This document includes software design and analysis of homework 5. Source code is included in under the ‘src’ directory in HW5\_dwilso.zip. Additionally, a jar and execution scripts (.bat and .sh) are included for running a preconfigured server and up to three clients. The jar is named SecureChat and is in the ‘bin’ directory. The demo scripts are in the ‘demo’ directory.

# Secure Chat Server

## Source Control

https://github.com/dwilso95/secure-chat-server

## Requirements

Build a secure, chat server to facilitate “party line” messaging from multiple clients. Each client should receive every message for which they are authorized. Authorization is handled using mandatory access control (MAC). Each client and message has an assigned clearance label. Clients can only send and receive messages at or below their clearance label. This is enforced by the server. The server must handle multithreaded, non-blocking I/O, use SSL for authentication rather than the usual username and password, and validate message signatures to guarantee no data was changed. The server also must log all messages including DN, message label, message, and timestamp to a file using XML.

## Execution

The server and client require a configuration file called a context. This file defines ports to use, the the location of key stores, trust stores, password, and other configurable information. The server also requires a clearance file which defines the clearance label for each client. **No guarantees are made on the server running on pre-Java 9.**

## Server

The server is started by executing:

**Pre-Java 9**java java.xml.bind -cp HW5\_dwilso.jar chat.server.RunServer ${context.location}

**Java 9**java --add-modules java.xml.bind -cp HW5\_dwilso.jar chat.server.RunServer ${context.location}  
  
A server can be started with a preconfigured context, keystore, and truststore, by running the included demo scripts **runServer.bat** or **runServer.sh**. These should be executed from the root directory. **No guarantees are made on the server running on pre-Java 9.**

**Example:** demo\runServer.bat

## Client

Client is started by executing:

**Pre-Java 9**java -cp HW5\_dwilso.jar chat.client.RunClient ${context.location}

**Java 9**java --add-modules java.xml.bind -cp HW5\_dwilso.jar chat.server.RunClient ${context.location}  
  
A client can be started with a preconfigured context, keystore, and truststore, by running the included demo scripts **runClient.bat** or **runClient.sh**. There are three potential clients that can be run. To run each specify a single argument, 1, 2, or 3. These should be executed from the root directory.

**Example:** demo\runClient.bat 1

To send a message, the client must use the format ${clearance}\_${message}. To stop a client, type ‘exit’ with no clearance. **No guarantees are made on the client running on pre-Java 9.**

## Design

The two entry points for this project are the Server and Client classes. Each of these is dependent on a Context which defines all configurable parameters. Contexts are defined in a simple file format. A parameter is defined per line. Each line is a property name and value delimited by ‘=’.

The Server class starts a thread which begins listening for socket connections. For each connection it creates two threads, a reader and writer. The server class creates and uses an instance of MessageVerifer to check the signature validity of a message. MessageVerifier uses an instance of ClearanceService for making decisions on which messages can be sent and received by which users. This service is also configured using a file formatted as a context.

The server uses the MessageQueue class to pass messages between threads. Server threads write messages to the queue and read messages from other threads off the queue. The MessageQueue class utilizes java.concurrent data structures like ConcurrentHashMap and ConcurrentLinkedQueue to guarantee messages are sent and received in order.

The Client class creates a socket connection to the server. After authentication it creates, like the server, two threads. One thread handles reading while the other handles writing. Clients send messages and its associated clearance. All messages received are printed to the console.

Both Client and Server utilize factory utility classes for creating key/trust stores, signatures, and SSL sockets/servers instances. These factories rely on some hardcoded arguments which could easily be included in server/client context files in the future.

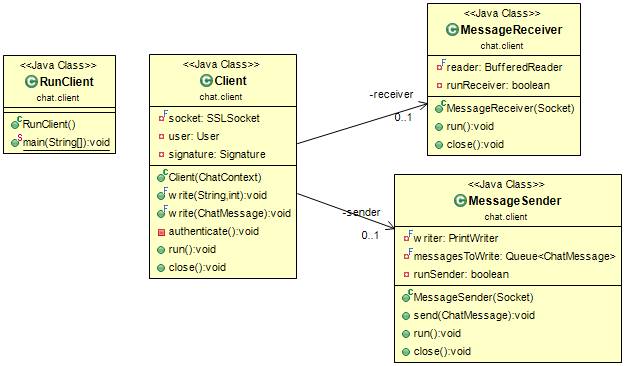
## UML Diagrams

### Package chat C:\Users\dwilson\workspace\secure-chat-server\class_diagrams\chat.png

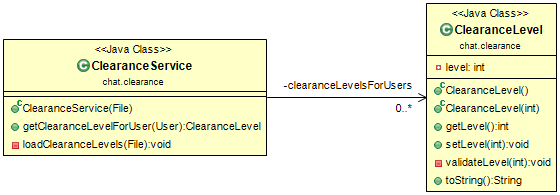
## Package chat.server

### C:\Users\dwilson\workspace\secure-chat-server\class_diagrams\chat.server.png

### Package chat.client



### Package chat.clearance



## Security Discussion

This server implementation offers security through three main mechanisms. First authentication is performed using private/public keys. This is achieved using Java key stores. Rather than utilizing a username/password like most web technologies, the server knows which clients have access based on which certificates are in its trust store. This is not ideal in the real world, as someone may want to sign in from many different devices, each one with a different private/public key pair.

The keys provided for demoing the secure chat server implementation are also problematic. All keys used are based on self-signed certificates. These certificates can only be trusted within this test. In the real world, a certificate authority is used as the basis of trust for certificates. These trust chains are the basis for almost all encryption on the web today. I was unable to obtain a certificate for this project from a trusted authority like Thawte or Symantec. The industry has moved to no longer giving out temporary certificates until they verify they are being issued to a registered domain. This validation is performed based on the domain’s registration information.

Authorization, specifically MAC, is performed using the concept of clearances. Each client and message is assigned a clearance. This enables to server to control sending messages only to the clients which have the message’s clearance or above. While simple, the implementation is functionally correct and secure. All clearance information used for making access control decisions is kept only within the server and unchangeable by the client.

The last piece of security included in this chat server is message verification. This is performed using signatures. The client signs the message content with its private key. The server verifies the signature using the public key. Signing the messages helps guarantee that what was delivered is what was sent and that it was not modified along the way.

# Demo

## Configuration

### Server

Port – 5000  
 Timeout – 60 sec

### Client 1

DN - CN=Client 1, OU=APL, O=JHU, L=Laurel, ST=Maryland, C=US  
Clearance – 4

### Client 2

DN - CN=Client 2, OU=APL, O=JHU, L=Laurel, ST=Maryland, C=US  
Clearance – 1

### Client 3

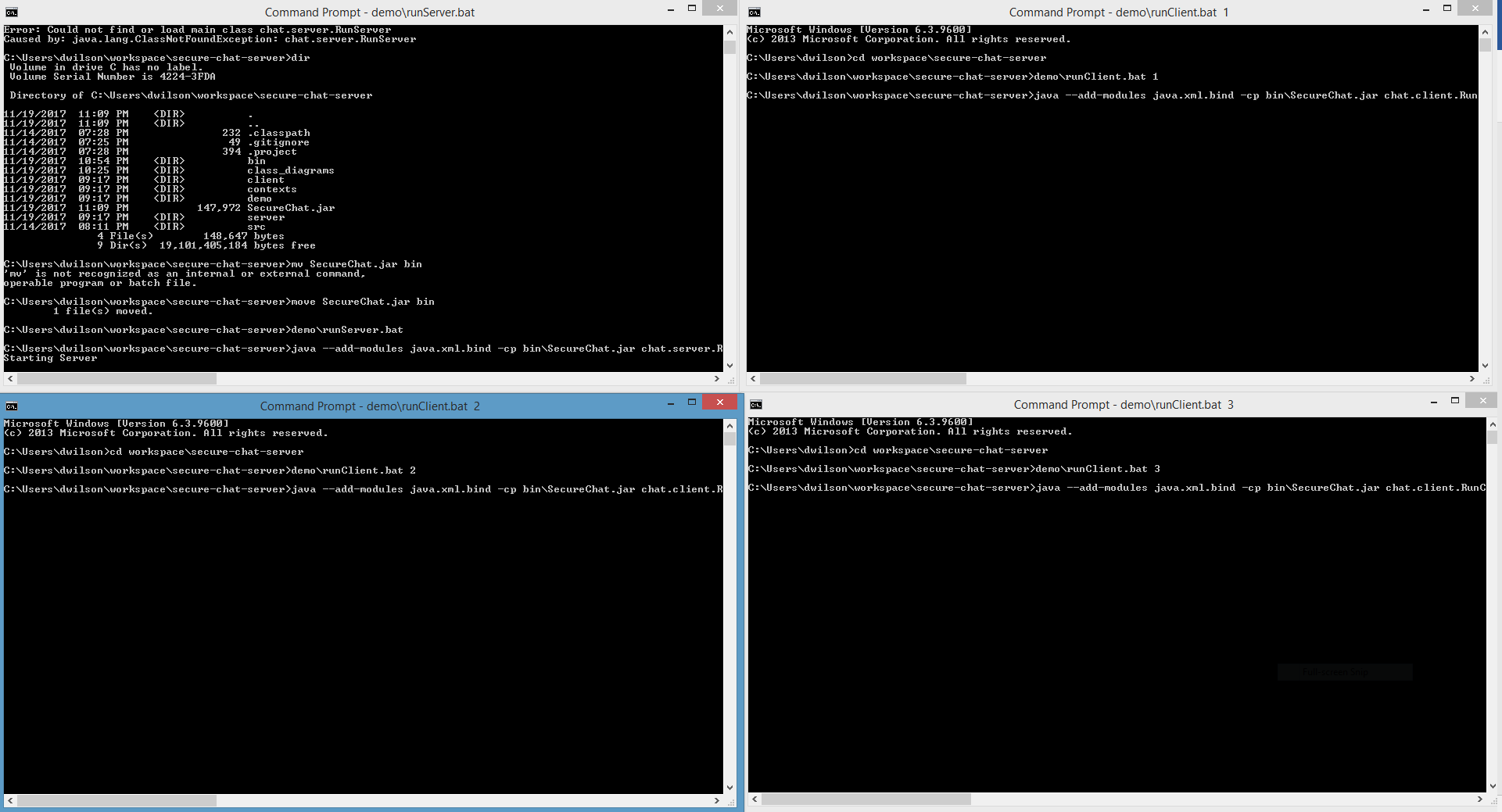
DN - CN=Client 1, OU=APL, O=JHU, L=Laurel, ST=Maryland, C=US   
Clearance – 4

## Execution

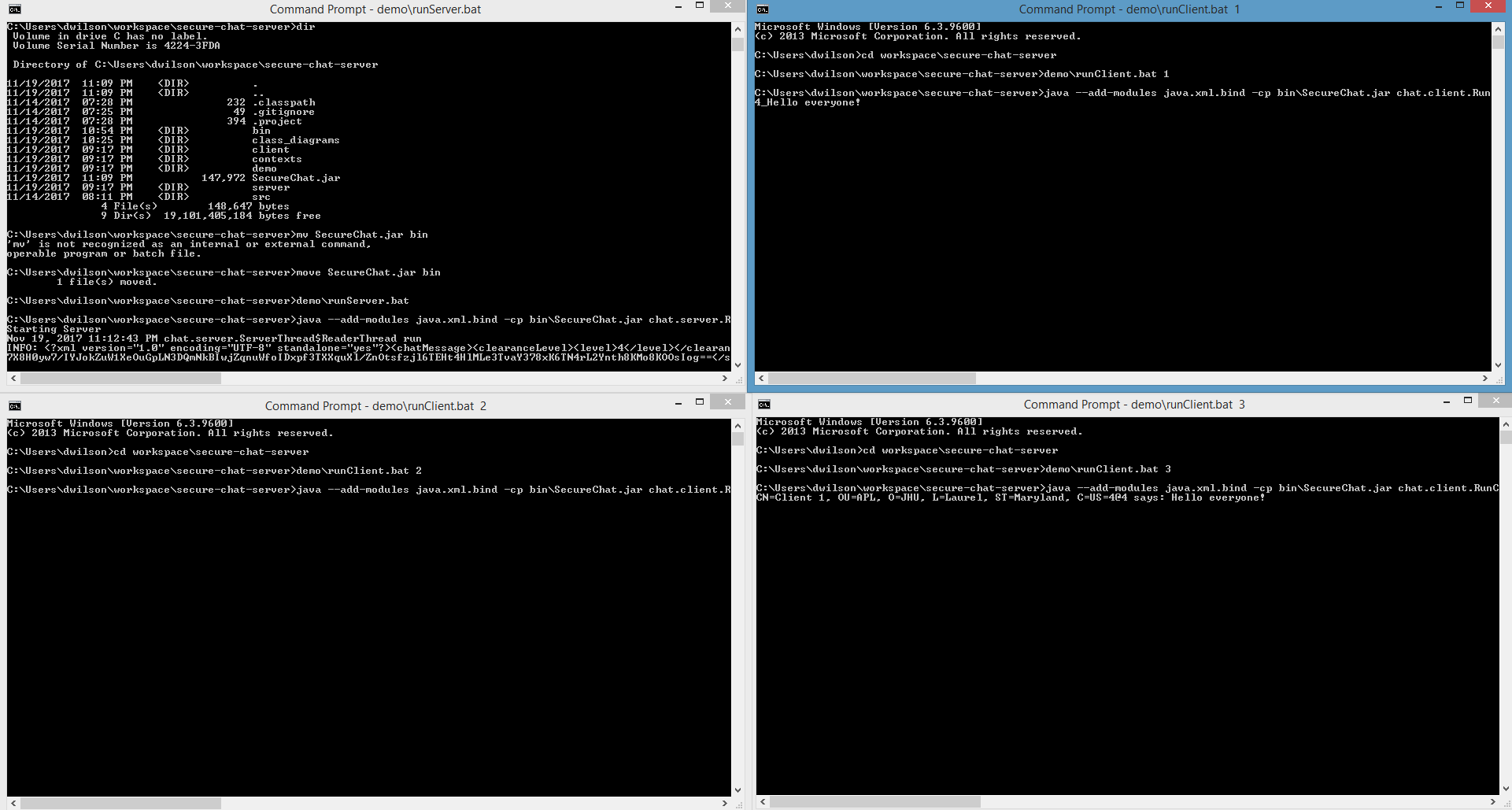
The server is run using **demo\runServer.bat**

Clients are run using **demo\runClient.bat 1**, **demo\runClient.bat 2**, and **demo\runClient.bat 3**

## Screenshots Starting server and clients 1 – 3 in 4 different windows:

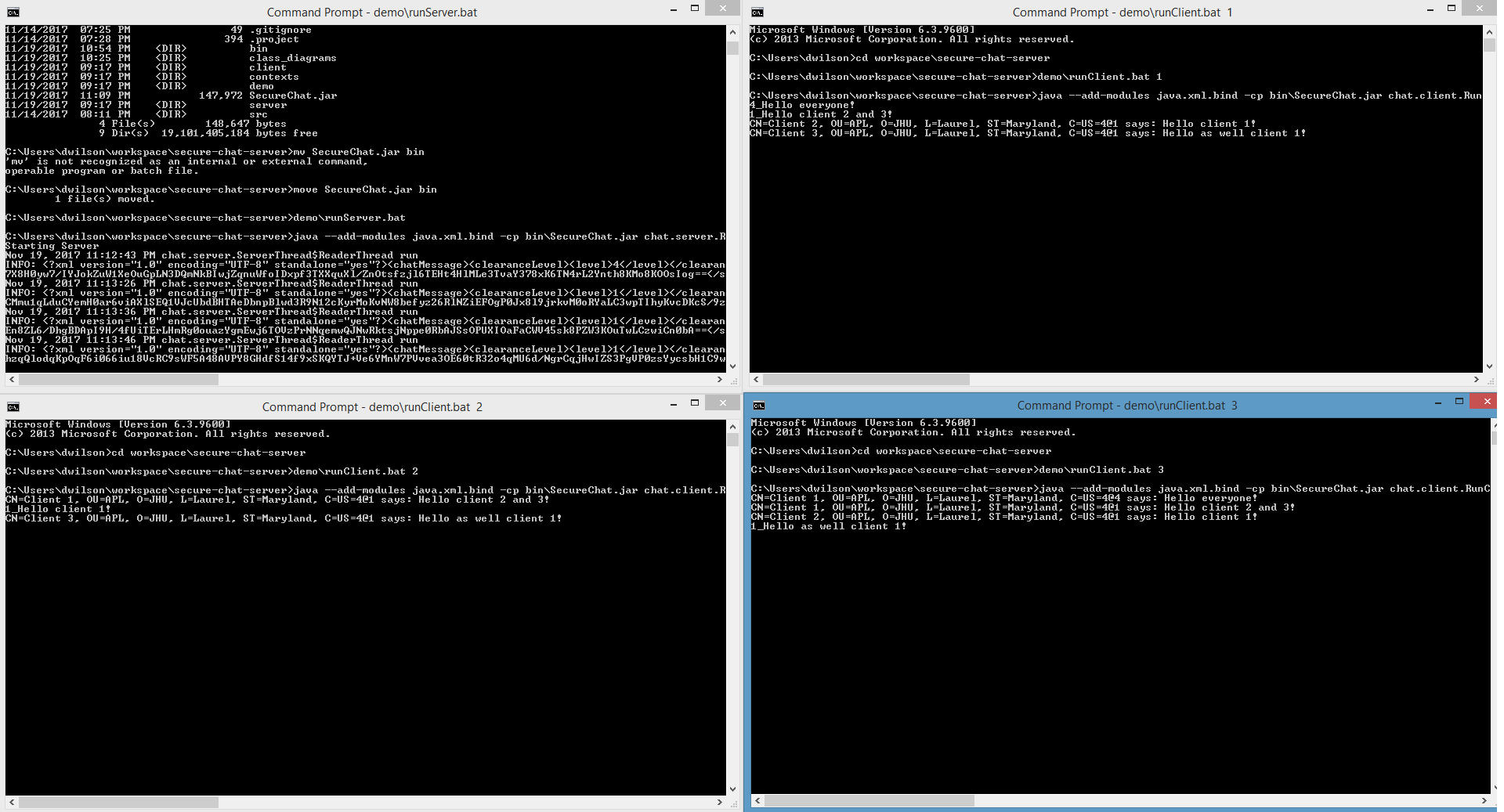


### Sending message from client 1 with clearance 4:



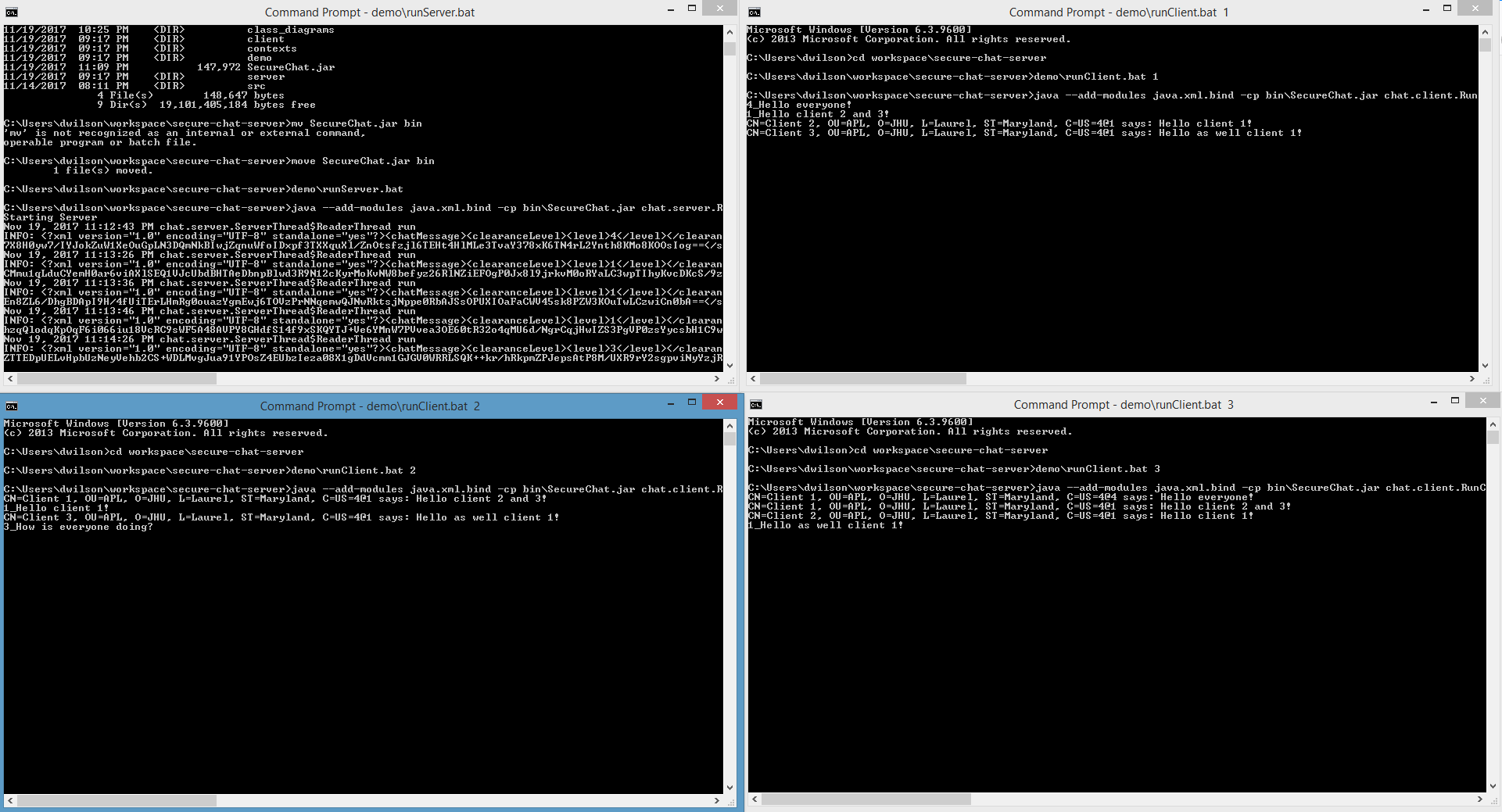
Notice client 2 does not receive the message.

### Sending messages from clients 1, 2 and 3 with clearance 1:



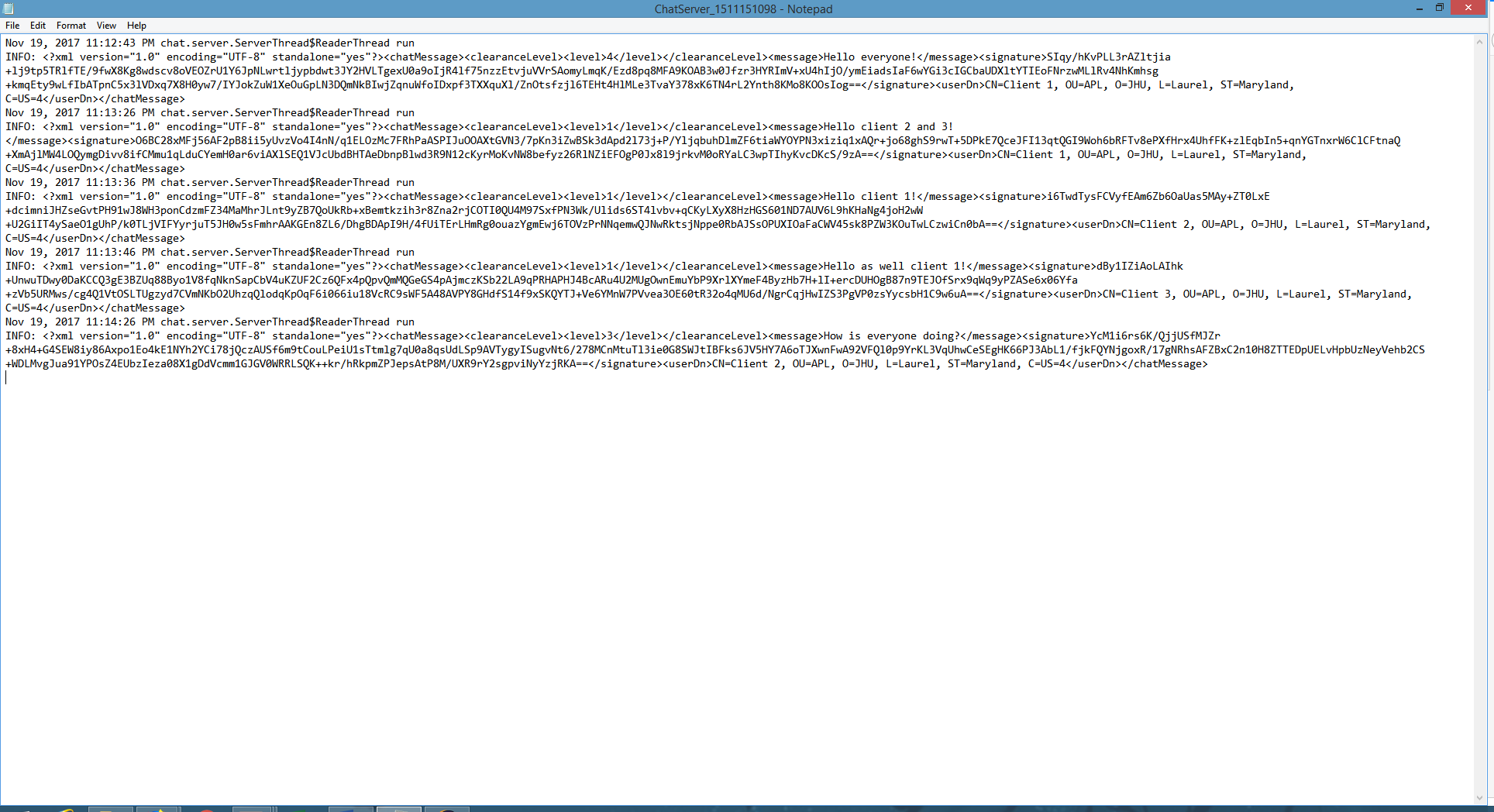
Notice every client receives the messages.

### Client 2 sends a message at clearance 3:



Notice no one receives the message because client 2 is not allowed to send messages at this clearance.

### Server log output:



All attempted messages are in the log.