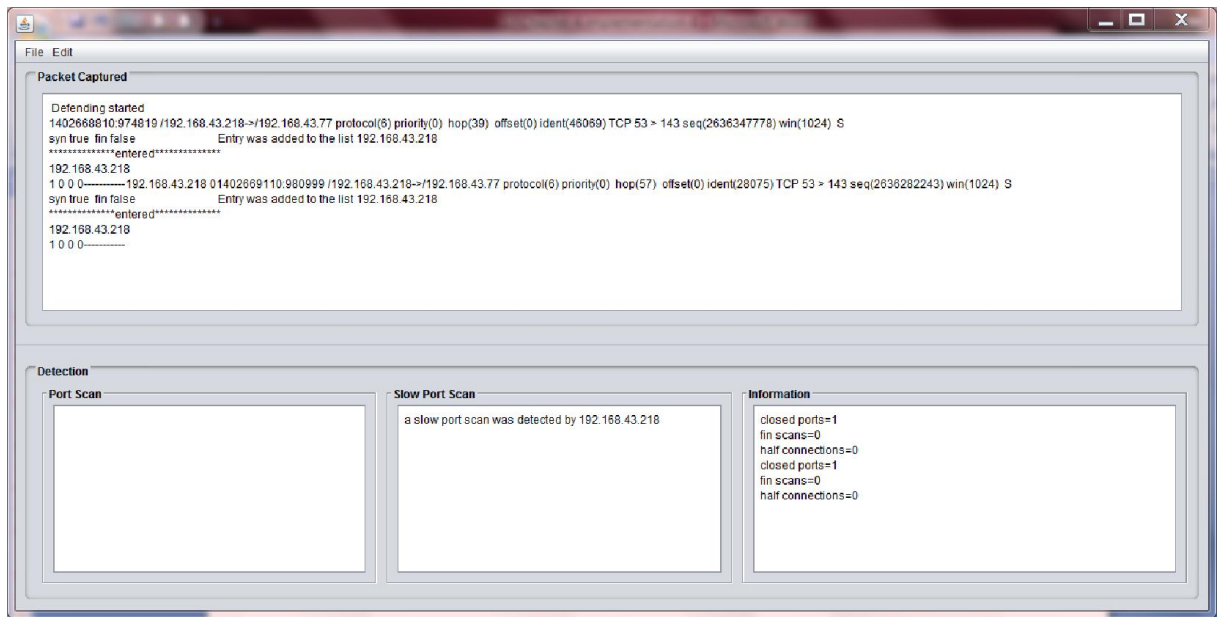


Figure 4.7 detect port scan

## 4.8.2 Detection Slow Port Scan:



**Figure 4.8** detect slow port scan

## 4.9 Conclusion:

In this chapter we presented the implementation of our approach and show the results of scan detector application. We scan ports of victim machine with the tool Nmap by using the three techniques of scanning: TCP SYN scan, connect scan and FIN scan, then capture packet with Jpcap after that detect port Scan and slow port scan on victim machine.