SYLLABUS

20ITC22 **NETWORKS AND SECURITY LAB**

Instruction	3 L Hours per week
Duration of SEE	3 Hours
CIE	50 Marks
SEE	50 Marks
Credits	1.5

Course Objectives:

1. To provide knowledge required to implement error detection, network routing algorithms and analyse traffic flow of the contents.
2. To present Client/Server applications based on TCP, UDP and SMTP using Java Socket API.
3. To facilitate knowledge required to handle rootkits, capture packets & interfaces.
4. To deal with the configuration and use of technologies designed to segregate the organization‘s systems from the insecure Network.
5. To familiarize with security policies of tcpdump, dumpcap and pentest tools using nmap.

Course Outcomes:

Upon successful completion of this course, students will be able to:

1. Identify Errors using CRC, Implement routing algorithms and congestion control algorithms.
2. Demonstrate client-server communication using TCP, UDP protocols.
3. Experiment with rootkits to detect malware, wire shark to capture the packets and interfaces.
4. Make use of tools, techniques to protect the system from attacks.
5. Acquire thorough knowledge on tcpdump, dumpcap and nmap.

Mapping of Course Outcomes with Program Outcomes and Program Specific Outcomes:

	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO1	PSO2	PSO3
CO 1	3	3	2	2	3	-	-	-	-	-	-	1	3	3	3
CO 2	3	3	2	2	3	-	-	-	-	-	-	1	3	3	2
CO 3	2	2	2	3	3	3	1	-	-	-	-	-	3	3	3
CO 4	2	2	2	3	3	3	1	-	-	-	-	-	3	3	2
CO 5	2	2	2	3	3	3	1	-	-	-	-	-	3	3	3

LIST OF PROGRAMS

1. Implement CRC Error detection technique.
2. Implement Dijkstra‘s and Distance Vector routing algorithms.
3. Implement congestion control using leaky bucket & Token bucket Algorithms.
4. Implementation of TCP (Server and client) and UDP (Server and client) .
5. Implement SMTP protocol.
6. Installation of rootkits and study about the variety of options.
7. Implement Wireshark to capture the packets and interfaces.
8. Demonstrate intrusion detection system using SNORT tool or any other software.
9. Setup a honey pot and monitor the honeypot on network using KF sensor.
10. Demonstrate how to managing securing policies using tcpdump, dumpcap using Wireshark.
11. Demonstration of pentest tools using Nmap, Wireshark.

Note:- Implement Programs 1 to 5 in C or Java

Text Books:

1. Andrew S. Tanenbaum, Computer Networks, Pearson Education,6th Edition,2021.
1. Michael Gregg, "Build Your Own Security Lab", Wiley Publishing, Inc., 2008.

2. Michael E.whitman, Herbert J.Mattord, Andrew Green ,"Hands on Information Security lab manual", Cengage Learning, Fourth edition, December 27, 2013.

Suggested Reading:

1. James F. Kurose, Keith W. Ross, "Computer Networking – A Top-Down Approach Featuring the Internet", 8th Edition, Pearson Education, 2022.
2. Alfred Basta, Wolf Halton, "Computer Security, concepts, issues and implementation", Cengage Learning India Pvt. Ltd, 2008.

Web Resources:

1. https://nmap.org
2. https://www.snort.org
3. https://www.wireshark.org
4. http://www.keyfocus.net/kfsensor/
5. http://www.gmer.net/

Cyclic Redundancy Check

CRC or Cyclic Redundancy Check is a method of detecting accidental changes/errors in the communication channel.

CRC uses **Generator Polynomial** which is available on both sender and receiver side. An example generator polynomial is of the form like $x^3 + x + 1$. This generator polynomial represents key 1011. Another example is $x^2 + 1$ that represents key 101.

```
n : Number of bits in data to be sent
    from sender side.
k : Number of bits in the key obtained
    from generator polynomial.
```

Sender Side (Generation of Encoded Data from Data and Generator Polynomial (or Key)):

1. The binary data is first augmented by adding k-1 zeros in the end of the data

2. Use ***modulo-2 binary division*** to divide binary data by the key and store remainder of division.
3. Append the remainder at the end of the data to form the encoded data and send the same.

Receiver Side (Check if there are errors introduced in transmission)
Perform modulo-2 division again and if the remainder is 0, then there are no errors

Modulo 2 Division:

The process of modulo-2 binary division is the same as the familiar division process we use for decimal numbers. Just that instead of subtraction, we use XOR here.

- In each step, a copy of the divisor (or data) is XORed with the k bits of the dividend (or key).
- The result of the XOR operation (remainder) is (n-1) bits, which is used for the next step after 1 extra bit is pulled down to make it n bits long.
- When there are no bits left to pull down, we have a result. The (n-1)-bit remainder which is appended at the sender side.

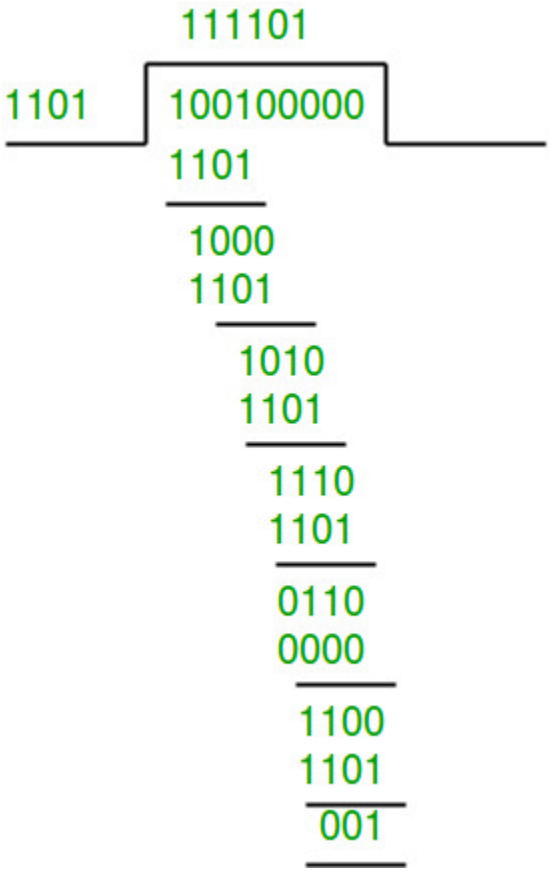
Illustration:

Example 1 (No error in transmission):

```
Data word to be sent - 100100


Key - 1101 [ Or generator polynomial x3 + x2 + 1]
```

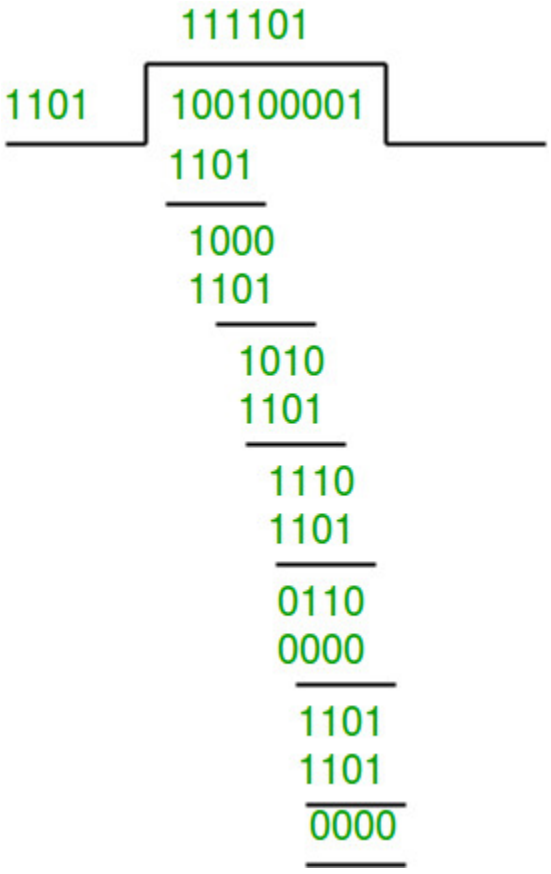
Sender Side:



Therefore, the remainder is 001 and hence the encoded data sent is 100100001.

Receiver Side:

Code word received at the receiver side 100100001



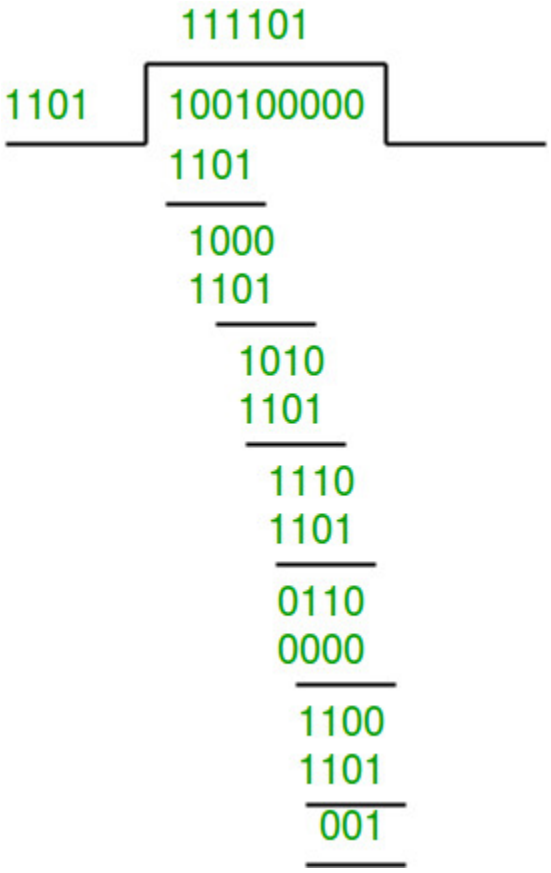
Therefore, the remainder is all zeros. Hence, the data received has no error.

Example 2: (Error in transmission)

Data word to be sent - 100100

Key - 1101

Sender Side:



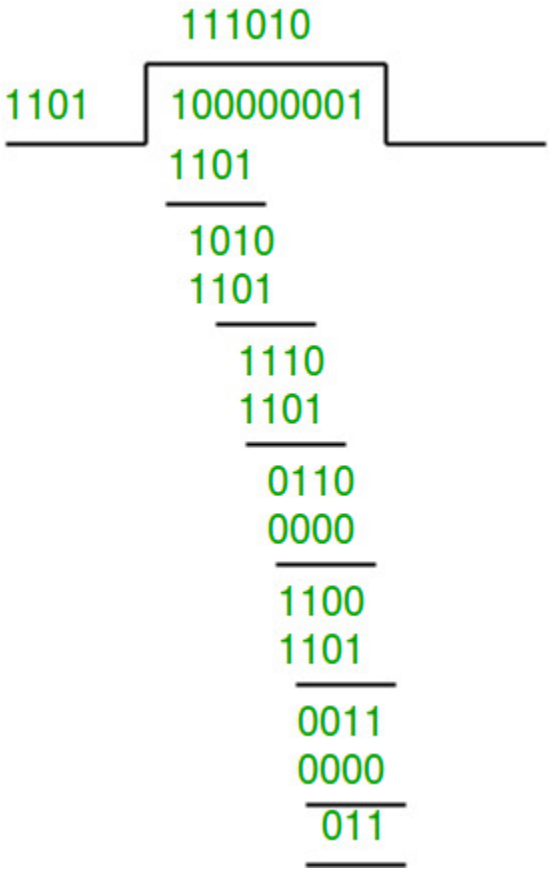
Therefore, the remainder is 001 and hence the

code word sent is 100100001.

Receiver Side

Let there be an error in transmission media

Code word received at the receiver side - 100000001



Since the remainder is not all zeroes, the error is detected at the receiver side.

Client Side Programming Establish a Socket Connection

To connect to other machine we need a socket connection. A socket connection means the two machines have information about each other's network location (IP Address) and TCP port.The java.net.Socket class represents a Socket. To open a socket:

```
Socket socket = new Socket("127.0.0.1", 5000)  ;
```

First argument – **IP address of Server**. (127.0.0.1 is the IP address of localhost, where code will run on single stand-alone machine).

Second argument – **TCP Port**. (Just a number representing which application to run on a server. For example, HTTP runs on port 80. Port number can be from 0 to 65535) To communicate over a socket connection, streams are used to both input and output the data.

Closing the connection

The socket connection is closed explicitly once the message to server is sent.

In the program, Client keeps reading input from user and sends to the server until "Over" is typed.

Java Implementation

```
// A Java program for a Client

import
java.net.*;
import   java.io.*;
public       class
Client

{

   // initialize socket and input output
   streams   private   Socket   socket
   =   null;   private   DataInputStream
   input          =       null;       private
   DataOutputStream out                =
   null;

    // constructor to put ip address and port

   public Client(String address, int port)

   {

      // establish a connection

      try

      {

         socket = new Socket(address, port); System.out.println("Connected");

          // takes input from terminal

         input = new DataInputStream(System.in);
```

```
    // sends output to the socket
   out        = new DataOutputStream(socket.getOutputStream());
}
catch(UnknownHostException u)
{
   System.out.println(u);
}
catch(IOException i)
{
   System.out.println(i);
}
 // string to read message from input
String line = "";
 // keep reading until "Over" is input
while (!line.equals("Over"))
{
   try
   {
      line    =    input.readLine();
      out.writeUTF(line);
   }
   catch(IOException i)
```

```
            {
                System.out.println(i);
            }
        }
        // close the connection
        try
        {
            input.clos
            e();
            out.close(
            );
            socket.cl
            ose();
        }
        catch(IOException i)
        {
            System.out.println(i);
        }}
    public static void main(String args[])
    {
        Client client = new Client("127.0.0.1", 5000);
    }
}
```

Server Programming Establish a Socket Connection

To write a server application two sockets are needed.

·A ServerSocket which waits for the client requests (when a client makes a new Socket())

A plain old Socket socket to use for communication with the client.

Communication

getOutputStream() method is used to send the output through the socket.

Close the Connection

After finishing, it is important to close the connection by closing the socket as well as input/output streams.

```
// A Java program for a Server

import java.net.*;

import java.io.*;

public class Server

{

   //initialize socket and input stream
   private                    Socket
   socket=          null;          private
   ServerSocket                server
   =  null;  private  DataInputStream
   in       = null;

   // constructor with port

   public Server(int port)

   {

      // starts server and waits for a connection

      try

      {

         server          =          new
         ServerSocket(port);
         System.out.println("Server
         started");
```

```
System.out.println("Waiting for a client ...");

socket = server.accept(); System.out.println("Client accepted");
// takes input from the
client socket in = new
DataInputStream(

  new BufferedInputStream(socket.getInputStream()));


String line = "";

// reads message from client until "Over" is sent

while (!line.equals("Over"))

{

  try

  {

    line = in.readUTF();
    System.out.println(line);



  }

  catch(IOException i)

  {

    System.out.println(i);

  }

}
```

```
            System.out.println("Closing connection");
            //          close
            connection
            socket.close(
            ); in.close();
        }
        catch(IOException i)
        {
            System.out.println(i);


        }
    }

    public static void main(String args[])
    {
        Server server = new Server(5000);
    }
}
```

Important Points

Server application makes a ServerSocket on a specific port which is 5000. This starts our Server listening for client requests coming in for port 5000.

- Then Server makes a new Socket to communicate with the client.

```
socket = server.accept()
```

- The accept() method blocks(just sits there) until a client connects to the server.
    - Then we take input from the socket using getInputStream() method. Our Server keeps receiving messages until the Client sends “Over”.
- After we’re done we close the connection by closing the socket and the input stream.
    - To run the Client and Server application on your machine, compile both of them. Then first run the server application and then run the Client application.

To run on Terminal or Command Prompt

Open two windows one for Server and another for Client

1. First run the Server application as ,

```
$ java Server
```

Server started

Waiting for a client …

2. Then run the Client application on another terminal as,

```
$ java Client
```

It will show – Connected and the server accepts the client and shows, Client accepted

3. Then you can start typing messages in the Client window. Here is a sample input to the Client

```
Hello
I made my first socket connection Over
```

Which the Server simultaneously receives and shows,

Notice that sending “Over” closes the connection between the Client and the Server just like said before.

Working with UDP DatagramSockets in Java

DatagramSockets are Java's mechanism for network communication via UDP instead of TCP. Java provides DatagramSocket to communicate over UDP instead of TCP. It is also built on top of IP. DatagramSockets can be used to both send and receive packets over the Internet.

One of the examples where UDP is preferred over TCP is the live coverage of TV channels. In this aspect, we want to transmit as many frames to live audience as possible not worrying about the loss of one or two frames. TCP being a reliable protocol add its own overhead while transmission.

Another example where UDP is preferred is online multiplayer gaming. In games like counter-strike or call of duty, it is not necessary to relay all the information but the most important ones. It should also be noted that most of the applications in real life uses careful blend of both UDP and TCP; transmitting the critical data over TCP and rest of the data via UDP.

This article is a simple implementation of one-sided client-server program wherein the client sends messages to server and server just prints it until the client sends "bye".

Java Datagram programming model Steps

1. **Creation of DatagramSocket:-** First, a datagramSocket object is created to carry the packet to the destination and to receive it whenever the server sends any data. To create a datagramSocket following constructors can be used:

.

protected DatagramSocket DatagramSocket():

1.

.

- **protected DatagramSocket DatagramSocket(int port):-**

protected DatagramSocket DatagramSocket(int port, InetAddress inetaddress):-

2. **Creation of DatagramPacket:** In this step, the packet for sending/receiving data via a datagramSocket is created.

- Constructor to send data: **DatagramPacket(byte buf[], int length, InetAddress inetaddress, int port):-**

.

- Constructor to receive the data:

DatagramPacket(byte buf[], int length):-

- **Invoke a send() or receive() call on socket object**

Client Side Implementation

```
// Java program to illustrate Client side

// Implementation using DatagramSocket

import  java.io.IOException;
import
java.net.DatagramPacket;
import
java.net.DatagramSocket;
import
java.net.InetAddress;
import java.util.Scanner;


public class udpBaseClient_2

{

   public static void main(String args[]) throws IOException

   {

      Scanner sc = new Scanner(System.in);



      // Step 1:Create the socket object for
```

```
// carrying the data.
DatagramSocket ds = new DatagramSocket();


InetAddress ip = InetAddress.getLocalHost();
byte buf[] = null;


// loop while user not enters "bye"
while (true)
{
   String inp = sc.nextLine();


   // convert the String input into the
   byte array. buf = inp.getBytes();


   // Step 2 : Create the datagramPacket for sending
```

```
        //        the        data.
        DatagramPacket
        DpSend =

            new DatagramPacket(buf, buf.length, ip, 1234);



        // Step 3 : invoke the send call to actually send

        //   the   data.
        ds.send(Dp
        Send);


        // break the loop if user enters "bye"

        if (inp.equals("bye"))

          break;

      }

    }

}
```

Output:

```
/ Java program to illustrate Server side

// Implementation using DatagramSocket

import  java.io.IOException;
import
java.net.DatagramPacket;
import
java.net.DatagramSocket;
import
java.net.InetAddress;
import
java.net.SocketException;


public class udpBaseServer_2

{

   public static void main(String[] args) throws IOException

   {

      // Step 1 : Create a socket to listen at
      port 1234 DatagramSocket ds = new
      DatagramSocket(1234); byte[] receive =
      new byte[65535];
```

```
DatagramPacket   DpReceive
= null; while (true)

{


   // Step 2 : create a DatgramPacket to receive the
   data. DpReceive = new DatagramPacket(receive,
   receive.length);


   // Step 3 : revieve the data in
   byte                    buffer.
   ds.receive(DpReceive);


   System.out.println("Client:-" + data(receive));



   // Exit the server if the client sends "bye"

   if (data(receive).toString().equals("bye"))


   {
      System.out.println("Client sent bye.... EXITING");
```

```
            break;
        }


        // Clear the buffer after every
        message.  receive  =  new
        byte[65535];
    }
}


// A utility method to convert the byte array
// data into a string representation.
public static StringBuilder data(byte[] a)
{
        if  (a  ==
            null)
            retur
            n
            null;
    StringBuilder ret = new StringBuilder();
    int i = 0;
    while (a[i] != 0)
```

```
        {
            ret.append((char
            ) a[i]); i++;
        }
        return ret;
    }
}
```

In a nutshell, we can summarize the steps of sending and recieving data over UDP as follows:-

1. For sending a packet via UDP, we should know 4 things, **the message to send, its length, ipaddress of destination, port at which destination is listening.**
2. Once we know all these things, we can create the socket object for carrying the packets and packets which actually possess the data.
3. Invoke send()/receive() call for actually sending/recieving packets.
4. Extract the data from the received packet.

Output:

Note:- In order to test the above programs on the system, Please make sure that you run the server program first and then the client one. Make sure you are in the client console and from there keep on typing your messages each followed with a carriage return. Every time you send a message you will be redirected to the server console depending on your environment settings. If not redirected automatically, switch to server console to make sure all your messages are received. Finally to terminate the communication, type "bye" (without quotes) and hit enter.