The book is organized into two modules: In the first module, we present a tutorial on socket programming in Java, illustrating complete examples for simplex and duplex communications with both connectionless datagram and connection-oriented stream-mode sockets. In addition, this module explains in detail, with examples, the differences between a concurrent server and iterative server and the use of the Multicast socket API. In the second module, we present the source code analysis of a file reader connection-oriented server socket Java program, to illustrate the identification, impact analysis and solutions to remove the following important software security vulnerabilities: (1) Resource Injection, (2) Path Manipulation, (3) System Information Leak, (4) Denial of Service and (5) Unreleased Resource vulnerabilities. We analyze the reason for these vulnerabilities to occur in the program, discuss the impact of leaving them unattended, and propose solutions to remove each of these vulnerabilities from the program. The proposed solutions are very generic in nature, and can be suitably modified to correct any such vulnerabilities in software developed in any other programming language.

Natarajan Meghanathan

A Tutorial on Java Socket Programming and Source Code Analysis

Complete Java Source Code Examples and Practice Exercises: Supplement for Computer Networks & Software Security Courses

Natarajan Meghanathan

Dr. Natarajan Meghanathan is an Associate Professor of Computer Science at Jackson State University, MS, USA. He has published more than 140 peer-reviewed papers and has worked on several federally funded grants. His teaching and research interests are Computer Networks, Algorithm Design & Analysis, Systems & Software Security and Cloud Computing.

978-3-659-31445-2

LAP LAMBERT Academic Publishing

Natarajan Meghanathan

A Tutorial on Java Socket Programming and Source Code Analysis

Natarajan Meghanathan

A Tutorial on Java Socket Programming and Source Code Analysis

Complete Java Source Code Examples and Practice Exercises: Supplement for Computer Networks & Software Security Courses

LAP LAMBERT Academic Publishing

Impressum / Imprint
Bibliografische Information der Deutschen Nationalbibliothek: Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.
Alle in diesem Buch genannten Marken und Produktnamen unterliegen warenzeichen-, marken- oder patentrechtlichem Schutz bzw. sind Warenzeichen oder eingetragene Warenzeichen der jeweiligen Inhaber. Die Wiedergabe von Marken, Produktnamen, Gebrauchsnamen, Handelsnamen, Warenbezeichnungen u.s.w. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutzgesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Bibliographic information published by the Deutsche Nationalbibliothek: The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.
Any brand names and product names mentioned in this book are subject to trademark, brand or patent protection and are trademarks or registered trademarks of their respective holders. The use of brand names, product names, common names, trade names, product descriptions etc. even without a particular marking in this works is in no way to be construed to mean that such names may be regarded as unrestricted in respect of trademark and brand protection legislation and could thus be used by anyone.

Coverbild / Cover image: www.ingimage.com

Verlag / Publisher:
LAP LAMBERT Academic Publishing
ist ein Imprint der / is a trademark of
AV Akademikerverlag GmbH & Co. KG
Heinrich-Böcking-Str. 6-8, 66121 Saarbrücken, Deutschland / Germany
Email: info@lap-publishing.com

Herstellung: siehe letzte Seite /
Printed at: see last page
ISBN: 978-3-659-31445-2

TABLE OF CONTENTS

MODULE I

A TUTORIAL ON JAVA SOCKET PROGRAMMING

1. Introduction

Interprocess communication (IPC) is the backbone of distributed computing. Processes are runtime representations of a program. IPC refers to the ability for separate, independent processes to communicate among themselves to collaborate on a task. The following figure illustrates basic IPC:

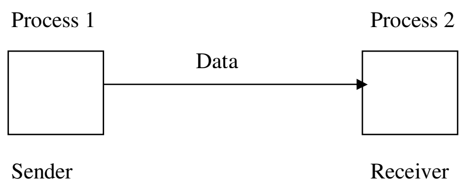


Figure 1: Interprocess Communication

Two or more processes engage in a protocol – a set of rules to be observed by the participants for data communication. A process can be a sender at some instant of the communication and can be a receiver of the data at another instant of the communication. When data is sent from one process to another single process, the communication is said to be unicast. When data is sent from one process to more than one process at the same time, the communication is said to be multicast. Note that multicast is not multiple unicasts.

Most of the operating systems (OS) like UNIX and Windows provide facilities for IPC. The system-level IPC facilities include message queues, semaphores and shared memory. It is possible to directly develop network software using these system-level facilities. Examples are network device drivers and system evaluation programs. On the other hand, the complexities of the applications require the use of some form of abstraction to spare the programmer of the system-level details. An IPC application programming interface (API) abstracts the details and intricacies of the system-level facilities and allows the programmer to concentrate on the application logic.

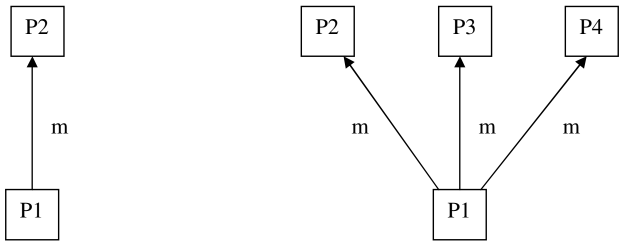


Figure 2: Unicast Vs. Multicast

The Socket API is a low-level programming facility for implementing IPC. The upper-layer facilities are built on top of the operations provided by the Socket API. The Socket API was originally provided as part of the Berkeley UNIX OS, but has been later ported to all operating systems including Sun Solaris and Windows systems. The Socket API provides a programming construct called a "socket". A process wishing to communicate with another process must create an instance or instantiate a socket. Two processes wishing to communicate can instantiate sockets and then issue operations provided by the API to send and receive data. Note that in network parlance, a packet is the unit of data transmitted over the network. Each packet contains the data (payload) and some control information (header) that includes the destination address.

A socket is uniquely identified by the IP address of the machine and the port number at which the socket is opened (i.e. bound to). Port numbers are allocated 16 bits in the packet headers and thus can be at most 66535. Well-known processes like FTP, HTTP and etc., have their sockets opened on dedicated port numbers (less than or equal to 1024). Hence, sockets corresponding to user-defined processes have to be run on port numbers greater than 1024.

In this chapter, we will discuss two types of sockets – “connectionless” and “connection-oriented” for unicast communication, multicast sockets and several programming examples to illustrate different types of communication using these sockets. All of the programming examples are illustrated in Java.

2. Types of Sockets

The User Datagram Protocol (UDP) transports packets in a connectionless manner [1]. In a connectionless communication, each data packet (also called datagram) is addressed and routed individually and may arrive at the receiver in any order. For example, if process 1 on host A sends datagrams m1 and m2 successively to process 2 on host B, the datagrams may be transported on the network through different routes and may arrive at the destination in any of the two orders: m1, m2 or m2, m1.

The Transmission Control Protocol (TCP) is connection-oriented and transports a stream of data over a logical connection established between the sender and the receiver [1]. As a result, data sent from a sender to a receiver is guaranteed to be received in the order they were sent. In the above example, messages m1 and m2 are delivered to process 2 on host B in the same order they were sent from process 1 on host A.

A socket programming construct can use either UDP or TCP transport protocols. Sockets that use UDP for transport of packets are called “datagram” sockets and sockets that use TCP for transport are called “stream” sockets.

3. The Connectionless Datagram Socket

In Java, two classes are provided for the datagram socket API: (a) The DatagramSocket class for the sockets (b) The DatagramPacket class for the packets exchanged. A process wishing to send or receive data using the datagram socket API must instantiate a DatagramSocket object, which is bound to a UDP port of the machine and local to the process.

To send a datagram to another process, the sender process must instantiate a DatagramPacket object that carries the following information: (1) a reference to a byte array that contains the payload data and (2) the destination address (the host ID and port number to which the receiver process' DatagramSocket object is bound).

Sender Program	**Receiver Program**
Create a DatagramSocket object and bind it to any local port;	Create a DatagramSocket object and bind it to a specific local port;
Place the data to send in a byte array;	Create a byte array for receiving the data;
Create a DatagramPacket object, specifying the data array and the receiver's address;	Create a DatagramPacket object, specifying the data array.
Invoke the send method of the DatagramSocket object and pass as argument, a reference to the DatagramPacket object.	Invoke the receive method of the socket with a reference to the DatagramPacket object.

Figure 3: Program flow in the sender and receiver process (adapted from [2])

At the receiving process, a DatagramSocket object must be instantiated and bound to a local port – this port should correspond to the port number carried in the datagram packet of the sender. To receive datagrams sent to the socket, the receiving process must instantiate a DatagramPacket object that references a byte array and call the receive method of the DatagramSocket object, specifying as argument, a reference to the DatagramPacket object. The program flow in the sender and receiver process is illustrated in Figure 3 and the key methods used for communication using connectionless sockets are summarized in Table 1.

Table 1: Key Methods of the DatagramSocket API (adapted from [3])

No.	Constructor/ Method	Description
	DatagramSocket class	
1	DatagramSocket( )	Constructs an object of class DatagramSocket and binds the object to any available port on the local host machine
2	DatagramSocket(int port)	Constructs an object of class DatagramSocket and binds it to the specified port on the local host machine
3	DatagramSocket (int port, InetAddress addr)	Constructs an object of class DatagramSocket and binds it to the specified local address and port
4	void close( )	Closes the datagram socket
5	void connect(InetAddress address, int port)	Connects the datagram socket to the specified remote address and port number on the machine with that address
6	InetAddress getLocalAddress( )	Returns the local InetAddress to which the socket is connected.
7	int getLocalPort( )	Returns the port number on the local host to which the datagram socket is bound
8	InetAddress getInetAddress( )	Returns the IP address to which the datagram socket is connected to at the remote side.
9	int getPort( )	Returns the port number at the remote side of the socket
10	void receive(DatagramPacket packet)	Receives a datagram packet object from this socket
11	void send(DatagramPacket packet)	Sends a datagram packet object from this socket
12	void setSoTimeout(int timeout)	Set the timeout value for the socket, in milliseconds
	DatagramPacket class	
13	DatagramPacket(byte[ ] buf, int length, InetAddress, int port)	Constructs a datagram packet object with the contents stored in a byte array, buf, of specified length to a machine with the specified IP address and port number
14	InetAddress getAddress( )	Returns the IP address of the machine at the remote side to which the datagram is being sent or from which the datagram was received
15	byte [ ] getData( )	Returns the data buffer stored in the packet as a byte array
16	int getLength( )	Returns the length of the data buffer in the datagram packet

		sent or received
17	int getPort( )	Returns the port number to which the datagram socket is bound to which the datagram is being sent or from which the datagram is received
18	void setData(byte [ ])	Sets the data buffer for the datagram packet
19	void setAddress(InetAddress iaddr)	Sets the datagram packet with the IP address of the remote machine to which the packet is being sent
20	void setPort(int port)	Sets the datagram packet with the port number of the datagram socket at the remote host to which the packet is sent

With connectionless sockets, a DatagramSocket object bound to a process can be used to send datagrams to different destinations. Also, multiple processes can simultaneously send datagrams to the same socket bound to a receiving process. In such a situation, the order of the arrival of the datagrams may not be consistent with the order they were sent from the different processes. Note that in connection-oriented or connectionless Socket APIs, the send operations are non-blocking and the receive operations are blocking. A process continues its execution after the issuance of a *send* method call. On the other hand, once a process calls the *receive* method on a socket, the process is suspended until a datagram is received. To avoid indefinite blocking, the *setSoTimeout* method can be called on the DatagramSocket object.

We now present several sample programs to illustrate the use of the DatagramSocket and DatagramPacket API. Note that in all these exercises, the receiver programs should be started first before starting the sender program. This is analogous to the fact that in any conversation, a receiver should be tuned and willing to hear and receive the information spoken (sent) by the sender. If the receiver is not turned on, then whatever the message was sent will be dropped at the receiving side. The following code segments illustrate the code to send to a datagram packet from one host IP address and port number and receive the same packet at another IP address and port number. Though the sender and receiver programs are normally run at two different hosts, sometimes one can test the correctness of their code by running the two programs on the same host using '*localhost*' as the name of the host at remote side. This is the approach we use in this book chapter. For all socket programs, the package java.net should be imported; and very often we need to also import the java.io package to do any input/output with the sockets. Of course, for any file access, we also need to import the java.io package. Also, since many of the methods (for both the Connectionless and Stream-mode API) could raise exceptions, it is recommended to put the entire code inside a *try-catch* block.

3.1 *Example Program to Send and Receive a Message using Connectionless Sockets*

```
--------------------------------------------------------------------------------
import java.net.*;
import java.io.*;
class datagramReceiver{
  public static void main(String[ ] args){
    try{
      int MAX_LEN = 40;
      int localPortNum = Integer.parseInt(args[0]);
      DatagramSocket mySocket  = new DatagramSocket(localPortNum);
      byte[] buffer = new byte[MAX_LEN];
      DatagramPacket packet = new DatagramPacket(buffer, MAX_LEN);
      mySocket.receive(packet);
      String message = new String(buffer);
      System.out.println(message);
      mySocket.close( );
     }
    catch(Exception e){e.printStackTrace( );}
   }
}
--------------------------------------------------------------------------------
```

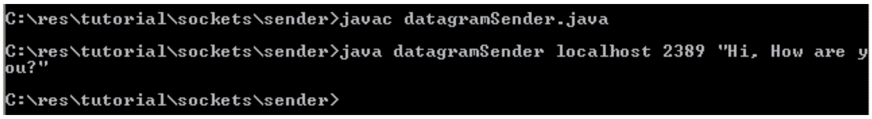
Figure 4: Program to Receive a Single Datagram Packet

```
------------------------------------------------------------------------------------------------------------
import java.net.*;
import java.io.*;

class datagramSender{
  public static void main(String[ ] args){
    try{
       InetAddress receiverHost = InetAddress.getByName(args[0]);
       int receiverPort = Integer.parseInt(args[1]);
       String message = args[2];
       DatagramSocket mySocket = new DatagramSocket( );
       byte[] buffer = message.getBytes( );
       DatagramPacket packet = new DatagramPacket(buffer, buffer.length, receiverHost,
                                                                              receiverPort);
       mySocket.send(packet);
       mySocket.close( );
     }
    catch(Exception e){ e.printStackTrace( ); }
   }
}
------------------------------------------------------------------------------------------------------------
```

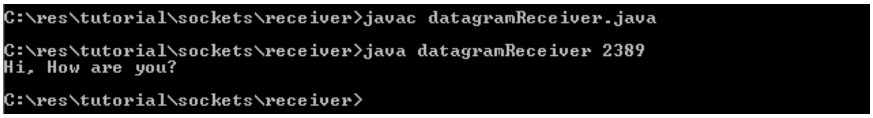
Figure 5: Program to Send a Single Datagram Packet

The datagram receiver (datagramReceiver.java) program illustrated below can receive a datagram packet of size at most 40 bytes. As explained before, the receive( ) method call on the
DatagramSocket is a binding call. Once a datagram packet arrives at the host at the specified local port number at which the socket is opened, the receiver program will proceed further – extract the bytes stored in the datagram packet and prints the contents as a String. The local port number at which the receiver should open its datagram socket is passed as an input command-line parameter and this port number should also be known to the sender so that the message can be sent to the same port number. The datagram sender (datagramSender.java) program creates a DatagramPacket object and sets its destination IP address to the IP address of the

remote host, the port number at which the message is expected to receive and the actual message. The sender program has nothing much to do after sending the message and hence the socket is closed. Similarly, the socket at the receiver side is also closed after receiving and printing the message. Figure 6 is a screenshot of the execution and output obtained for the code segments illustrated in Figures 4 and 5. Note that the datagramReceiver program should be started first. The maximum size of the message that could be received is 40 bytes (characters) as set by the datagramReceiver.



```
C:\res\tutorial\sockets\sender>javac datagramSender.java

C:\res\tutorial\sockets\sender>java datagramSender localhost 2389 "Hi, How are y
ou?"

C:\res\tutorial\sockets\sender>
```



```
C:\res\tutorial\sockets\receiver>javac datagramReceiver.java

C:\res\tutorial\sockets\receiver>java datagramReceiver 2389
Hi, How are you?

C:\res\tutorial\sockets\receiver>
```

Figure 6: Screenshots of the Execution of datagramSender.java and datagramReceiver.java

3.2 *Example Program to Send and Receive a Message in both Directions (Duplex Communication) using Connectionless Sockets*

The program illustrated in this example is an extension of the program in Section 3.1. Here, we describe two programs – datagramSenderReceiver.java (refer Figure 7) and datagramReceiverSender.java (refer Figure 8). The sender-receiver program will first send a message and then wait for a response for the message. Accordingly, the first half of the sender-receiver program would be to send a message and the second half of the program would be to receive a response. Note that to get the response, the sender-receiver program should invoke the receive( )

method on the same DatagramSocket object and port number that were used to send the message. The receiver-sender program will have to first receive the message and then respond to the sender of the message. It extracts the sender information from the Datagram Packet object received and uses the sender IP address and port number retrieved from the Datagram Packet received as the destination IP address and port number for the response Datagram Packet sent. This is analogous to replying to a mail or an email using the sender information in the mail (i.e., reply to the same address from which the message was sent). The above logic could be used to develop chatting programs using connectionless sockets. The maximum size of the messages that could be sent and received is 60 bytes in this example.

```
------------------------------------------------------------------------------------------------------------
import java.net.*;
import java.io.*;

class datagramSenderReceiver{
  public static void main(String[ ] args){
   try{
      InetAddress receiverHost = InetAddress.getByName(args[0]);
      int receiverPort = Integer.parseInt(args[1]);
      String message = args[2];

      DatagramSocket mySocket = new DatagramSocket( );
      byte[] sendBuffer = message.getBytes( );
      DatagramPacket packet = new DatagramPacket(sendBuffer, sendBuffer.length,
                                                        receiverHost, receiverPort);
      mySocket.send(packet);

      // to receive a message

      int MESSAGE_LEN = 60;
      byte[ ] recvBuffer = new byte[MESSAGE_LEN];
```

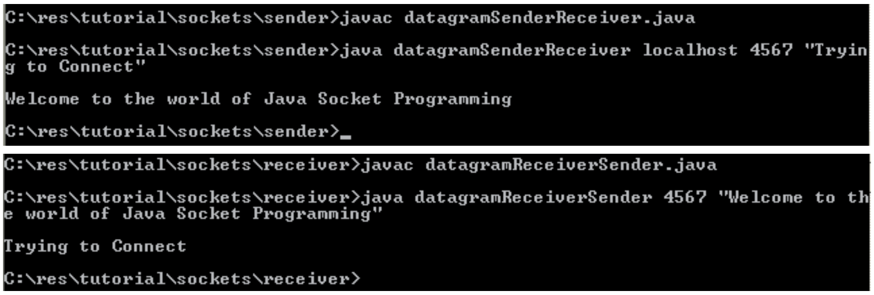
```
        DatagramPacket datagram = new DatagramPacket(recvBuffer, MESSAGE_LEN);
        mySocket.receive(datagram);
        String recvdString = new String(recvBuffer);
        System.out.println("\n"+recvdString);

        mySocket.close( );
      }
    catch(Exception e){ e.printStackTrace( ); }
   }
}
```

Figure 7: Datagram Sender and Receiver Program (Sends First, Receives Next)

```
import java.net.*;
import java.io.*;

class datagramReceiverSender{
   public static void main(String[ ] args){
     try{
        int MAX_LEN = 60;
        int localPortNum = Integer.parseInt(args[0]);
        DatagramSocket mySocket  = new DatagramSocket(Integer.parseInt(localPortNum);
        byte[ ] recvBuffer = new byte[MAX_LEN];
        DatagramPacket packet = new DatagramPacket(recvBuffer, MAX_LEN);
        mySocket.receive(packet);
        String message = new String(recvBuffer);

        System.out.println("\n"+message);

        // to reply back to sender
        InetAddress senderAddress = packet.getAddress( );
        int senderPort = packet.getPort( );
        String messageToSend = args[1];
        byte[ ] sendBuffer = messageToSend.getBytes();
        DatagramPacket datagram = new DatagramPacket(sendBuffer, sendBuffer.length,
                                                      senderAddress, senderPort);
```

```
      mySocket.send(datagram);
      mySocket.close( );
    }
   catch(Exception e){e.printStackTrace( );}
  }
}
```

Figure 8: Datagram Receiver and Sender Program (Receives First, Sends Next)



```
C:\res\tutorial\sockets\sender>javac datagramSenderReceiver.java

C:\res\tutorial\sockets\sender>java datagramSenderReceiver localhost 4567 "Tryin
g to Connect"

Welcome to the world of Java Socket Programming

C:\res\tutorial\sockets\sender>_
```

```
C:\res\tutorial\sockets\receiver>javac datagramReceiverSender.java

C:\res\tutorial\sockets\receiver>java datagramReceiverSender 4567 "Welcome to th
e world of Java Socket Programming"

Trying to Connect

C:\res\tutorial\sockets\receiver>
```

Figure 9: Screenshots of the Execution of datagramSenderReceiver.java and datagramReceiverSender.java

In the above example, the datagramReceiverSender.java program is waiting at a local port number of 4567 to greet with a message "Welcome to the world of Java Socket Programming" for an incoming connection request message from any host. When the datagramSenderReceiver.java program attempts to connect at port number 4567 with a "Trying to Connect" request, it gets the welcome message as the response from the other end.

4 The Connection-Oriented (Stream-Mode) Socket

The Connection-Oriented Sockets are based on the stream-mode I/O model of the UNIX Operating System – data is transferred using the concept of a continuous data stream flowing from a source to a destination (program flow is illustrated in Figure 10). Data is inserted into the stream by a sender process usually called the server and is extracted from the stream by the receiver process usually called the client. The server process establishes a connection socket and then listens for connection requests from other processes. Connection requests are accepted one at a time. When the connection request is accepted, a data socket is created using which the server process can write or read from/to the data stream. When the communication session between the two processes is over, the data socket is closed and the server process is free to accept another connection request. Note that the server process is blocked when it is listening or waiting for incoming connection requests. This problem could be alleviated by spawning threads, one for each incoming client connection request and a thread would then individually handle the particular client.

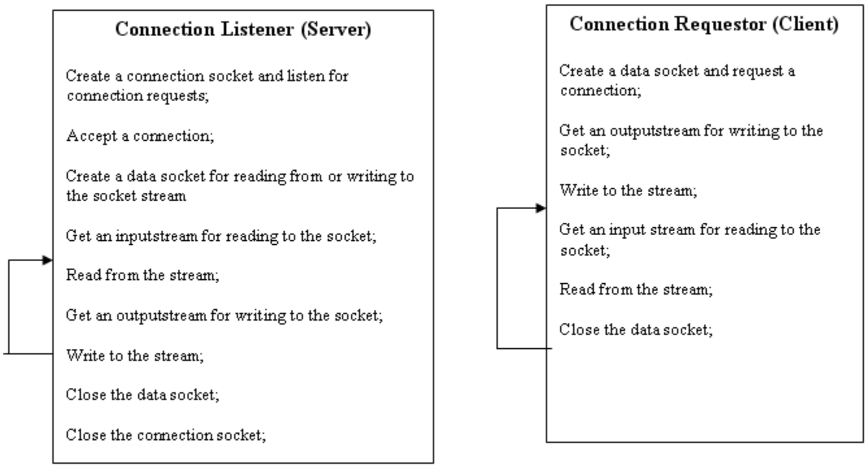

Connection Listener (Server)
Create a connection socket and listen for connection requests;
Accept a connection;
Create a data socket for reading from or writing to the socket stream
Get an inputstream for reading to the socket;
Read from the stream;
Get an outputstream for writing to the socket;
Write to the stream;
Close the data socket;
Close the connection socket;
Connection Requestor (Client)
Create a data socket and request a connection;
Get an outputstream for writing to the socket;
Write to the stream;
Get an input stream for reading to the socket;
Read from the stream;
Close the data socket;

Figure 10: Program Flow in the Connection Listener (Server) and Connection Requester (Client) Processes – adapted from [2]

In the next few sections, we illustrate examples to illustrate the use of Stream-mode sockets to send and receive messages. These sockets are typically used for connection-oriented communication during which a sequence of bytes (not discrete messages) need to be transferred in one or both directions. The connection listener program (Server program) should be started first and ServerSocket object should be the first to be created to accept an incoming client connection request. The connection requester program (Client program) should be then started.

Table 2: Key Methods of the Stream-Mode Socket API (adapted from [3])

No.	Constructor/ Method	Description
		ServerSocket class
1	ServerSocket(int port)	Constructs an object of class ServerSocket and binds the object to the specified port – to which all clients attempt to connect
2	accept( )	This is a blocking method call – the server listens (waits) for any incoming client connection request and cannot proceed further unless contacted by a client. When a client contacts, the method is unblocked and returns a Socket object to the server program to communicate with the client.
3	close( )	Closes the ServerSocket object
4	void setSoTimeout(int timeout)	The ServerSocket object is set to listen for an incoming client request, under a particular invocation of the accept ( ) method on the object, for at most the milliseconds specified in "timeout". When the timeout expires, a java.net.SocketTimeoutException is raised. The timeout value must be > 0; a timeout value of 0 indicates infinite timeout.
		Socket class
5	Socket(InetAddress host, int port)	Creates a stream socket and connects it to the specified port number at the specified IP address
6	InetAddress getInetAddress( )	Returns the IP address at the remote side of the socket
7	InetAddress getLocalAddress( )	Returns the IP address of the local machine bound by the socket
8	int getPort( )	Returns the remote port number to which this socket is connected
9	int getLocalPort( )	Returns the local port number to which this socket is bound
10	InputStream getInputStream( )	Returns an input stream for this socket to read data sent from the other end of the connection
11	OutputStream getOutputStream( )	Returns an output stream for this socket to send data to the other end of the connection
12	close( )	Closes this socket
13	void setSoTimeout(int timeout)	Sets a timeout value to block on any read( ) call on the InputStream associated with this socket object. When the timeout expires, a java.net.SocketTimeoutException is raised. The timeout value must be > 0; a timeout value of 0 indicates infinite timeout.

4.1 *Example Program to Send a Message from Server to Client when Contacted*

Here, the connectionServer.java program creates a ServerSocket object bound to port 3456 and waits for an incoming client connection request. When a client contacts the server program, the accept( ) method is unblocked and returns a Socket object for the server to communicate with the particular client that contacted. The server program then creates a PrintStream object through the output stream extracted from this socket and uses it to send a welcome message to the contacting client. The client program runs as follows: The client creates a Socket object to connect to the server running at the specified IP address or hostname and at the port number 3456. The client creates a BufferedReader object through the input stream extracted from this socket and waits for an incoming line of message from the other end. The readLine( ) method of the BufferedReader object blocks the client from proceeding further unless a line of message is received. The purpose of the flush( ) method of the PrintStream class is to write any buffered output bytes to the underlying output stream and then flush that stream to send out the bytes. Note that the server program in our example sends a welcome message to an incoming client request and then stops.

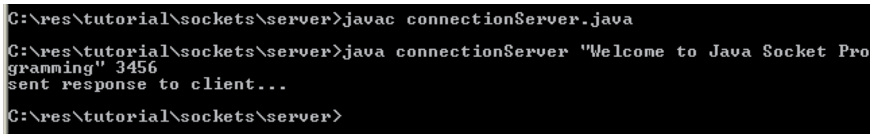
```
-------------------------------------------------------------------------------
import java.net.*;
import java.io.*;

class connectionServer{
   public static void main(String[ ] args){
   try{
      String message = args[0];
      int serverPortNumber = Integer.parseInt(args[1]);
      ServerSocket connectionSocket = new ServerSocket(serverPortNumber);
      Socket dataSocket = connectionSocket.accept();
      PrintStream socketOutput = new PrintStream(dataSocket.getOutputStream());
      socketOutput.println(message);
      System.out.println("sent response to client…");
      socketOutput.flush( );
      dataSocket.close( );
      connectionSocket.close( );
     }
    catch(Exception e){
             e.printStackTrace( );
     }
   }
 }
```

Figure 11: Connection Server Program to Respond to a Connecting Client

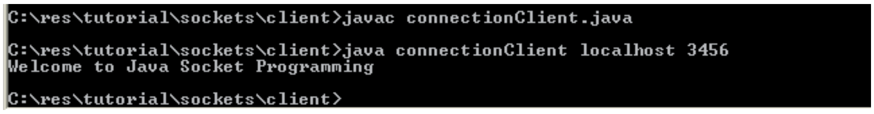
----

```
import java.net.*;
import java.io.*;

class connectionClient{
  public static void main(String[ ] args){
    try{
      InetAddress acceptorHost = InetAddress.getByName(args[0]);
      int serverPortNum = Integer.parseInt(args[1]);
      Socket clientSocket = new Socket(acceptorHost, serverPortNum);
      BufferedReader br = new BufferedReader(new
                                  InputStreamReader(clientSocket.getInputStream( )));
      System.out.println(br.readLine( ));
      clientSocket.close();
     }
    catch(Exception e){
        e.printStackTrace( );
       }
   }
 }
```

Figure 12: Connection Client Program to Connect to a Server Program



```
C:\res\tutorial\sockets\server>javac connectionServer.java

C:\res\tutorial\sockets\server>java connectionServer "Welcome to Java Socket Pro
gramming" 3456
sent response to client...

C:\res\tutorial\sockets\server>
```



```
C:\res\tutorial\sockets\client>javac connectionClient.java

C:\res\tutorial\sockets\client>java connectionClient localhost 3456
Welcome to Java Socket Programming

C:\res\tutorial\sockets\client>
```

Figure 13: Screenshots of the Execution of the Programs to Send a Message to a Client from a Server when Contacted

```
import java.net.*;
import java.io.*;

class connectionClient{
  public static void main(String[ ] args){
    try{
      InetAddress acceptorHost = InetAddress.getByName(args[0]);
      int serverPortNum = Integer.parseInt(args[1]);
      Socket clientSocket = new Socket(acceptorHost, serverPortNum);
      BufferedReader br = new BufferedReader(new
                                  InputStreamReader(clientSocket.getInputStream( )));
      System.out.println(br.readLine( ));
      PrintStream ps = new PrintStream(clientSocket.getOutputStream( ));
      ps.println("received your message.. Thanks");
      ps.flush( );
      clientSocket.close( );
     }
    catch(Exception e){e.printStackTrace( );}
   }
  }
```

Figure 14: Code for the Client Program for Duplex Connection Communication

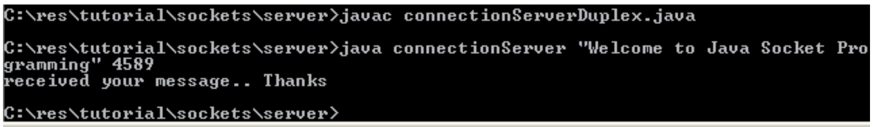
4.2 *Example Program to Illustrate Duplex Nature of Stream-mode Socket Connections*

This program is an extension of the program illustrated in Section 4.1. Here, we illustrate that the communication using stream-mode sockets could occur in both directions. When the client receives a response for its connection request from the server, the client responds back with an acknowledgement that the server response was received. The server waits to receive such a response from the client. All of these communication occur using the Socket object returned by the accept( ) method call on the ServerSocket object at the server side and using the Socket object originally created by the client program to connect to the server. It is always recommended to close the Socket objects at both sides after their use. Even though server programs typically run without being stopped, our example server program terminates after one duplex communication with the client. Before the server program terminates, the ServerSocket object should be closed.

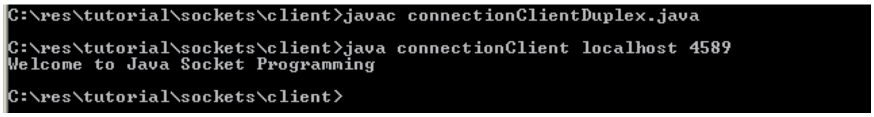
```
import java.net.*;
import java.io.*;

class connectionServer{
 public static void main(String[ ] args){
  try{
    String message = args[0];
    int serverPortNum = Integer.parseInt(args[1]);
    ServerSocket connectionSocket = new ServerSocket(serverPortNum);
    Socket dataSocket = connectionSocket.accept( );
    PrintStream socketOutput = new PrintStream(dataSocket.getOutputStream( ));
    socketOutput.println(message);
    socketOutput.flush( );
    BufferedReader br = new BufferedReader(new
                                    InputStreamReader(dataSocket.getInputStream( )));
    System.out.println(br.readLine( ));
```

```
      dataSocket.close();
      connectionSocket.close();
     }
    catch(Exception e){e.printStackTrace();}
   }
 }
```

Figure 15: Code for the Server Program for Duplex Connection Communication



```
C:\res\tutorial\sockets\server>javac connectionServerDuplex.java

C:\res\tutorial\sockets\server>java connectionServer "Welcome to Java Socket Pro
gramming" 4589
received your message.. Thanks

C:\res\tutorial\sockets\server>
```



```
C:\res\tutorial\sockets\client>javac connectionClientDuplex.java

C:\res\tutorial\sockets\client>java connectionClient localhost 4589
Welcome to Java Socket Programming

C:\res\tutorial\sockets\client>
```

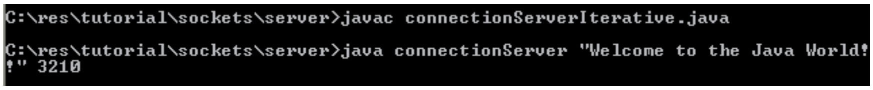
Figure 16: Screenshots of the Execution of the Programs for Duplex Communication

4.3 *Example Program to Illustrate the Server can Run in Infinite Loop handling Multiple Client Requests, one at a time*

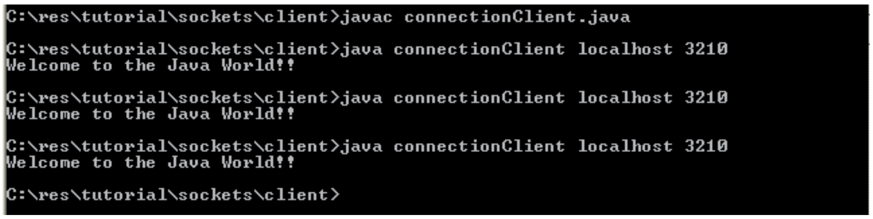
In this example, we illustrate a server program (an iterative server) that can service multiple clients, though, one at a time. The server waits for incoming client requests. When a connection request is received, the accept ( ) method returns a Socket object that will be used to handle all the communication with the client. During this time, if any connection requests from any other client reach the server, these requests have to wait before the server has completed its communication with the current client. To stop a server program that runs in an infinite loop, we press Ctrl+C. This terminates the program as well as closes the ServerSocket object.

```
------------------------------------------------------------------------------------------------------------
import java.net.*;
import java.io.*;

class connectionServer{
  public static void main(String[ ] args){
    try{
       String message = args[0];
       int serverPortNum = Integer.parseInt(args[1]);
       ServerSocket connectionSocket = new ServerSocket(serverPortNum);

       while (true){
           Socket dataSocket = connectionSocket.accept( );
           PrintStream socketOutput = new PrintStream(dataSocket.getOutputStream( ));
           socketOutput.println(message);
           socketOutput.flush( );
           dataSocket.close( );
         }
     }
     catch(Exception e){e.printStackTrace( );}
   }
 }
------------------------------------------------------------------------------------------------------------
```

Figure 17: Iterative Server Program to Send a Message to each of its Clients



```
C:\res\tutorial\sockets\server>javac connectionServerIterative.java

C:\res\tutorial\sockets\server>java connectionServer "Welcome to the Java World!
!" 3210
```



```
C:\res\tutorial\sockets\client>javac connectionClient.java

C:\res\tutorial\sockets\client>java connectionClient localhost 3210
Welcome to the Java World!!

C:\res\tutorial\sockets\client>java connectionClient localhost 3210
Welcome to the Java World!!

C:\res\tutorial\sockets\client>java connectionClient localhost 3210
Welcome to the Java World!!

C:\res\tutorial\sockets\client>
```

Figure 18: Screenshots of the Execution of an Iterative Server along with its Multiple Clients

Note that the client code need not be modified to communicate with the iterative server. The same client code that was used in Section 4.1 or 4.2 could be used here, depending on the case. In this example (illustrated in Figure 17), the iterative server just responds back with a welcome message and does not wait for the client response. Hence, one would have to use a client program, like the one shown in Section 4.1, to communicate with this iterative server program.

4.4 *Example Program to Send Objects of User-defined Classes using Stream-mode Sockets*

In this example, we illustrate how objects of user-defined classes could be sent using stream-mode sockets. An important requirement of classes whose objects needs to be transmitted across sockets is that these classes should implement the *Serializable* interface defined in the java.io. package. Figure 19 shows the code for an Employee class (that has three member variables – ID, Name and Salary), implementing the Serializable interface. An object of the Employee class, with all the member variables set by obtaining inputs from the user, is being sent by the client program (code in Figure 20) to a server program (code in Figure 21) that extracts the object from the socket and prints the values of its member variables. To write an object to a socket, we use the ObjectOutputStream and to extract an object from a socket, we use the ObjectInputStream.

```
---------------------------------------------------------------------------------------------------------
import java.io.*;

class EmployeeData implements Serializable{
  int empID;
  String empName;
  double empSalary;
  void setID(int id){     empID = id;   }
  void setName(String name){      empName = name;     }
  void setSalary(double salary){       empSalary = salary;     }
  int getID( ){      return empID;     }
  String getName( ){      return empName;     }
  double getSalary( ){      return empSalary;     }
}
```

Figure 19: Code for the Employee Class, Implementing the *Serializable* Interface

----

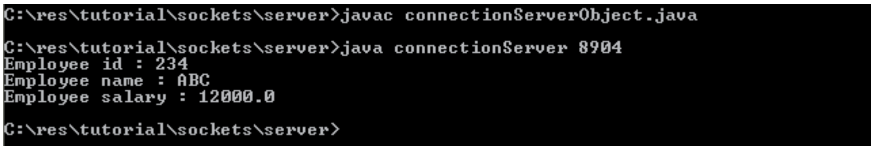
```
import java.io.*;
import java.net.*;
import java.util.*;

class connectionClient{
  public static void main(String[] args){
    try{
      InetAddress serverHost = InetAddress.getByName(args[0]);
      int serverPortNum = Integer.parseInt(args[1]);
      Socket clientSocket = new Socket(serverHost, serverPortNum);
      EmployeeData empData = new EmployeeData( );
      Scanner input = new Scanner(System.in);
      System.out.print("Enter employee id: ");
      int id = input.nextInt( );
      System.out.print("Enter employee name: ");
      String name = input.next( );
      System.out.print("Enter employee salary: ");
      double salary = input.nextDouble( );
      empData.setID(id);
      empData.setName(name);
      empData.setSalary(salary);
     ObjectOutputStream oos = new ObjectOutputStream(clientSocket.getOutputStream( ));
     oos.writeObject(empData);
     oos.flush( );
     clientSocket.close( );
    }
   catch(Exception e){e.printStackTrace( );}
  }
}
```

Figure 20: Client Program to Send an Object of User-defined Class across Stream-mode Socket

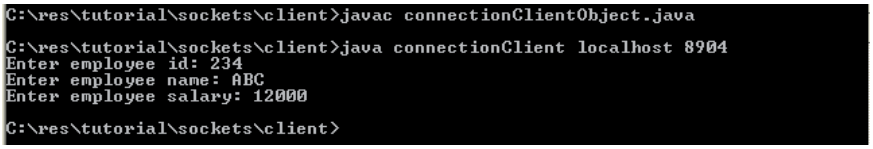
----

```
import java.io.*;
import java.net.*;

class connectionServer{
   public static void main(String[] args){
     try{
       int serverListenPortNum = Integer.parseInt(args[0]);
       ServerSocket connectionSocket = new ServerSocket(serverListenPortNum);
       Socket dataSocket = connectionSocket.accept( );
       ObjectInputStream ois = new ObjectInputStream(dataSocket.getInputStream( ));
       EmployeeData eData = (EmployeeData) ois.readObject( );
       System.out.println("Employee id : "+eData.getID( ));
       System.out.println("Employee name : "+eData.getName( ));
       System.out.println("Employee salary : "+eData.getSalary( ));
       dataSocket.close( );
       connectionSocket.close( );
      }
    catch(Exception e){e.printStackTrace( );}
   }
}
```

Figure 21: Server Program to Receive an Object of User-defined Class across a Stream-mode Socket



```
C:\res\tutorial\sockets\server>javac connectionServerObject.java

C:\res\tutorial\sockets\server>java connectionServer 8904
Employee id : 234
Employee name : ABC
Employee salary : 12000.0

C:\res\tutorial\sockets\server>
```



```
C:\res\tutorial\sockets\client>javac connectionClientObject.java

C:\res\tutorial\sockets\client>java connectionClient localhost 8904
Enter employee id: 234
Enter employee name: ABC
Enter employee salary: 12000

C:\res\tutorial\sockets\client>
```

Figure 22: Screenshots of the Client-Server Program to Send and Receive Object of User-defined Class across Stream-mode Sockets

4.5 *Example Program to Illustrate Sending and Receiving of Integers across a Stream-mode Socket*

In this example program, we illustrate the sending and receiving of integers across a stream-mode socket. The client program sends two integers using the PrintStream object; the server program receives them, computes and prints their sum.

```
--------------------------------------------------------------------------------
import java.io.*;
import java.net.*;
import java.util.*;

class connectionClient{
 public static void main(String[ ] args){
  try{
   InetAddress serverHost = InetAddress.getByName(args[0]);
   int serverPortNum = Integer.parseInt(args[1]);
   Socket clientSocket = new Socket(serverHost, serverPortNum);
   PrintStream ps = new PrintStream(clientSocket.getOutputStream());
   ps.println(2);
   ps.flush( );
   ps.println(3);
   ps.flush( );
   clientSocket.close( );
   }
catch(Exception e){e.printStackTrace( );}
  }
}
```

Figure 23: Client Program to Send Two Integers across a Stream-mode Socket

---
```
import java.io.*;
import java.net.*;

class connectionServer{
  public static void main(String[] args){
   try{
      int serverListenPortNum = Integer.parseInt(args[0]);
      ServerSocket connectionSocket = new ServerSocket(serverListenPortNum);
      Socket dataSocket = connectionSocket.accept( );
      BufferedReader br = new BufferedReader(new
                                         InputStreamReader(dataSocket.getInputStream( )));
      int num1 = Integer.parseInt(br.readLine( ));
      int num2 = Integer.parseInt(br.readLine( ));
      System.out.println(num1+" + "+num2+" = "+(num1+num2));
      dataSocket.close( );
      connectionSocket.close( );
     }
   catch(Exception e){e.printStackTrace( );}
  }
}
```

Figure 23: Server to Receive Integers across a Stream-mode Socket

4.6 *Iterative Server vs. Concurrent Server*

4.6.1 Iterative Server

As illustrated in Section 4.3, an iterative server is a server program that handles one client at a time. If one or more client connection requests reach the server while the latter is in communication with a client, these requests have to wait for the existing communication to be completed. The pending client connection requests are handled on a First-in-First-Serve basis. However, such a design is not efficient. Clients may have to sometime wait for excessive amount of time for the requests ahead of theirs in the waiting queue to be processed. When the client requests differ in the amount of time they take to be handled by the server, it would then lead to a situation where a client with a lower execution time for its request at the server may have to wait for the requests (ahead in the queue) that have a relatively longer execution time to be completed first. The code in Figure 25 illustrates one such example of an iterative server that has to add integers from 1 to a "count" value (i.e., 1+2+…+count) sent by a client program (Figure 24) and return the sum of these integers to the requesting client. In order to simulate the effect of time-consuming client requests, we make the server program to sleep for 200 milliseconds after performing each addition. As iterative servers are single-threaded programs, the whole program sleeps when we invoke the sleep( ) static function of the Thread class. The execution screenshots illustrated in Figure 26 show that a client with a request to add integers from 1 to 5 will have to wait for 19500 milliseconds (i.e., 19.5 seconds) as the client's request reached the server while the latter was processing a request from another client to add integers from 1 to 100, which takes 20031 milliseconds (i.e., 20.031 seconds).

```
import java.io.*;
import java.net.*;

class summationClient{
  public static void main(String[ ] args){
    try{
      InetAddress serverHost = InetAddress.getByName(args[0]);
      int serverPort = Integer.parseInt(args[1]);
      long startTime = System.currentTimeMillis( );
      int count = Integer.parseInt(args[2]);

      Socket clientSocket = new Socket(serverHost, serverPort);
```

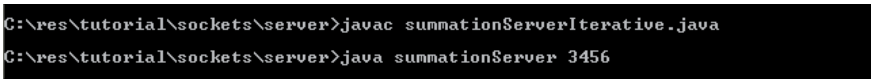
```
        PrintStream ps = new PrintStream(clientSocket.getOutputStream());
        ps.println(count);
        BufferedReader br = new BufferedReader(new
                InputStreamReader(clientSocket.getInputStream()));

        int sum = Integer.parseInt(br.readLine());
        System.out.println(" sum = "+sum);

        long endTime = System.currentTimeMillis();

        System.out.println(" Time consumed for receiving the feedback from the server:
                                        "+(endTime-startTime)+" milliseconds");
        clientSocket.close( );
        }
      catch(Exception e){e.printStackTrace( );}
    }
}
```

Figure 24: Code for a Client Program that Requests a Server to Add Integers (from 1 to a Count value) and Return the Sum

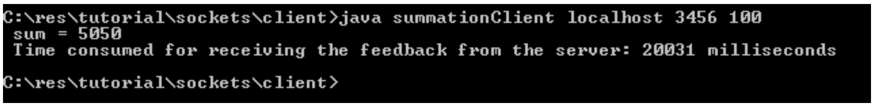
---

```
import java.io.*;
import java.net.*;

class summationServer{
    public static void main(String[] args){
      try{
        int serverPort = Integer.parseInt(args[0]);
        ServerSocket calcServer = new ServerSocket(serverPort);
        while (true){
        Socket clientSocket = calcServer.accept( );
        BufferedReader br = new BufferedReader(new
                                    InputStreamReader(clientSocket.getInputStream( )));
        int count = Integer.parseInt(br.readLine( ));

        int sum = 0;
        for (int ctr = 1; ctr <= count; ctr++){
          sum += ctr;
          Thread.sleep(200);
        }

        PrintStream ps = new PrintStream(clientSocket.getOutputStream( ));
        ps.println(sum);
        ps.flush( );
        clientSocket.close( );

        }
      }
    catch(Exception e){e.printStackTrace( );}
  }
}
```

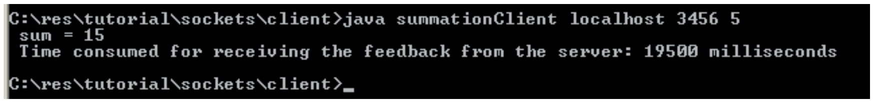
Figure 25: Code for an Iterative Server that Adds (from 1 to a Count value) and Returns the Sum



```
C:\res\tutorial\sockets\server>javac summationServerIterative.java

C:\res\tutorial\sockets\server>java summationServer 3456
```



```
C:\res\tutorial\sockets\client>java summationClient localhost 3456 100
 sum = 5050
 Time consumed for receiving the feedback from the server: 20031 milliseconds

C:\res\tutorial\sockets\client>
```



```
C:\res\tutorial\sockets\client>java summationClient localhost 3456 5
 sum = 15
 Time consumed for receiving the feedback from the server: 19500 milliseconds

C:\res\tutorial\sockets\client>_
```

Figure 26: Screenshots of Execution of an Iterative Summation Server and its Clients

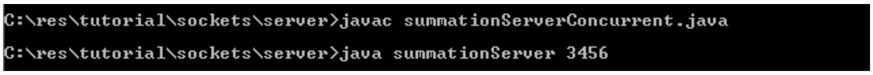
4.6.2 Concurrent Server

An alternative design is the idea of using a concurrent server, especially to process client requests with variable service time. When a client request is received, the server process spawns a separate thread, which is exclusively meant to handle the particular client. So, if a program has to sleep after each addition, it would be the particular thread (doing the addition) that will sleep and not the whole server process, which was the case with an iterative server. While a thread of a process is sleeping, the operating system could schedule the other threads of this process to run. With such a design, the waiting time of client requests, especially for those with a relatively shorter processing time, could be significantly reduced. Note that the code for the client program is independent of the design choice for the server. In other words, one should be able to use the same client program with either an iterative server or a concurrent server.

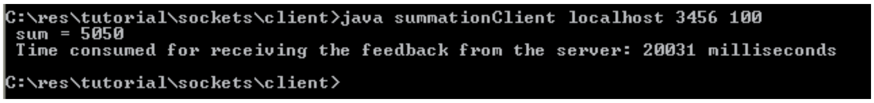
```
-------------------------------------------------------------------------------------------------------
import java.io.*;
import java.net.*;

class summationThread extends Thread{
   Socket clientSocket;
   summationThread(Socket cs){      clientSocket = cs;      }

   public void run( ){
     try{
        BufferedReader br = new BufferedReader(new
                                        InputStreamReader(clientSocket.getInputStream( )));
        int count = Integer.parseInt(br.readLine( ));
        int sum = 0;
        for (int ctr = 1; ctr <= count; ctr++){
          sum += ctr;
          Thread.sleep(200);
        }

        PrintStream ps = new PrintStream(clientSocket.getOutputStream( ));
        ps.println(sum);
        ps.flush( );
        clientSocket.close( );
```

```
       }
      catch(Exception e){e.printStackTrace( );}
     }
}

class summationServer{
     public static void main(String[ ] args){
       try{
          int serverPort = Integer.parseInt(args[0]);
          ServerSocket calcServer = new ServerSocket(serverPort);
          while (true){
              Socket clientSocket = calcServer.accept( );
              summationThread thread = new summationThread(clientSocket);
              thread.start( );
           }
         }
      catch(Exception e){e.printStackTrace( );}
   }
}
```

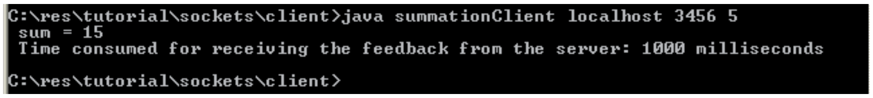
Figure 27: Concurrent Server Program and the Implementation of a Summation Thread



```
C:\res\tutorial\sockets\server>javac summationServerConcurrent.java

C:\res\tutorial\sockets\server>java summationServer 3456
```



```
C:\res\tutorial\sockets\client>java summationClient localhost 3456 100
 sum = 5050
 Time consumed for receiving the feedback from the server: 20031 milliseconds

C:\res\tutorial\sockets\client>
```



```
C:\res\tutorial\sockets\client>java summationClient localhost 3456 5
 sum = 15
 Time consumed for receiving the feedback from the server: 1000 milliseconds

C:\res\tutorial\sockets\client>
```

Figure 28: Screenshots of Execution of a Concurrent Summation Server and its Clients

Figure 27 presents the code for a concurrent server. We implement the addition module inside the run( ) function of the SummationThread class, an object of which is spawned (i.e., a server thread) for each incoming client connection requests. From then on, the server thread handles the communication with the particular client through the Socket object returned by the accept( ) method of the ServerSocket class and passed as an argument to the constructor of the SummationThread class. The effectiveness of using a concurrent server (illustrated in Figure 28) can be validated through the reduction in the processing time for a client whose request to add integers from 1 to 5 sent to the server after the latter started to process a client request to add integers from 1 to 100. The result at the client that requests to add integers 1 to 5 (it takes only 1000 milliseconds – i.e., 1 second) would actually appear well ahead of that at the other client (it takes 20031 milliseconds – i.e. 20.031 seconds).

5 Multicast Sockets

Multicasting is the process of sending messages from one process to several other processes concurrently. It supports one-to-many Inter Process Communication (IPC). Multicasting is useful for applications like groupware, online conferences and interactive learning, etc. In applications or network services that make use of multicasting, we have a set of processes that form a group called multicast group. Each process in the group can send and receive messages. A message sent by any process in the group will be received by each participating process in the group.

A Multicast API should support the following primitive operations:

(1) Join – allows a process to join a specific multicast group. A process may be a member of one or more multicast groups at the same time.

(2) Leave – allows the process to stop participating in a multicast group.

(3) Send – allows a process to send a message to all the processes of a multicast group.
(4) Receive – allows a process to receive messages sent to a multicast group.

At the transport layer, the basic multicast supported by Java is an extension of UDP, which is connectionless and unreliable. There are four major classes in the API: (1) InetAddress (2) DatagramPacket (3) DatagramSocket and (4) MulticastSocket. The MulticastSocket class is an extension of the DatagramSocket class and provides capabilities for joining and leaving a multicast group. The constructor of the MulticastSocket class takes an integer argument that corresponds to the port number to which the object of this class would be bound to. The receive( ) method of the MulticastSocket class is a blocking-method, which when invoked on a MulticastSocket object will block the execution of the receiver until a message arrives to the port to which the object is bound to.

An IP multicast datagram must be received by all the processes that are currently members of a particular multicast group. Hence, each multicast datagram needs to be addressed to a specific multicast group instead of an individual process. In IPv4, a multicast group is specified by a class D IP address combined with a standard port number. In this chapter, we will use the static address 224.0.0.1, with an equivalent domain name ALL-SYSTTEMS.MCAST.NET, for processes running on all machines on the local area network.

5.1 *Example Program to Illustrate an Application in which a Message Sent by a Process Reaches all the Processes Constituting the Multicast Group*

In this example, we illustrate an application wherein there are two types of processes: (i) multicast sender – that can only send a message to a multicast group and (ii) multicast receiver – that can only receive a message sent to the multicast group. Similar to the case of connection-oriented and connectionless sockets, the multicast receiver (Figure 30) should be started first and should be ready to receive messages sent to the port number to which the multicast socket is bound to. We then start the multicast sender (Figure 29).

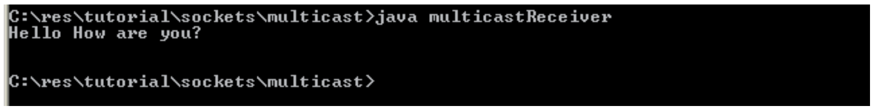
To keep it simple, the multicast sender program stops after sending one message to the multicast group and the multicast receiver program stops after receiving one message for the group. The maximum size of the message that can be received in this example is 100 bytes.

```
import java.io.*;
import java.net.*;

class multicastSender{
  public static void main(String[ ] args){
   try{
     InetAddress group = InetAddress.getByName("224.0.0.1");
     MulticastSocket multicastSock = new MulticastSocket(3456);
     String msg = "Hello How are you?";
     DatagramPacket packet = new DatagramPacket(msg.getBytes( ), msg.length( ), group,
                                                                                3456);
     multicastSock.send(packet);
     multicastSock.close( );
    }
   catch(Exception e){e.printStackTrace( );}
  }
}
```

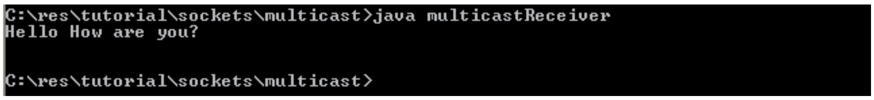
Figure 29: Code for Multicast Sender Program

```
import java.io.*;
import java.net.*;

class multicastReceiver{
  public static void main(String[ ] args){
    try{
      InetAddress group = InetAddress.getByName("224.0.0.1");
      MulticastSocket multicastSock = new MulticastSocket(3456);
      multicastSock.joinGroup(group);
      byte[ ] buffer = new byte[100];
      DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
      multicastSock.receive(packet);
      System.out.println(new String(buffer));
      multicastSock.close( );
     }
   catch(Exception e){e.printStackTrace( );}
  }
}
```

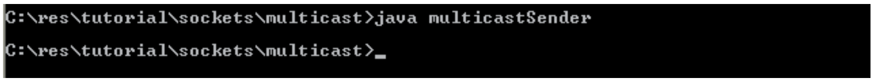
Figure 30: Code for Multicast Receiver Program



```
C:\res\tutorial\sockets\multicast>java multicastReceiver
Hello How are you?

C:\res\tutorial\sockets\multicast>
```



```
C:\res\tutorial\sockets\multicast>java multicastReceiver
Hello How are you?

C:\res\tutorial\sockets\multicast>
```



```
C:\res\tutorial\sockets\multicast>java multicastSender

C:\res\tutorial\sockets\multicast>_
```

Figure 31: Screenshots of Execution of a Multicast Sender and Receiver Program

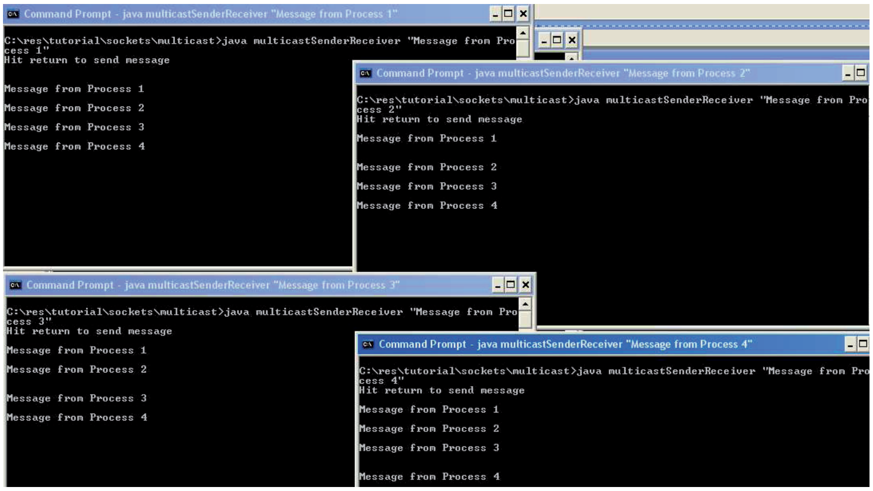
5.2 *Example Program to Illustrate an Application in which each Process of the Multicast Group Sends a Message that is Received by all the Processes Constituting the Group*

In this example, each process should be both a multicast sender as well as a receiver such that the process can send only one message (to the multicast group); but, should be able to receive several messages. Since a process can send only one message, the number of messages received by a process would equal the number of processes that are part of the multicast group. Since a process should have both the sending and receiving functionality built-in to its code, we implement the relatively simpler sending module in the main( ) function; whereas, the receiving functionality is implemented as a thread (readThread class in Figure 32). The readThread object is spawned and starts to run before the sending module begins its execution. In order to facilitate this, the code in Figure 32 will require all the processes to be started first. After all the processes have begun to run (i.e., the readThread has been spawned), we then press the Enter-key in each process command window. This will trigger the sending of a message by each process. The readThread displays the received message (refer to Figure 33).

```
------------------------------------------------------------------------------------------------------------
import java.net.*;
import java.io.*;

class readThread extends Thread{
   InetAddress group;
   int multicastPort;
   int MAX_MSG_LEN = 100;

   readThread(InetAddress g, int port){
     group = g;
     multicastPort = port;
   }

   public void run( ){

    try{

      MulticastSocket readSocket = new MulticastSocket(multicastPort);
      readSocket.joinGroup(group);

      while (true){
       byte[ ] message = new byte[MAX_MSG_LEN];
```

```
        DatagramPacket packet = new DatagramPacket(message, message.length, group,
                                                                          multicastPort);
        readSocket.receive(packet);
        String msg = new String(packet.getData());
        System.out.println(msg);
        }
      }
     catch(Exception e){e.printStackTrace( );}
    }
}

class multicastSenderReceiver{
  public static void main(String[] args){

   try{

    int multicastPort = 3456;
    InetAddress group = InetAddress.getByName("224.0.0.1");
    MulticastSocket socket = new MulticastSocket(multicastPort);
    readThread rt = new readThread(group, multicastPort);
    rt.start( );

    String message = args[0];
    byte[ ] msg = message.getBytes( );
    DatagramPacket packet = new DatagramPacket(msg, msg.length, group,
                                                                          multicastPort);
    System.out.print("Hit return to send message\n\n");
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    br.readLine( );
    socket.send(packet);
    socket.close( );
   }
   catch(Exception e){e.printStackTrace( );}
  }
}
```

Figure 32: Code for the Multicast Sender and Receiver Program that can both Send and Receive



```
C:\res\tutorial\sockets\multicast>java multicastSenderReceiver "Message from Pro
cess 1"
Hit return to send message

Message from Process 1
Message from Process 2
Message from Process 3
Message from Process 4
```

```
C:\res\tutorial\sockets\multicast>java multicastSenderReceiver "Message from Pro
cess 2"
Hit return to send message
Message from Process 1

Message from Process 2
Message from Process 3
Message from Process 4
```

```
C:\res\tutorial\sockets\multicast>java multicastSenderReceiver "Message from Pro
cess 3"
Hit return to send message
Message from Process 1
Message from Process 2

Message from Process 3
Message from Process 4
```

```
C:\res\tutorial\sockets\multicast>java multicastSenderReceiver "Message from Pro
cess 4"
Hit return to send message
Message from Process 1
Message from Process 2
Message from Process 3

Message from Process 4
```

Command Prompt - java multicastSenderReceiver "Message from Process 1"
Command Prompt - java multicastSenderReceiver "Message from Process 2"
Command Prompt - java multicastSenderReceiver "Message from Process 3"
Command Prompt - java multicastSenderReceiver "Message from Process 4"

Figure 33: Execution of Multicast Sender and Receiver Program that can both Send and Receive

6 Exercises

1. Implement a simple file transfer protocol (FTP) using connection-oriented and connectionless sockets. The connection-oriented FTP works as follows: At the client side, the file to be transferred is divided into units of 100 bytes (and may be less than 100 bytes for the last unit depending on the size of the file). The client transfers each unit of the file to the server and expects an acknowledgment from the server. Only after receiving an acknowledgment from the server, the client transmits the next unit of the file. If the acknowledgment is not received within a timeout period (choose your own value depending on your network delay), the client retransmits the unit. The above process is repeated until all the contents of the file are transferred. The connectionless FTP works simply as follows: The file is broken down into lines

and the client sends one line at a time as a datagram packet to the server. There is no acknowledgment required from the server side.

2. Develop a concurrent file server that spawns several threads, one for each client requesting a specific file. The client program sends the name of the file to be downloaded to the server. The server creates the thread by passing the name of the file as the argument for the thread constructor. From then on, the server thread is responsible for transferring the contents of the requested file. Use connection-oriented sockets (let the transfer size be at most 1000 bytes per flush operation). After a flush operation, the server thread sleeps for 200 milliseconds.
3. Develop a "Remote Calculator" application that works as follows: The client program inputs two integers and an arithmetic operation ('*','/','%','+','-') from the user and sends these three values to the server side. The server does the binary operation on the two integers and sends backs the result of the operation to the client. The client displays the result to the user.
4. Develop a streaming client and server application using connectionless sockets that works as follows: The streaming client contacts the streaming server requesting a multi-media file (could be an audio or video file) to be sent. The server then reads the contents of the requested multi-media file in size randomly distributed between 1000 and 2000 bytes and sends the contents read to the client as a datagram packet. The last datagram packet that will be transmitted could be of size less than 1000 bytes, if required. The client reads the bytes, datagram packets, sent from the server. As soon as a reasonable number of bytes are received at the client side, the user working at the client side should be able to launch a media player and view/hear the portions of the received multi-media file while the downloading is in progress.

5. Develop a simple chatting application using (i) Connection-oriented and (ii) Connectionless sockets. In each case, when the user presses the "Enter" key, whatever characters have been typed by the user until then are transferred to the other end. You can also assume that for every message entered from one end, a reply must come from the other end, before another message could be sent. In other words, more than one message cannot be sent from a side before receiving a response from the other side. For connectionless communication, assume the maximum number of characters that can be transferred in a message to be 1000. The chat will be stopped by pressing Ctrl+C on both sides.
6. Extend the single client – single server chatting application developed in Q5 using connection-oriented sockets to a multiple client – single server chatting application. The single server should be able to chat simultaneously with multiple clients. In order to do this, you will have to implement the server program using threads. Once a client program contacts a server, the server process spawns a thread that will handle the client. The communication between a client and its server thread will be like a single client-single server chatting application.
7. Develop a multicast chatting tool that will be used to communicate among a group of processes. Each process should be able to send and receive any number of messages. The chat tool should have the following functionalities:
    1) Get the message from the user and send it to all the other processes belonging to the group. A process can receive a copy of the message.
    2) Read the messages sent by any other process and display the message to the user.
8. Develop an election vote casting application as follows: There are two candidates A and B contesting an election. There are five electorates (processes)

and each electorate can cast their vote only once and for only one of the two candidates (A or B). The vote cast by an electorate is a character ‘A’ or ‘B’, sent as a multicast message to all the other electorates. The winner is the candidate who gets the maximum number of votes. After casting the vote and also receiving the vote messages from all other electorates, each electorate should be able to independently determine the winner and display it.

References

[1] D. E. Comer, “Computer Networks and Internets,” 5th Edition, Prentice Hall, 2008.

[2] M. L. Liu, “Distributed Computing: Principles and Applications,” Addison Wesley, 2004.

[3] Java API: http://download.oracle.com/javase/1.4.2/docs/api/

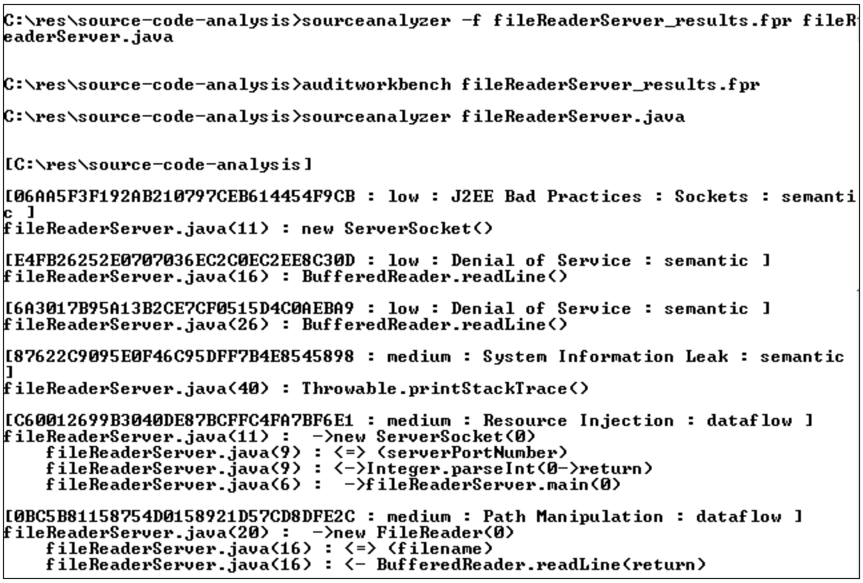
MODULE II

A TUTORIAL ON SOURCE CODE ANALYSIS OF JAVA PROGRAMS

1 Introduction

With the phenomenal growth of the Internet, it is imperative to test for the security of software during its developmental lifecycle and fix the vulnerabilities, if any is found, before deployment. Until recently, security has been often considered as an afterthought, and the bugs are mostly detected post-deployment through user experiences and attacks reported. The bugs are often controlled through patch code (more formally called 'security updates') that is quite often sent to customers via Internet. Sometimes, patch codes developed to fix one bug may often open several new vulnerabilities, which if left unattended, can pose a significant risk for the system (and its associated resources) on which the software is run. It is critical that software be built-in with security features (starting from the requirement analysis stage itself, and implemented with appropriate modules as well as tested with suitable analysis techniques) during its entire development lifecycle.

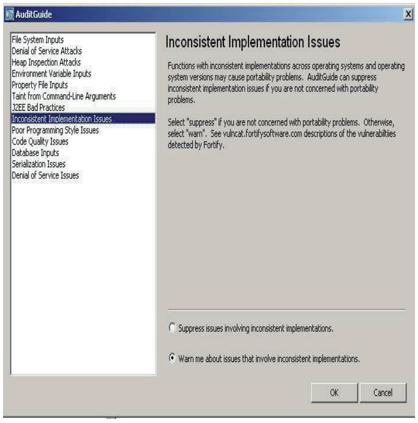
In this chapter, we focus on testing for software security using source code analysis (also invariably referred to as *static code analysis*). Static code analysis refers to examining a piece of code without actually executing it [1]. The technique of evaluating software during its execution is referred to as run-time code analysis (also called dynamic code analysis) [2] – the other commonly used approach to test for software security. While dynamic code analysis is mainly used to test for logical errors and stress test the software by running it in an environment with limited resources, static or source code analysis has been the principal means to evaluate the software with respect to functional, semantic and structural issues including, but not limited to, type checking, style checking, program verification, property checking and bug finding [3]. On the top of these issues, the use of static code analysis to analyze the security aspects of software is gaining prominence. Static code analysis helps to identify the potential threats (vulnerabilities) associated with the software, analyze the complexity involved (in terms of increase in code size, development time, and code run time, etc) and the impact on user experiences in fixing these vulnerabilities through appropriate security controls [4]. Static code analysis also facilitates evaluating the risks involved in only mitigating or just leaving these vulnerabilities unattended – thus, leading to an attack, the consequences of such attacks and the cost of developing security controls and integrating them to the software after the attack has occurred [5]. In addition, static code analysis is also used to analyze the impact of the design and the use of the underlying platform and technologies on the security of the software [6]. For example, programs developed in C/Unix platforms may have buffer overflow vulnerabilities, which are very critical to be identified and mitigated; whereas, buffer overflow vulnerabilities are not an issue for software developed in Java. Software developed for J2EE platforms are strictly forbidden from using a *main* function as the starting point of a program, whereas the *main* function is traditionally considered the starting point of execution of software programs developed in standard J2SE development kits and other high-level programming languages. It would be very time consuming and often ineffective to manually conduct static code analysis on software and analyze the above issues as well as answer pertinent questions related to the security of software. One also needs to have a comprehensive knowledge of possible exploits and their solutions to manually conduct static code analysis.



```
C:\res\source-code-analysis>sourceanalyzer -f fileReaderServer_results.fpr fileR
eaderServer.java

C:\res\source-code-analysis>auditworkbench fileReaderServer_results.fpr

C:\res\source-code-analysis>sourceanalyzer fileReaderServer.java


[C:\res\source-code-analysis]

[06AA5F3F192AB210797CEB614454F9CB : low : J2EE Bad Practices : Sockets : semanti
c ]
fileReaderServer.java(11) : new ServerSocket()

[E4FB26252E0707036EC2C0EC2EE8C30D : low : Denial of Service : semantic ]
fileReaderServer.java(16) : BufferedReader.readLine()

[6A3017B95A13B2CE7CF0515D4C0AEBA9 : low : Denial of Service : semantic ]
fileReaderServer.java(26) : BufferedReader.readLine()

[87622C9095E0F46C95DFF7B4E8545898 : medium : System Information Leak : semantic
]
fileReaderServer.java(40) : Throwable.printStackTrace()

[C60012699B3040DE87BCFFC4FA7BF6E1 : medium : Resource Injection : dataflow ]
fileReaderServer.java(11) :  ->new ServerSocket(0)
    fileReaderServer.java(9) : <=> (serverPortNumber)
    fileReaderServer.java(9) : <->Integer.parseInt(0->return)
    fileReaderServer.java(6) :  ->fileReaderServer.main(0)

[0BC5B81158754D0158921D57CD8DFE2C : medium : Path Manipulation : dataflow ]
fileReaderServer.java(20) :  ->new FileReader(0)
    fileReaderServer.java(16) : <=> (filename)
    fileReaderServer.java(16) : <- BufferedReader.readLine(return)
```

Figure 1: Command-line Execution of the Source Code Analyzer on a Java Program and Forwarding the Results to an Audit Workbench Format File

Various automated tools have been recently developed to conduct static code analysis [7][8]. In this chapter, we illustrate the use of a very effective tool developed by Fortify Inc., called the Source Code Analyzer (SCA) [9]. The Fortify SCA can be used to conduct static code analysis on C/C++ or Java code and can be run in Windows, Linux or Mac platforms. The SCA can analyze individual program files or entire projects collectively. The analyzer uses criteria that are embedded into a generic rulepack (a set of rules) to analyze programs developed in a specific platform/ language. Users may use these generic rulepacks that come with the SCA or develop their own customized sets of rules.



AuditGuide

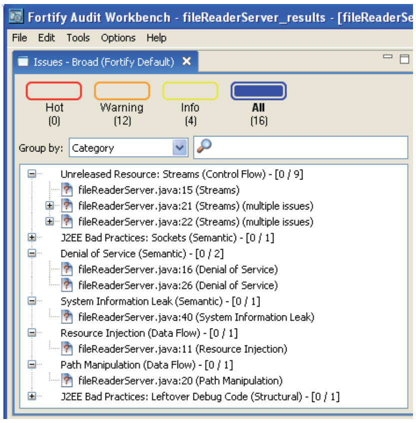
- File System Inputs
- Denial of Service Attacks
- Heap Inspection Attacks
- Environment Variable Inputs
- Property File Inputs
- Taint from Command-Line Arguments
- J2EE Bad Practices
- Inconsistent Implementation Issues
- Poor Programming Style Issues
- Code Quality Issues
- Database Inputs
- Serialization Issues
- Denial of Service Issues

Inconsistent Implementation Issues

Functions with inconsistent implementations across operating systems and operating system versions may cause portability problems. AuditGuide can suppress inconsistent implementation issues if you are not concerned with portability problems.

Select "suppress" if you are not concerned with portability problems. Otherwise, select "warn". See vulncat.fortifysoftware.com descriptions of the vulnerabiltiies detected by Fortify.

Suppress issues involving inconsistent implementations. [✓ Warn me about issues that involve inconsistent implementations.]



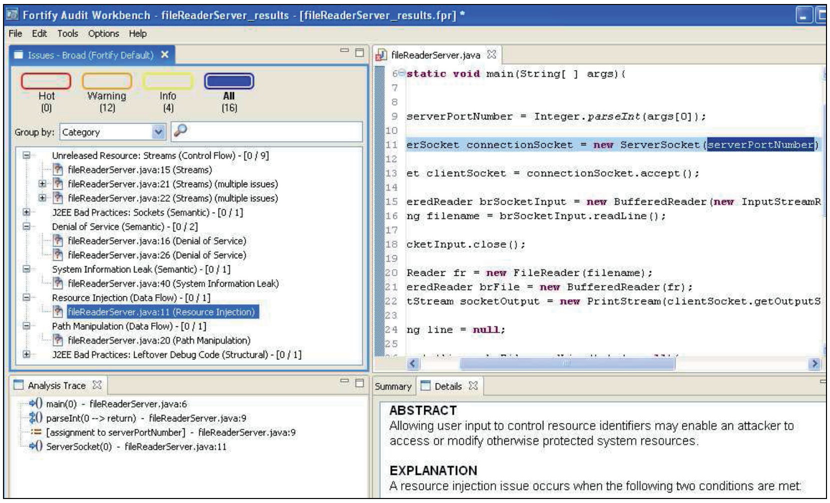
Fortify Audit Workbench - fileReaderServer_results - [fileReaderSe

File Edit Tools Options Help
Issues - Broad (Fortify Default)
Hot (0)
Warning (12)
Info (4)
All (16)

Group by: Category

- Unreleased Resource: Streams (Control Flow) - [0 / 9]
  - fileReaderServer.java:15 (Streams)
  - fileReaderServer.java:21 (Streams) (multiple issues)
  - fileReaderServer.java:22 (Streams) (multiple issues)
- J2EE Bad Practices: Sockets (Semantic) - [0 / 1]
- Denial of Service (Semantic) - [0 / 2]
  - fileReaderServer.java:16 (Denial of Service)
  - fileReaderServer.java:26 (Denial of Service)
- System Information Leak (Semantic) - [0 / 1]
  - fileReaderServer.java:40 (System Information Leak)
- Resource Injection (Data Flow) - [0 / 1]
  - fileReaderServer.java:11 (Resource Injection)
- Path Manipulation (Data Flow) - [0 / 1]
  - fileReaderServer.java:20 (Path Manipulation)
- J2EE Bad Practices: Leftover Debug Code (Structural) - [0 / 1]

Figure 2: Audit Workbench Audit Guide **Figure 3:** List of Issues identified



Fortify Audit Workbench - fileReaderServer_results - [fileReaderServer_results.fpr] *

File Edit Tools Options Help
Issues - Broad (Fortify Default)
Hot (0)
Warning (12)
Info (4)
All (16)

Group by: Category

- Unreleased Resource: Streams (Control Flow) - [0 / 9]
  - fileReaderServer.java:15 (Streams)
  - fileReaderServer.java:21 (Streams) (multiple issues)
  - fileReaderServer.java:22 (Streams) (multiple issues)
- J2EE Bad Practices: Sockets (Semantic) - [0 / 1]
- Denial of Service (Semantic) - [0 / 2]
  - fileReaderServer.java:16 (Denial of Service)
  - fileReaderServer.java:26 (Denial of Service)
- System Information Leak (Semantic) - [0 / 1]
  - fileReaderServer.java:40 (System Information Leak)
- Resource Injection (Data Flow) - [0 / 1]
  - fileReaderServer.java:11 (Resource Injection)
- Path Manipulation (Data Flow) - [0 / 1]
  - fileReaderServer.java:20 (Path Manipulation)
- J2EE Bad Practices: Leftover Debug Code (Structural) - [0 / 1]

fileReaderServer.java

```
static void main(String[ ] args){


serverPortNumber = Integer.parseInt(args[0]);

erSocket connectionSocket = new ServerSocket(serverPortNumber)

et clientSocket = connectionSocket.accept();

eredReader brSocketInput = new BufferedReader(new InputStreamR
ng filename = brSocketInput.readLine();

cketInput.close();

Reader fr = new FileReader(filename);
eredReader brFile = new BufferedReader(fr);
tStream socketOutput = new PrintStream(clientSocket.getOutputS

ng line = null;
```

Analysis Trace

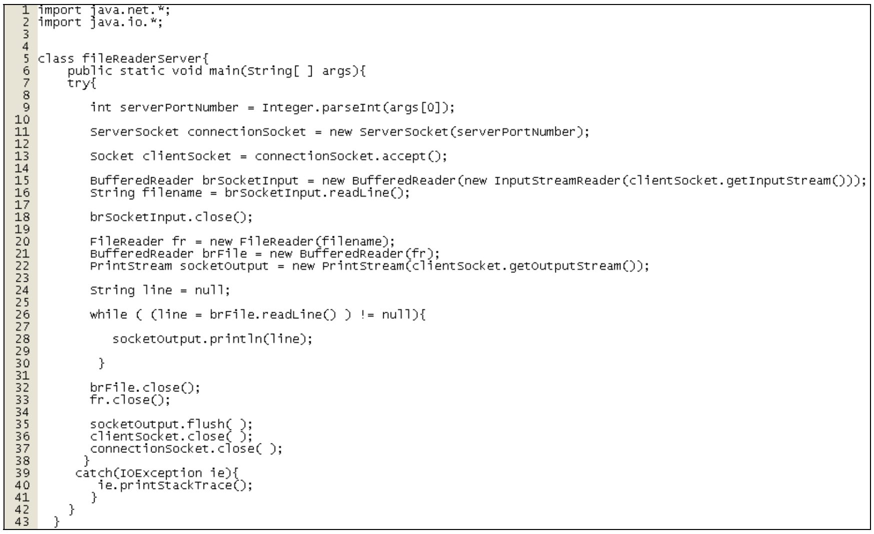
```
main(0) - fileReaderServer.java:6
parseInt(0 --> return) - fileReaderServer.java:9
[assignment to serverPortNumber] - fileReaderServer.java:9
ServerSocket(0) - fileReaderServer.java:11
```

ABSTRACT
Allowing user input to control resource identifiers may enable an attacker to access or modify otherwise protected system resources.

EXPLANATION
A resource injection issue occurs when the following two conditions are met:

Figure 4: Audit Workbench: Issues Panel and Code Editor displaying Details of a Specific Security Issue

The SCA has to be first used in command line (Figure 1) to generate a report, in *.fpr* format (as shown in the first command executed in Figure 1), which can be loaded (second command in Figure 1) into the Audit Workbench utility (screenshot shown in Figure 2), a graphical-user interface utility, included with the Fortify suite of tools. The Workbench interface displays a list of the issues that have been flagged and groups these issues according to their severity (hot, warning, or info). Figure 3 shows a listing of all the issues identified with the file reader server socket program of the case study presented in Section 2.



```java
import java.net.*;
import java.io.*;


class fileReaderServer{
     public static void main(String[ ] args){
     try{

         int serverPortNumber = Integer.parseInt(args[0]);

         ServerSocket connectionSocket = new ServerSocket(serverPortNumber);

         Socket clientSocket = connectionSocket.accept();

         BufferedReader brSocketInput = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
         String filename = brSocketInput.readLine();

         brSocketInput.close();

         FileReader fr = new FileReader(filename);
         BufferedReader brFile = new BufferedReader(fr);
         PrintStream socketOutput = new PrintStream(clientSocket.getOutputStream());

         String line = null;

         while ( (line = brFile.readLine() ) != null){

             socketOutput.println(line);

          }

         brFile.close();
         fr.close();

         socketOutput.flush( );
         clientSocket.close( );
         connectionSocket.close( );
        }
       catch(IOException ie){
          ie.printStackTrace();
         }
     }
  }
```

Figure 5: Case Study: Original Java Code for the File Reader Server Program

The Workbench includes an editor that can highlight the troublesome code identified to be the source of a particular vulnerability listed in the Issues panel, and also allows users to make changes to the code within the application. Figure 4 shows a comprehensive picture of the Issues panel with the code editor. One significant use of the Workbench utility is that for each generic issue flagged by the analyzer, the utility provides a description of the problem and how it may be averted. If users think that a security issue raised by the analyzer is of no interest to them (i.e. can be left unattended in the code), then the Workbench utility can be set to suppress the raising of the issue in subsequent instantiations of running the analyzer. At any point of time, the suppressed issues can be unchecked and the issues will be raised if found in the code being analyzed at that time. Note that it is important to make sure the source code that is being analyzed compiles without any error prior to running it with the SCA.

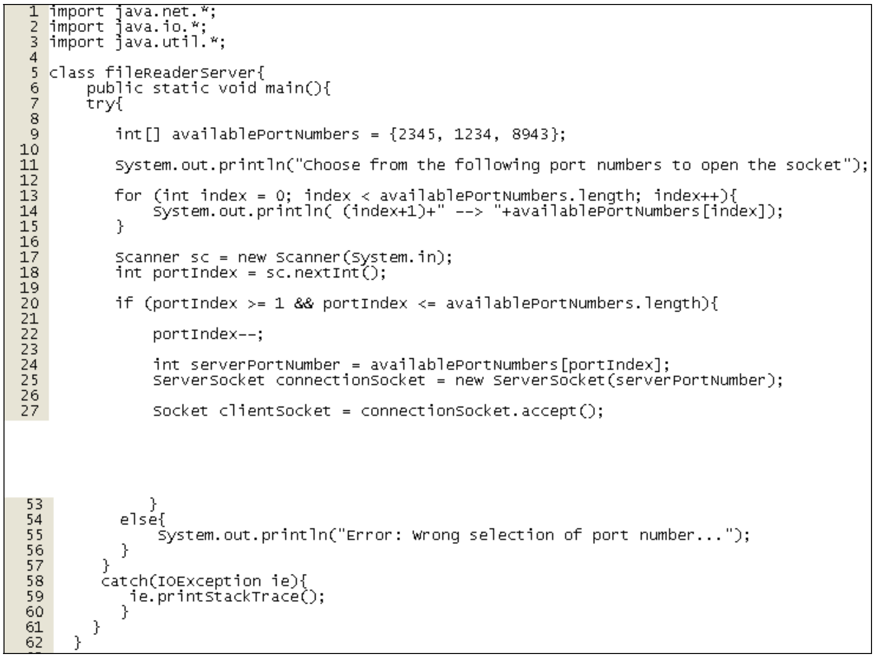
2 Case Study on a Connection-Oriented File Reader Server Socket Program

In this section, we present a case study on a file reader server socket program, based on connection-oriented sockets. For simplicity, the server program is considered to serve only one client. The file reader server basically lets a client to read the contents of a file whose name or the path is sent by the client over a socket and the file is locally stored at the server. The server program (whose original source code is shown in Figure 5) works as follows: An object of class *ServerSocket* is instantiated at a port number input by the user. The *ServerSocket* is the class used to open server sockets that wait on a certain port number (publicly known to the clients) for incoming client requests. Once a client contacts the server, the *ServerSocket* is unblocked (through the accept( ) method) and a *Socket* object (in our program the *clientSocket* object) is created as a reference to communicate with the client at the other side. The server waits for the client to send a filename or a pathname through the socket and reads it through a *BufferedReader* object (*brSocket*). Since the server is not sure of the number of characters that would constitute the filename or the pathname, the server uses the *readLine*( ) method of the *BufferedReader* class to read the filename/pathname as a line of characters stored as a String. This String object is directly passed to the *FileReader* constructor to load the file the client wishes to read. The contents of the file are read line-by-line and sent to the client using an object of the *PrintStream* class invoked on the *ClientSocket* object (of class Socket).

We conduct source code analysis of the file reader server socket program (shown in Figure 5) using the Fortify SCA and the output of all the issues identified are shown in Figure 3. Note that the *poor logging practice* warning shown in Figure 3 is due to the use of print statements. We do not bother to remove the print statements and so neglect those warnings. Similarly, we discard the warning message appearing related to *J2EE Bad Practices*: *Sockets*; J2EE standard considers socket-based communication in web applications as prone to error, and permits the use of sockets only for the purpose of communication with legacy systems when no higher-level protocol is available. The Fortify Source Code Analyzer subscribes to the J2EE standards and flags some of the commonly used J2SE features like sockets as something that is vulnerable in the context of security. As mentioned before, the Audit Workbench does provide the flexibility to turn off these flags which do not appear relevant to the programming environment. The goal of the case study is thus to modify the file reader server socket program (and still does what it is intended to do) to the extent that the source code analyzer only outputs warnings corresponding to the poor logging practice and the use of sockets as bad practice, and all the other vulnerabilities and warnings associated with the program are taken care of (i.e., removed).

2.1 Resource Injection Vulnerability

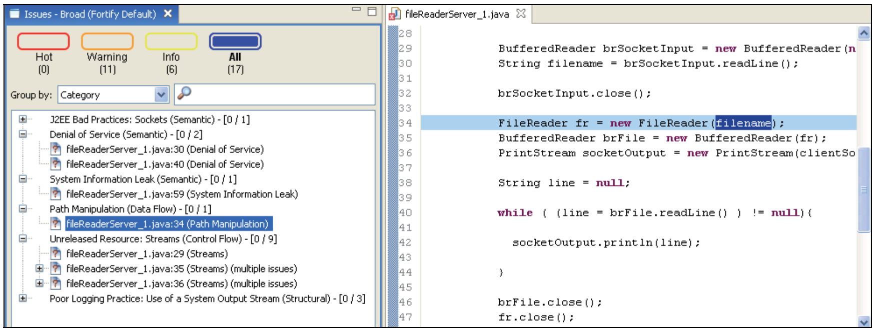
The Resource Injection vulnerability (a dataflow issue) arises because of the functionality to let the user (typically the administrator) starting the server program to open the server socket on any port number of his choice. The vulnerability allows user input to control resource identifiers enabling an attacker to access or modify otherwise protected system resources [1]. In the connection server socket program of Figure 5, a Resource Injection vulnerability exist in line 11, wherein the program opens a socket on the port number whose value is directly input by the user. If the server program has privileges to open the socket at any specified port number and the attacker does not have such a privilege on his own, the Resource Injection vulnerability allows an attacker to gain capability to open a socket at the port number of his choice that would not otherwise be permitted. This way, the program could even give the attacker the ability to transmit sensitive information to a third-party server.



```
import java.net.*;
import java.io.*;
import java.util.*;

class fileReaderServer{
    public static void main(){
    try{

       int[] availablePortNumbers = {2345, 1234, 8943};

       System.out.println("Choose from the following port numbers to open the socket");

       for (int index = 0; index < availablePortNumbers.length; index++){
           System.out.println( (index+1)+" --> "+availablePortNumbers[index]);
       }

       Scanner sc = new Scanner(System.in);
       int portIndex = sc.nextInt();

       if (portIndex >= 1 && portIndex <= availablePortNumbers.length){

           portIndex--;

           int serverPortNumber = availablePortNumbers[portIndex];
           ServerSocket connectionSocket = new ServerSocket(serverPortNumber);

           Socket clientSocket = connectionSocket.accept();



           }
        else{
            System.out.println("Error: Wrong selection of port number...");
        }
      }
     catch(IOException ie){
        ie.printStackTrace();
      }
    }
  }
```

Figure 6: Modification to the File Reader Server Program to Remove the Resource Injection Vulnerability (fileReaderServer_1.java)

We present two solution approaches to completely avoid or at least mitigate the Resource Injection vulnerability: (1) *Use a blacklist or white list*: Blacklisting selectively rejects potentially dangerous characters before further processing the input in a program. However, any such list of unsafe characters is likely to be incomplete and will almost certainly become out of date with time. A white list of allowable characters may be a better strategy because it allows only those inputs whose characters are exclusively listed in the approved set. Due to the difficulty in coming up with a complete list of allowable or non-allowable characters, the approaches of using a blacklist or white list can only mitigate the Resource Injection attack. Nevertheless, if the set of legitimate resource names is too large or too hard to keep track of, it may be more practical to follow a blacklist or white list approach. We will use this approach to remove the Path Manipulation vulnerability in Section 2.2. (2) *Use a level of indirection*: This approach involves creating a list of legitimate resource names that a user is allowed to specify, and only allow the user to select from the list. This approach can help us to completely avoid having Resource Injection vulnerability in the code, because a user cannot directly specify the resource name of his choice and can only chose from what is presented to him. The tradeoff is with the approach of providing a list of port numbers (the resources in our case) to choose from, we are revealing the available port numbers to a user (even though he is constrained only to choose from this list). Note that with the blacklist or white list approach, the user has to merely enter an input of his choice and the program internally processes the input and filters it (thus not revealing information regarding acceptable inputs to the user).



```
J2EE Bad Practices: Sockets (Semantic) - [0 / 1]
Denial of Service (Semantic) - [0 / 2]
    fileReaderServer_1.java:30 (Denial of Service)
    fileReaderServer_1.java:40 (Denial of Service)
System Information Leak (Semantic) - [0 / 1]
    fileReaderServer_1.java:59 (System Information Leak)
Path Manipulation (Data Flow) - [0 / 1]
    fileReaderServer_1.java:34 (Path Manipulation)
Unreleased Resource: Streams (Control Flow) - [0 / 9]
    fileReaderServer_1.java:29 (Streams)
    fileReaderServer_1.java:35 (Streams) (multiple issues)
    fileReaderServer_1.java:36 (Streams) (multiple issues)
Poor Logging Practice: Use of a System Output Stream (Structural) - [0 / 3]
```

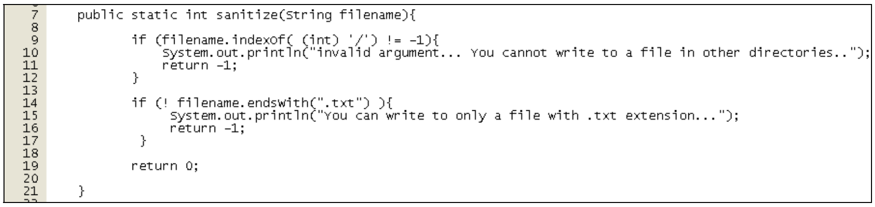
```
    BufferedReader brSocketInput = new BufferedReader(n
    String filename = brSocketInput.readLine();

    brSocketInput.close();

    FileReader fr = new FileReader(filename);
    BufferedReader brFile = new BufferedReader(fr);
    PrintStream socketOutput = new PrintStream(clientSo

    String line = null;

    while ( (line = brFile.readLine() ) != null){

      socketOutput.println(line);

    }

    brFile.close();
    fr.close();
```

Figure 7: Results of the Source Code Analysis of the File Reader Server Program after the Removal of the Resource Injection Vulnerability (fileReaderServer_1.java)

In this section, we present the use of the second approach (i.e. using a level of indirection) to remove the Resource Injection vulnerability (refer to the modified code, especially lines 9 through 25 and 54-56, in Figure 6). The user starting the server program is presented with a list of port numbers to choose from. Each valid port number is presented with a serial number and the user has to choose one among these serial numbers. If the user choice falls outside the valid range of these serial numbers, then the server program terminates printing a simple error message. The limitation is that the user no longer has the liberty to open the server socket at a port number of his choice. This is quite acceptable because often the server sockets are run on specific well-defined port numbers (like HTTP on 80, FTP on 21, etc) and not on arbitrary port numbers, even if the administrator wishes to run the server program on a port number of his choice. Figure 7 presents the results of the source code analysis on the modified connection server socket program (fileReaderServer_1.java) to fix the Resource Injection vulnerability. We have also removed the use of command-line arguments to get inputs and instead use the Scanner class; thus, taking care of the *Leftover debug code* warning.

2.2 Path Manipulation Vulnerability

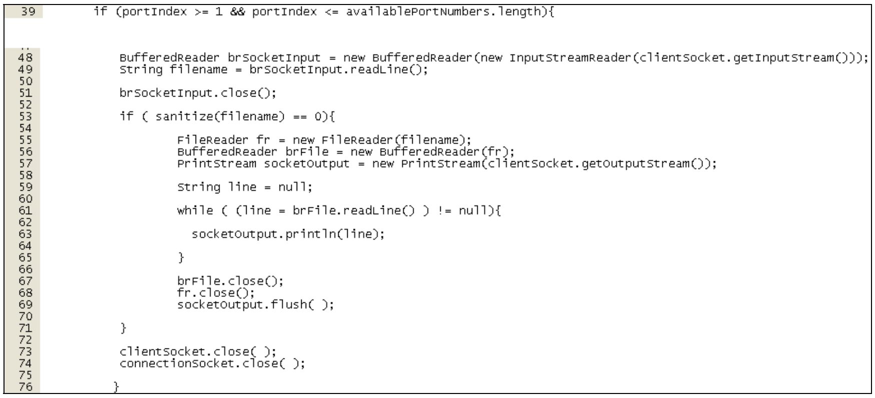
The Path Manipulation vulnerability occurs when user input is directly embedded to the program statements thereby allowing the user to directly control paths employed in file system operations [10]. In our file reader server program, the name or path for the file sent by the client through the socket is received as a String object at the server side, and directly passed onto the FileReader constructor (line 20 in Figure 5). The practice of directly embedding a file name or a path for the file name in the program to access the system resources could be cleverly exploited by a malicious user who may pass an unexpected value for the argument and the consequences of executing the program, especially if it runs with elevated privileges, with that argument may turn out to be fatal. Thus, Path Manipulation vulnerability is a very serious issue and should be definitely not left unattended in a code. Such a vulnerability may enable an attacker to access or modify otherwise protected system resources.



```
public static int sanitize(String filename){

        if (filename.indexOf( (int) '/') != -1){
            System.out.println("Invalid argument... You cannot write to a file in other directories..");
            return -1;
        }

        if (! filename.endsWith(".txt") ){
             System.out.println("You can write to only a file with .txt extension...");
             return -1;
         }

        return 0;

}
```

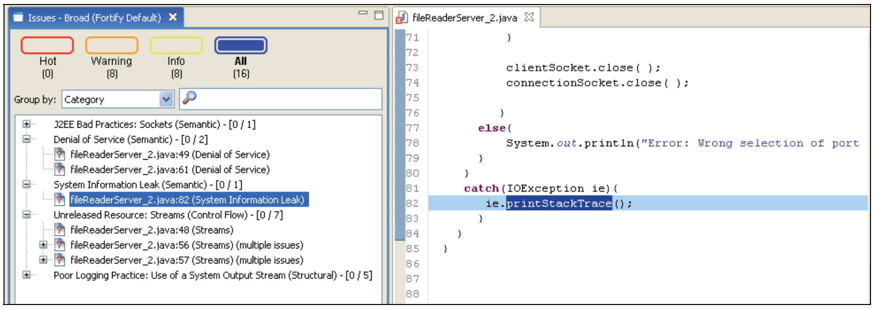
Figure 8: Java Code Snippet for the Sanitize Method to Validate the Filename Received through Socket

As suggested in Section 2.1, we propose to use the approach of filtering user inputs using the blacklist/white list approach. It would not be rather advisable to present the list of file names to the client at the remote side – because this would reveal unnecessary system information to a remote user. It would be rather more prudent to let the client to send the name or the path for the file he wants to open, and we validate the input against a set of allowable and non-allowable characters. In this chapter, we assume the file requested to be read is located in the same directory from which the server program is run, and that the file is a text file. Hence, the last four characters of the input received through the socket should be “.txt” and nothing else (thus, .txt at the end of the String input constitutes a white list). Also, since the user is not permitted to read a file that is in a directory other than the one in which the server program is running, the input should not have any ‘/’ character (constituting a blacklist) to indicate a path for the file to be read. In this chapter, we have implemented the solution of using white list and blacklist through the *sanitize*( ) method, the code for which is illustrated in Figure 8. The modified file server program that calls the sanitize method to validate the filename before opening the file for read is shown in Figure 9. The results of the source code analysis of the modified file reader server program are shown in Figure 10.



```
        if (portIndex >= 1 && portIndex <= availablePortNumbers.length){

..
            BufferedReader brSocketInput = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
            String filename = brSocketInput.readLine();

            brSocketInput.close();

            if ( sanitize(filename) == 0){

                    FileReader fr = new FileReader(filename);
                    BufferedReader brFile = new BufferedReader(fr);
                    PrintStream socketOutput = new PrintStream(clientSocket.getOutputStream());

                    String line = null;

                    while ( (line = brFile.readLine() ) != null){

                      socketOutput.println(line);

                    }

                    brFile.close();
                    fr.close();
                    socketOutput.flush( );

            }

            clientSocket.close( );
            connectionSocket.close( );

          }
```

Figure 9: Modified File Reader Server Socket Program to Call the Sanitize Method to Validate the Filename before Opening it to Read (fileReaderServer_2.java)



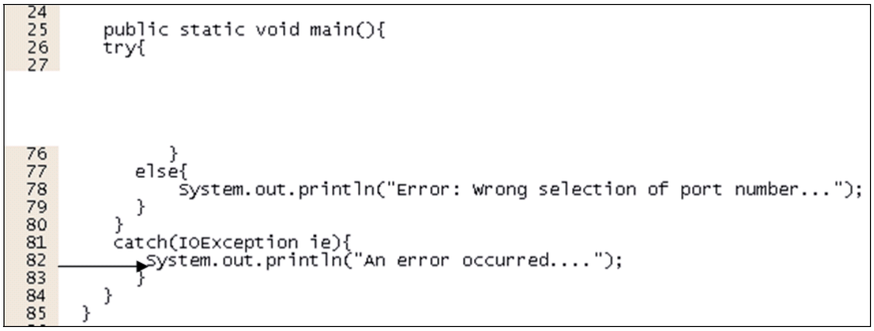
```
J2EE Bad Practices: Sockets (Semantic) - [0 / 1]
Denial of Service (Semantic) - [0 / 2]
    fileReaderServer_2.java:49 (Denial of Service)
    fileReaderServer_2.java:61 (Denial of Service)
System Information Leak (Semantic) - [0 / 1]
    fileReaderServer_2.java:82 (System Information Leak)
Unreleased Resource: Streams (Control Flow) - [0 / 7]
    fileReaderServer_2.java:48 (Streams)
    fileReaderServer_2.java:56 (Streams) (multiple issues)
    fileReaderServer_2.java:57 (Streams) (multiple issues)
Poor Logging Practice: Use of a System Output Stream (Structural) - [0 / 5]
```

```
71              }
72
73              clientSocket.close( );
74              connectionSocket.close( );
75
76            }
77        else{
78              System.out.println("Error: Wrong selection of port
79          }
80      }
81      catch(IOException ie){
82        ie.printStackTrace();
83      }
84    }
85  }
86
87
88
```

Figure 10: Results of the Source Code Analysis of the File Reader Server Program after the Removal of the Path Manipulation Vulnerability (fileReaderServer_2.java)

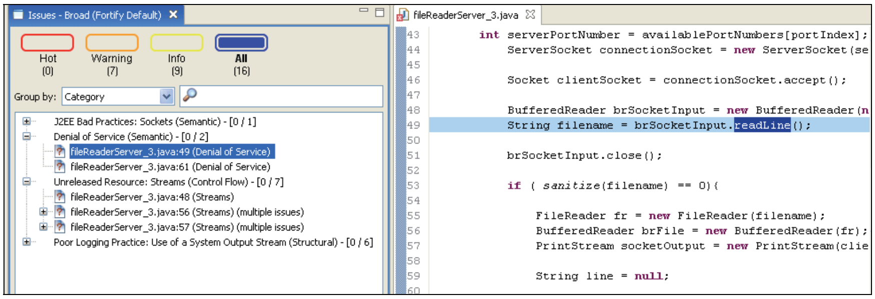
2.3 System Information Leak Vulnerability

The "System Information Leak" vulnerability (a semantic issue) refers to revealing critical system data, program structure including call stack or debugging information that may help an adversary to learn about the software and the system, and form a plan of attack [12]. In our file reader server program (see Figure 6), we observe that in line 82 (as also indicated by the SCA in Audit Workbench Issues panel in Figure 10), the *printStackTrace*( ) method called on the object of the class *IOException* has the vulnerability to leak out sensitive system and program information including its structure and the call stack. While revealing the information about the call stack leading to an exception may be useful for programmers to debug the program and quickly as well as effectively trace out the cause of an error, the *printStackTrace*() method needs to be removed from the final program prior to deployment.



```
24
25     public static void main(){
26     try{
27

76             }
77         else{
78             System.out.println("Error: wrong selection of port number...");
79         }
80       }
81       catch(IOException ie){
82           System.out.println("An error occurred....");
83           }
84      }
85   }
```

Figure 11: Modified File Reader Server Program to Remove the System Information Leak Vulnerability (fileReaderServer_3.java)



```
J2EE Bad Practices: Sockets (Semantic) - [0 / 1]
Denial of Service (Semantic) - [0 / 2]
    fileReaderServer_3.java:49 (Denial of Service)
    fileReaderServer_3.java:61 (Denial of Service)
Unreleased Resource: Streams (Control Flow) - [0 / 7]
    fileReaderServer_3.java:48 (Streams)
    fileReaderServer_3.java:56 (Streams) (multiple issues)
    fileReaderServer_3.java:57 (Streams) (multiple issues)
Poor Logging Practice: Use of a System Output Stream (Structural) - [0 / 6]
```

fileReaderServer_3.java

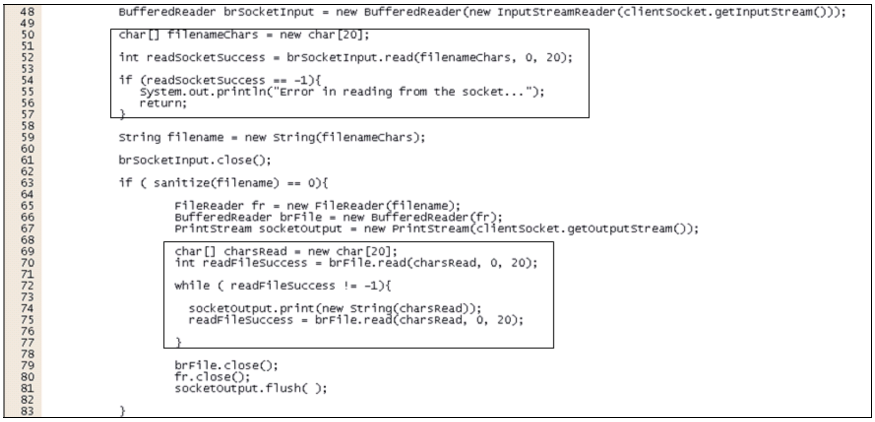
```
        int serverPortNumber = availablePortNumbers[portIndex];
            ServerSocket connectionSocket = new ServerSocket(se

            Socket clientSocket = connectionSocket.accept();

            BufferedReader brSocketInput = new BufferedReader(n
            String filename = brSocketInput.readLine();

            brSocketInput.close();

            if ( sanitize(filename) == 0){

                FileReader fr = new FileReader(filename);
                BufferedReader brFile = new BufferedReader(fr);
                PrintStream socketOutput = new PrintStream(clie

                String line = null;
```

Figure 12: Results of the Source Code Analyzer of fileReaderServer_3.java after the Removal of the System Information Leak Vulnerability and Indicating the Presence of the Denial of Service Vulnerability

A simple fix to this vulnerability is not to reveal much information about the error, and simply state that an error has occurred. The attacker, if he was contemplating to leverage the error information to plan for an attack, would not be able to gain much information from the error message. In this context, we remove the call to the *printStackTrace*( ) method from line 82 and replace it with a print statement just indicating that an error occurred. The modified version of the file reader server socket program is shown in Figure 11 and the results of its source code analysis are shown in Figure 12.

2.4 Denial of Service Vulnerability

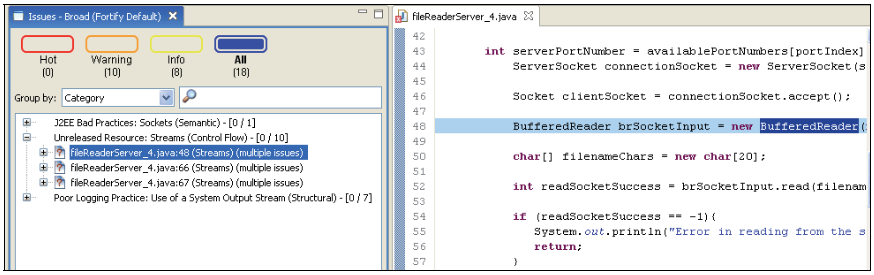
A ‘Denial of Service’ vulnerability (a semantic issue) is the one with which an attacker can cause the program to crash or make it unavailable to legitimate users [10]. Lines 49 and 61 of the file reader server socket program (as indicated in Figure 12) contain the Denial of Service vulnerability, and this is attributed to the use of the readLine( ) method. It is always not a good idea to read a line of characters from a file through a program because the line could contain an arbitrary number of characters, without a prescribed upper limit. An attacker could misuse this and force the program to read an unbounded amount of input as a line through the readLine( ) method. An attacker can take advantage of this code to cause an *OutOfMemoryException* or to consume a large amount of memory so that the program spends more time performing garbage collection or runs out of memory during some subsequent operation.



```
        BufferedReader brSocketInput = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));

        char[] filenameChars = new char[20];

        int readSocketSuccess = brSocketInput.read(filenameChars, 0, 20);

        if (readSocketSuccess == -1){
           System.out.println("Error in reading from the socket...");
           return;
        }

        String filename = new String(filenameChars);

        brSocketInput.close();

        if ( sanitize(filename) == 0){

                FileReader fr = new FileReader(filename);
                BufferedReader brFile = new BufferedReader(fr);
                PrintStream socketOutput = new PrintStream(clientSocket.getOutputStream());

                char[] charsRead = new char[20];
                int readFileSuccess = brFile.read(charsRead, 0, 20);

                while ( readFileSuccess != -1){

                  socketOutput.print(new String(charsRead));
                  readFileSuccess = brFile.read(charsRead, 0, 20);

                }

                brFile.close();
                fr.close();
                socketOutput.flush( );

        }
```

Figure 13: Modified Code for the File Reader Server Socket Program to Remove the Denial of Service Vulnerability by Replacing the readLine( ) Method with the read( ) Method (fileReaderServer_4.java)

The solution we suggest is to impose an upper bound on the number of characters that can be read from the file and buffered at a time (i.e., in one single read operation). In this context, we suggest to use the *read*( ) method of the *BufferedReader* class that takes three arguments: a character array to which the characters read from the buffer are stored, the starting index in the character array to begin storing characters and the number of characters to be read from the buffer stream. In the context of lines 49 through 54 in the fileReaderServer_4.java program (boxed in Figure 13), we replace the readLine( ) method with a read( ) method to read the name of the file or the pathname. If we do not read sufficient number of characters, then the name of the file stored in the String object *filename* would be incorrect and this could be detected through the current implementation of the sanitize( ) method (Figure 8) itself, as the last four characters of the file has to end in ".txt". In the context of lines 62 through 71 (boxed in Figure 13), there would not be a problem in reading certain number of characters (rather than a line of characters) for every read operation, because – whatever is read is stored as a String and is sent across the socket.



```
J2EE Bad Practices: Sockets (Semantic) - [0 / 1]
Unreleased Resource: Streams (Control Flow) - [0 / 10]
    fileReaderServer_4.java:48 (Streams) (multiple issues)
    fileReaderServer_4.java:66 (Streams) (multiple issues)
    fileReaderServer_4.java:67 (Streams) (multiple issues)
Poor Logging Practice: Use of a System Output Stream (Structural) - [0 / 7]
```

```
42
43          int serverPortNumber = availablePortNumbers[portIndex]
44              ServerSocket connectionSocket = new ServerSocket(s
45
46              Socket clientSocket = connectionSocket.accept();
47
48              BufferedReader brSocketInput = new BufferedReader(
49
50              char[] filenameChars = new char[20];
51
52              int readSocketSuccess = brSocketInput.read(filenam
53
54              if (readSocketSuccess == -1){
55                 System.out.println("Error in reading from the s
56                 return;
57              }
```

Figure 14: Results of the Source Code Analysis of the File Reader Server Program after Removing the Denial of Service Vulnerability (fileReaderServer_4.java)

In order to preserve the structure of the text, we have to simply use the print( ) method instead of the println( ) method of the PrintStream class. If there is a line break in the text of the file, it would be captured through an embedded '\n' line break character and sent across the socket.

In this section, we choose to read 20 characters for each read operation at both the instances and replace the readLine( ) method with the read( ) method accordingly. In the second case, we read every 20 characters from the file, and the last read operation may read less than 20 characters if there are not sufficient characters. The subsequent read will return -1 if no character is read. Our logic (as shown in lines 63-71 of the code in Figure 13) is to check for the return value of the read operation every time and exit the while loop if the return value is -1, indicating the entire file has been read.

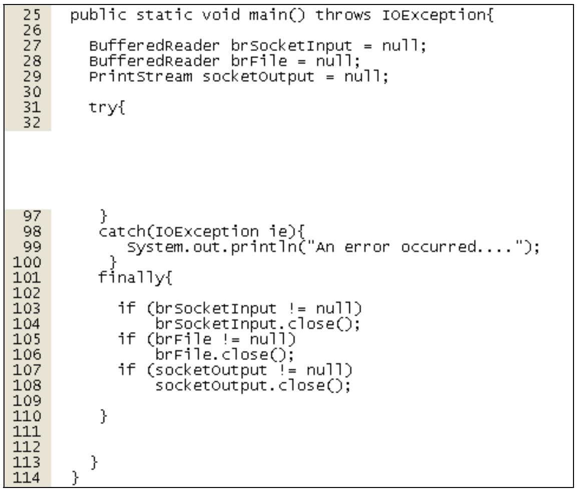
Note that the length 20 we used here is arbitrary, and could be even set to 100. The bottom line is there should be a definite upper bound on the number of characters that can be read into the character buffer, and one should not be allowed to read an arbitrary number of characters with no defined upper limit. The modified file reader server socket program is shown in Figure 13 and the results of the source code analysis are shown in Figure 14.

2.5 Unreleased Resource Vulnerability

The "Unreleased Resource" vulnerability (a control flow issue) occurs if the program has been coded in such a way that it can potentially fail to release a system resource [11]. In our file reader server socket program, the vulnerability arose due to the use of the *BufferedReader* stream class (lines 48 and 66 of Figure 13) to read the contents from the socket and the text file and the *PrintStream* class to send the contents across the socket to the remote side. Even though we have called the close( ) methods on the objects of the above two stream classes immediately after their use is no longer needed, it may be possible that due to abrupt termination of the program, the close( ) method calls are not executed (also listed in the Issues panel of Figure 14). One possible reason for the program control to skip the execution of the close( ) method calls could be a file read error, which could happen if the name of the file read from the socket is not in the location from which the program is trying to open and read the file. Another reason (which is very unlikely to happen though, given the smaller size of the file) could be that there is no sufficient memory in the system to load the contents of the text file and read them. Similarly, in the case of sending across the socket, there may be an error if the client abruptly closes the socket while the server attempts to transmit them across the socket. Either way, if any such buffer reading or sending errors occur, the program control immediately shifts from the *try* block to the *catch* block and the streams corresponding to the *BufferedReader* and *PrintStream* classes will never be released until the operating system explicitly forces the release of these resources upon the termination of the program. From a security standpoint, if an attacker could sense the presence of Unreleased Resource vulnerability in a program, he can intentionally trigger resource leaks and failures in the operating environment of the program (like making the file unavailable to be read or closing the socket from the remote side, if the client is compromised) to cause a depletion of the resource pool.

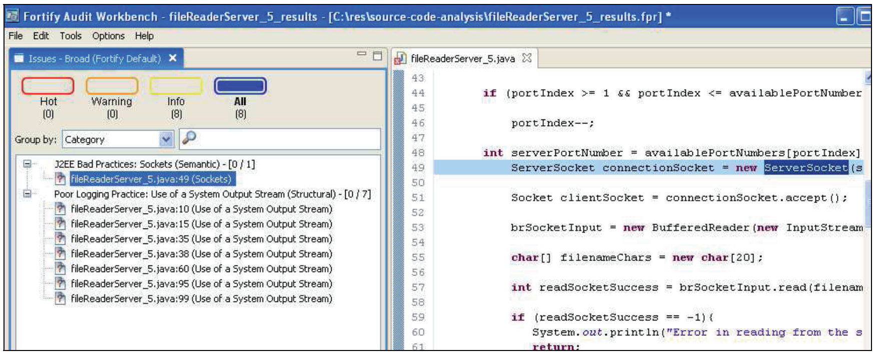
The solution we suggest is to add a *finally* { … } block after the *try* {…} *catch* {…} blocks and release all the resources that were used by the code in the corresponding *try* block. Note that in order to do so, the variables associated with the resources have to be declared outside and before the *try* block so that they can be accessed inside the *finally* block. In our case, we have to declare the stream objects of the *BufferedReader* and *PrintStream* classes outside the *try* block and close them explicitly in the *finally* block. The modified code segment (fileReaderServre_5.java) is shown in Figure 15. The results generated from analyzing the fileReaderServer_5.java code with the Source Code Analyzer are shown in Figure 16. Note that in order to close the two *FileReader* and *BufferedReader* streams in lines 66 and 68 of the *finally* {...} block, we have to declare that the main function throws the *IOException* in line 7.

Note that as shown in Figure 15, the reason why we are insisting on including the close( ) method calls on the two stream objects in a *finally* block instead of a *catch* block, even though it is supposed to catch the *IOException*, is that in case a *try* block can generate multiple exceptions – there has to be multiple *catch* blocks for the *try* block, one for each exception, and these *catch* blocks have to be listed in the order of increasing scope – i.e., the exception that is the bottommost in the hierarchy of exceptions has to be caught first, followed by exceptions further up in the hierarchy. However, if at run-time, an exception higher up in the hierarchy is generated, the control transfers to the *catch* block of that particular exception, and only the subsequent *catch* blocks are executed, and not the *catch* blocks prior to it. This way, if we had included the close( ) methods in the *catch* block for the *IOException* class and relied on it to be called in case of a file read error, there might be a situation that another *catch* block downstream is called due to the generation of an exception higher up in the exception hierarchy, and the two stream objects would not be released. Thus, in any situation, we do not recommend releasing system resources inside *catch* blocks. The results of the source code analysis of the final version of the file reader server socket program (fileReaderServer_5.java) are shown in Figure 16, after removing all the five main vulnerabilities in the code. The only warnings remaining in Figure 16 are those corresponding to the *poor logging practice* and *J2EE Bad Practices*: *Sockets*, which are not critical to be removed for standard Java programming environments.



```
public static void main() throws IOException{

  BufferedReader brSocketInput = null;
  BufferedReader brFile = null;
  PrintStream socketOutput = null;

  try{




   }
   catch(IOException ie){
      System.out.println("An error occurred....");
    }
   finally{

     if (brSocketInput != null)
         brSocketInput.close();
     if (brFile != null)
         brFile.close();
     if (socketOutput != null)
         socketOutput.close();

   }


  }
}
```

Figure 15: Modified Code Segment to Remove the Unreleased Resource Vulnerability (fileReaderServer_5.java)



Fortify Audit Workbench - fileReaderServer_5_results - [C:\res\source-code-analysis\fileReaderServer_5_results.fpr] *

```
J2EE Bad Practices: Sockets (Semantic) - [0 / 1]
    fileReaderServer_5.java:49 (Sockets)
Poor Logging Practice: Use of a System Output Stream (Structural) - [0 / 7]
    fileReaderServer_5.java:10 (Use of a System Output Stream)
    fileReaderServer_5.java:15 (Use of a System Output Stream)
    fileReaderServer_5.java:35 (Use of a System Output Stream)
    fileReaderServer_5.java:38 (Use of a System Output Stream)
    fileReaderServer_5.java:60 (Use of a System Output Stream)
    fileReaderServer_5.java:95 (Use of a System Output Stream)
    fileReaderServer_5.java:99 (Use of a System Output Stream)
```

```

    if (portIndex >= 1 && portIndex <= availablePortNumber

        portIndex--;

    int serverPortNumber = availablePortNumbers[portIndex]
        ServerSocket connectionSocket = new ServerSocket(s

        Socket clientSocket = connectionSocket.accept();

        brSocketInput = new BufferedReader(new InputStream

        char[] filenameChars = new char[20];

        int readSocketSuccess = brSocketInput.read(filenam

        if (readSocketSuccess == -1){
           System.out.println("Error in reading from the s
          return;
```

Figure 16: Results of Source Code Analysis on the Final Version of the File Reader Server Socket Program with all the Main Vulnerabilities and Warnings Removed (fileReaderServer_5.java)

Before we conclude, we also argue that it is not advisable to include a *finalize*( ) method for the particular classes of the objects for which the resources allocated need to be reclaimed. In order for an object's *finalize*( ) method to be invoked, the garbage collector must first of all determine that the object is eligible for garbage collection. However, the garbage collector is not required to run unless the Java Virtual Machine (JVM) is low on memory, and hence there is no guarantee that an object's *finalize*( ) method will be invoked in an expedient fashion. Even if the garbage collector gets to run, all the resources will be reclaimed in a short period of time, and this can lead to "bursty" performance and a reduction in the overall system throughput. Such an effect is more pronounced as the load on the system increases. Also, it is possible for the thread executing the *finalize*( ) method to hang if the resource reclamation operation requires communication over a network or a database connection to complete the operation.

3 Conclusions and Future Work

Software security is a rapidly growing field and is most sought after in both industry and academics. With the development of automated tools such as Fortify Source Code Analyzer, it becomes more tenable for a software developer to fix, in-house, the vulnerabilities associated with the software prior to its release and reduce the number of patches that need to be applied to the software after its release. In this chapter, we have discussed the use of an automated tool called the Source Code Analyzer (SCA), developed by Fortify, Inc., and illustrated the use of its command line and graphical user interface (Audit Workbench) options to present and analyze the vulnerabilities identified in a software program. The SCA could be used in a variety of platforms and several object-oriented programming languages. We present an exhaustive case study of a file reader server socket program, developed in Java, which looks fine at the outset; but is analyzed to contain critical vulnerabilities that could have serious impacts when exploited.

The five different vulnerabilities we have studied in this research are: Resource Injection vulnerability, Path Manipulation vulnerability, System Information Leak vulnerability, Denial of Service vulnerability, and Unreleased Resource vulnerability in the context of streams. We discussed the reasons these vulnerabilities appeared in the code and how they could be exploited if left unattended and the consequences of an attack. We have provided detailed solutions to efficiently and effectively remove each of these vulnerabilities, presented the appropriate code snippets and the results of source code analysis when the vulnerabilities are fixed one after the other. The tradeoffs incurred due to the incorporation of appropriate solutions to fix these vulnerabilities are the increase in code size and decrease in the comfort level for a naïve authentic user who could face some initial technical difficulties in getting the program to run as desired. With generic error messages that are not so detailed, an authentic (but relatively unfamiliar) user ends up spending more time to run the system as desired. The original file reader server program had 43 lines of code, and the final version of the program (fileReaderServer_5.java) contains 114 lines – thus, an increase in the size of the code by a factor of about 2.65 (i.e., 165% increase). However, the increase in code size is worth because even if one the above 5 vulnerabilities is exploited by an attacker, it could be catastrophic for the entire network hosting the server.

As part of future work, we plan to conduct exhaustive source code analysis on network socket programs developed in C/C++, for Windows and Linux platforms, and analyze their impacts and develop effective solutions to fix (i.e., completely remove or mitigate the effects as much as possible) the characteristic vulnerabilities identified for the specific platform/ programming language. Even though the code snippets provided as solutions to remove the various software security vulnerabilities discussed in this chapter are written in Java, the solutions proposed and implemented here for each of the vulnerabilities are more generic and can be appropriately modified and applied in other programming language environments.

4 Acknowledgments

The work leading to this module is partly funded through the U. S. National Science Foundation (NSF) CCLI/TUES grant (DUE-0941959) on “Incorporating Systems Security and Software Security in Senior Projects.” The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the funding agency.

5 Exercises

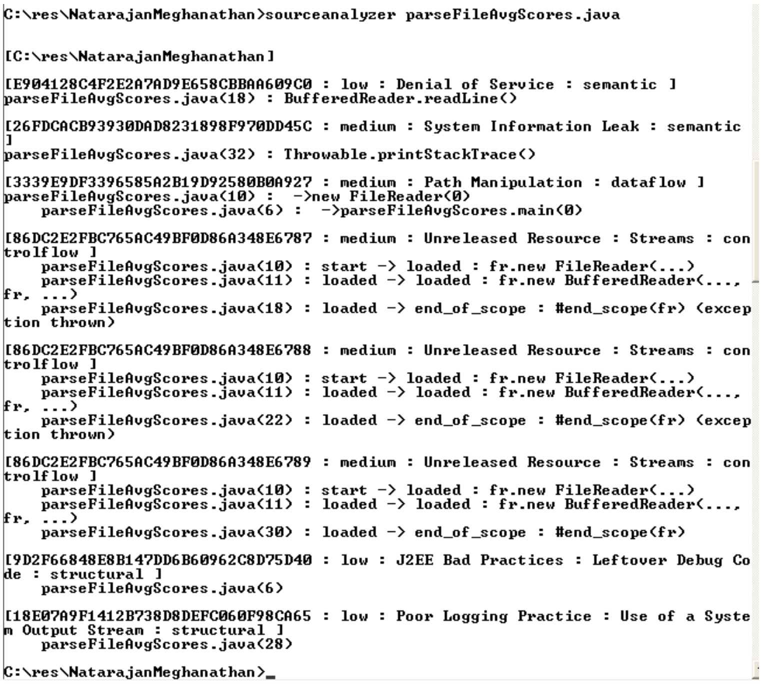
1. Conduct source code analysis on the following Java program that is supposed to read a list of integer scores from a text file (one integer per line in the file) and compute the average score. The number of integers to compute the average score is not known a priori and has to be determined based on the number of integers read from the file. Fix all the vulnerabilities (except the Poor Logging practice warnings raised due to the System.out.println( ) statements and the J2EE Bad Practices due to the use of the main() function) that are identified by the Sourceanalyzer.

Given Code

```
import java.util.*;
import java.io.*;

class parseFileAvgScores{

   public static void main(String[] args){

   try{

     FileReader fr = new FileReader(args[0]);
     BufferedReader br = new BufferedReader(fr);

     String line = null;

     int sum = 0;
     int numScores = 0;
```

```
      while ( (line = br.readLine() ) != null){

        StringTokenizer stk = new StringTokenizer(line);
        String strScore = stk.nextToken();
        int score = Integer.parseInt(strScore);
        sum += score;
        numScores++;
      }

      System.out.println("Average Score: "+( ((double) sum)/numScores));

      }
    catch(Exception e){
       e.printStackTrace();
     }
    }
}
```

The following is the output returned by the sourceanalyzer when run on the given Java code.



```
C:\res\NatarajanMeghanathan>sourceanalyzer parseFileAvgScores.java


[C:\res\NatarajanMeghanathan]

[E904128C4F2E2A7AD9E658CBBAA609C0 : low : Denial of Service : semantic ]
parseFileAvgScores.java(18) : BufferedReader.readLine()

[26FDCACB93930DAD8231898F970DD45C : medium : System Information Leak : semantic
]
parseFileAvgScores.java(32) : Throwable.printStackTrace()

[3339E9DF3396585A2B19D92580B0A927 : medium : Path Manipulation : dataflow ]
parseFileAvgScores.java(10) :  ->new FileReader(0)
    parseFileAvgScores.java(6) :  ->parseFileAvgScores.main(0)

[86DC2E2FBC765AC49BF0D86A348E6787 : medium : Unreleased Resource : Streams : con
trolflow ]
    parseFileAvgScores.java(10) : start -> loaded : fr.new FileReader(...)
    parseFileAvgScores.java(11) : loaded -> loaded : fr.new BufferedReader(...,
fr, ...)
    parseFileAvgScores.java(18) : loaded -> end_of_scope : #end_scope(fr) (excep
tion thrown)

[86DC2E2FBC765AC49BF0D86A348E6788 : medium : Unreleased Resource : Streams : con
trolflow ]
    parseFileAvgScores.java(10) : start -> loaded : fr.new FileReader(...)
    parseFileAvgScores.java(11) : loaded -> loaded : fr.new BufferedReader(...,
fr, ...)
    parseFileAvgScores.java(22) : loaded -> end_of_scope : #end_scope(fr) (excep
tion thrown)

[86DC2E2FBC765AC49BF0D86A348E6789 : medium : Unreleased Resource : Streams : con
trolflow ]
    parseFileAvgScores.java(10) : start -> loaded : fr.new FileReader(...)
    parseFileAvgScores.java(11) : loaded -> loaded : fr.new BufferedReader(...,
fr, ...)
    parseFileAvgScores.java(30) : loaded -> end_of_scope : #end_scope(fr)

[9D2F66848E8B147DD6B60962C8D75D40 : low : J2EE Bad Practices : Leftover Debug Co
de : structural ]
    parseFileAvgScores.java(6)

[18E07A9F1412B738D8DEFC060F98CA65 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    parseFileAvgScores.java(28)

C:\res\NatarajanMeghanathan>_
```

2. Conduct the source code analysis on the following file writer program that is supposed to accept 5 lines of information from the user and write them out to a file, line-by-line, as they are input by the user. Fix all the vulnerabilities (except the Poor Logging practice warnings raised due to the System.out.println( ) statements) that are identified by the Sourceanalyzer. Fix the vulnerabilities one-by-one and show the modified code in each phase.

File Writer Program
```
import java.io.*;

class fileWriter {

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

    try{
         FileWriter fw = new FileWriter(args[0]);
       PrintWriter pw = new PrintWriter(fw);
       BufferedReader        br        =        new        BufferedReader(new
InputStreamReader(System.in));

       for (int lineNum = 1; lineNum <=5; lineNum++){
               System.out.print("Enter line # "+lineNum+" : ");
           String line = br.readLine( );
           pw.println(line);
       }

       pw.close( );
       fw.close( );

      }
     catch(IOException ie){
        ie.printStackTrace( );
     }
  }

}
```

3. Conduct source code analysis on the following Java program that is supposed to allow a user to write to the file *Logfile.dat* provided the password entered by the user (as a command line input captured through args[0]) matches with the password *3dTAqb.7* that is currently hard coded in the program. Fix all the vulnerabilities (except the Poor Logging practice warnings raised due to the System.out.println( ) statements) that are identified by the Sourceanalyzer.

Given Code

```
import java.io.*;
import java.util.*;

class SCAExample1
{
      public static void main(String args[ ])
      {
            try
            {
                  File f = new File("Logfile.dat");
                  boolean access_granted = false;
                  String password = "";
                  int integer = 5;

                  if (args.length == 1)
                  {
                        System.out.println("Checking command-line password");
                        password = password + args[0];
                        if (password.equals("3dTAqb.7"))
                        {
                              access_granted = true;
                              System.out.println("Password matches.");
                        }
                        else
                              System.out.println("Command-line password does
not match");
                  }//end if

                  if (access_granted)
                  {
                        System.out.println("Access granted!");
                        PrintWriter out = new PrintWriter(new
FileOutputStream(f, true));
                        out.println( );
                        out.print("Updated...");
                        out.println( );
                        out.flush( );
                        out.close( );
                  }//end if

            }//end try
            catch (Exception e)
            {
                  System.out.println("an error has occured.");
                  e.printStackTrace( );
            }
      }//end main

}//end class
```

References

[1] B. Chess, and J. West, *Secure Programming with Static Analysis*, Addison Wesley, 1st Edition, Boston, MA, USA, 2008.

[2] M. R. Stytz, and S. B. Banks, "Dynamic Software Security Testing," *IEEE Security and Privacy*, vol. 4, no. 3, pp. 77-79, 2006.

[3] D. Baca, "Static Code Analysis to Detect Software Security Vulnerabilities – Does Experience Matter?," *Proceedings of the IEEE International Conference on Availability, Reliability and Security*, pp. 804-810, 2009.

[4] P. R. Caseley, and M. J. Hadley, "Assessing the Effectiveness of Static Code Analysis," *Proceedings of the 1st Institution of Engineering and Technology International Conference on System Safety*, pp. 227-237, 2006.

[5] I. A. Tondel, M. G. Jaatun and J. Jensen, "Learning from Software Security Testing," *Proceedings of the International Conference on Software Testing Verification and Validation Workshop*, pp. 286-294, 2008.

[6] H. Mcheick, H. Dhiab, M. Dbouk and R. Mcheik, "Detecting Type Errors and Secure Coding in C/C++ Applications," *Proceedings of the IEEE/ACS International Conference on Computer Systems and Applications*, pp. 1-9, 2010.

[7] M. Mantere, I. Uusitalo and J. Roning, "Comparison of Static Code Analysis Tools," *Proceedings of the 3rd International Conference on Emerging Security Information, Systems and Technologies*, pp. 15-22, 2009.

[8] J. Novak, A. Krajnc and R. Zontar, "Taxonomy of Static Code Analysis Tools," *Proceedings of the 33rd IEEE International Conference on Information and Communication Technology, Electronics and Microelectronics*, pp. 418-422, 2010.

[9] https://www.fortify.com/products/hpfssc/source-code-analyzer.html, last accessed: July 2, 2012.

[10] M. G. Graff, and K. R. Van Wyk, *Secure Coding: Principles and Practices*, O'Reilly Media, Sebastopol, CA, USA, 2003.

[11] M. Howard, D. Leblanc, and J. Viega, *24 Deadly Sins of Software Security*: *Programming Flaws and How to Fix them*, McGraw-Hill, New York City, NY, USA, 2009.

[12] J. A. Whittaker, *How to Break Software*, Addison-Wesley, Boston, MA, USA, 2002.