Project Guidelines

1 Introduction

The project for ECSE 489 this semester will build on the skills you have developed by completing the first three experiments (Wireshark, OPNET, and implementing a chat-server client using sockets). This handout describes three possible projects. Your team should either choose one of these three or you should propose your own. If you wish to propose your own project topic, please discuss it with the instructor as soon as possible.

The project involves two demos and a final report. The deliverables for each demo, as well as the content of the report, will depend on which project option you choose.

The deadlines and grade breakdown for the project are as follows:

- Notify the course staff (via email) which project your group has chosen by March 10
- Demo 1 (worth 10% of your final grade): Schedule a demo time with a TA.
    - Monday section: Demo on or before Monday, March 24
    - Tuesday section: demo on or before Tuesday, March 25
- Demo 2 (worth 10% of your final grade): Schedule a demo time with a TA.
    - Monday section: Demo on April 7
    - Tuesday section: Demo by April 8
- Report (worth 15% of your final grade):
    - Both sections: Due by midnight on Friday, April 11.

Project 1: Extensions to Chat Client

This project extends the chat client created in Experiment 3 to include additional features. The first feature you are asked to add is securing the connection between client and server via SSL. The second feature you are asked to add is the ability to send files to a user and receive files sent to you. This first feature is there to protect the system users' names and passwords from attack from any other party which might wish to do harm.

In order to complete this project, you need to be familiar with SQL programming (though not a big proficiency is necessary), Java sockets, and Java file-storage techniques. If this is not your team, then another project may be advisable.

SSL Security

In order to implement the features requested, you will of course need to implement the server part of this project. You will be given server code used in Experiment 3 (written in Java). You are then asked to modify the current ServerSocket code in the networking.ServerSocketListener to support hosting an SSL secured socket. In order to modify this you will need to use the following class (javax.net.SSLServerSocketFactory) instead of the standard "ServerSocket" in java.net.*.

This will allow you to create a SSL secured server socket once you create a certificate to sign the socket against and secure it. You can test that you have a valid SSL encrypted channel via the openssl s_client utility as

```
openssl s_client –connect <hostname>:<port>
```

which will bind to the SSL socket and allow you to run text commands to the server. This is essentially a SSL secured telnet application. As a small note with the s_client application in openssl, sending a capitol "Q" will close the client and not be transmitted to the server, therefore a suggestion is to send your commands in lower-case.

You will also need to add the SSL security in the client-side application you created in Experiment 3.

Sending Files via the Chat Server

This problem has multiple issues which need to be addresses. In order to send and receive files through the chat server you need some medium for storing the file on the server. You will also need to define a method of notifying the server that a file is incoming (this is inherently extending the API from Experiment 3). The server will need to realize that a file is incoming and who it is destined for.

A simple suggestion for storing the file on the server is via a blob (which is just a bunch of data) in the SQLite database[1] on the server. This is how messages are stored in the server currently, and simply adding a table would be an easy solution. Another solution is to store the filename in SQL and then the actual files on disk. The solution is up to you, however you must define what you chose and why.

Considerations for determining if a file is incoming are the following

1. Need to determine that a file is about to be received
2. Who is the file destined for?
3. How big is the file?
4. Does file type matter? (You'll want to think about this before just jumping in)

[1] See http://www.sqlite.org/datatype3.html for SQLite datatype information

Lastly you will of course need to extend your client application to support uploading and transmitting files to the server (with the appropriate transmission commands).

Division of Labor

This is not a mandatory division of labor, however with the two features being well separated in terms of the systems they affect it is recommended that one person focus on SSL and the other on sending files. If one partner finishes easier, they can easily continue by working on the UI for the sending files feature. Again this is only a recommendation, however it is a logical division of work.

Demonstration 1

For demo 1, you will need to show your team's current progress in extending the chat software. You should at least have been able to modify the server software in order to support SSL by now. Having a well signed certificate should also be done by now and you should be able to show how you generated it. You should also be able to demonstrate that the server is correctly supporting a SSL socket.

You should also be able to explain the current progress in the sending files feature. Having the ability to fully send a file is unnecessary, however being able to store a file in the SQL might be good to be done at this point.

Demonstration 2

For demo 2, you should have a fully functional system and be able to demonstrate all features requested. You should also be able to show via Wireshark that the information transmitted is indeed encrypted via SSL. You're UI, as in Experiment 3, should be well designed and thought-out for ease-of-use to an end-user. You will be asked about the various design decisions you needed to make in order to complete the project.

Project 2: Linux and IPTables

[*Note:* If you choose this project you will need to use your personal laptops because the school computers don't have VirtualBox or VMWare installed.]

For this project, you will implement and test a firewall. First, you will simulate a firewall scenario which implements restrictions described below using OPNET. Then, you will implement the firewall on a real Linux host using the IPTables package to enforce the firewall rules. An easy-to-follow HowTo tutorial on IPTables (on Ubuntu) can be found at the following website: https://help.ubuntu.com/community/IptablesHowTo. Also a quick background on IpTables can be found on Wikipedia here: http://en.wikipedia.org/wiki/Iptables.

Firewall Rules

The following scenario must be implemented in OPNET and IPTables:

- 1 Server (hosting an FTP server, SSH server, and HTTP server)
- 1 Firewall
- 3 Clients (Client A requesting only FTP, B requesting all 3 services, and C only requesting HTTP)
    - The clients can query these systems using simple BASH shell scripts which utilize wget (for http and ftp) and ssh <hostname> -c "echo hi" for the ssh test

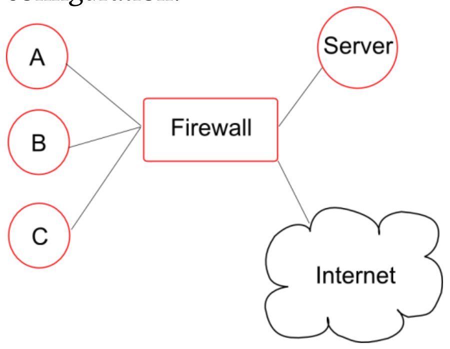
Once you have these 3 systems, you need to build the following firewall rules (see the figure below as well).

- Client A can only access FTP from the server and HTTP from the world (i.e., internet cloud)
- Client B can only access SSH and HTTP from the server and cannot access external addresses.
- Client C can't access anything from the server but can access anything from external addresses (i.e., internet cloud).
- The server can't be accessed from the internet (only from the internal subnet) and it also can't access the internet.

You will need to setup routing to the world for all nodes (an open system) then start restricting by IP Address in the firewall.

OPNET

This is the same design system you have used before, therefore the model you need to create shouldn't be too complex. The firewall and hosts should be connected in the following configuration:



Linux Hosts implementing the rules

You'll need to install VirtualBox and create five virtual machines (one firewall, one server, and three client) with a Linux distribution. The recommended version is Ubuntu 12.04 LTS or greater. This can be downloaded from:
http://www.ubuntu.com/download/ .
Once you have the base system installed, you'll want to install the following packages (if not already installed) on the server

```
apt-get install –y wget vstfpd apache openssh-server
```

and on the clients:

```
apt-get install –y wget openssh-server
```

and finally the firewall:

```
apt-get install –y iptables-persistent openssh-server
```

[Note: All iptables commands will need sudo privileges]
Once you have the environment setup, you'll want to start building the firewall using iptables rules. On the firewall, you'll also want to do iptables-save > /etc/iptables/rules.v4. The tool iptables-persistent allows one to save the iptables configurations so that, upon a system restart, the settings aren't lost.
You can install all these systems as virtual machines in VMWare (commercial) or VirtualBox (free). VirtualBox is available at: https://www.virtualbox.org/. When installing the VMs you'll also need to define a private network for the nodes you're creating. This is done in the settings of each node as you create it. Make sure you don't set it on NAT because that will share your actual network card and all public access can't be blocked. The only node which will need 3 interfaces is the firewall (1 NAT for public internet access, 1 to the Server, and 1 to the clients). If you have any issues, email Sean to meet and he can help you work through any issues.

HINT

To flush the firewall (if you need to reset the configuration) and start fresh, use the following commands (as sudo or root)

```
iptables –F
iptables –X
iptables -t nat –F
iptables -t nat –X
iptables -t mangle –F
iptables -t mangle –X
iptables -P INPUT ACCEPT
iptables -P FORWARD ACCEPT
iptables -P OUTPUT ACCEPT
```

In order to modify the OPNET firewall rules on a router (you cannot just use basic Firewalls as in the OPNET lab, you need to specify port rules as you would in iptables directly) you can find the routing table at
// TODO: TABLE LOCATION

For Demo 1:

For the first demo, you'll need to demonstrate your OPNET design to the TA as well as any design choices you decided to implement. You should also explain how you plan to setup the iptables systems and build the firewall for real. By the first demo, you should show that the Linux virtual machines are setup and installed (but you do not have to have completed the ipTables configurations yet.)

For Demo 2:

You need to show that all systems follow the specified rules. We recommend implementing tests using command prompt calls (wget, etc.), as well as demonstrating, using Wireshark captures of the packets in the network, that traffic is being filtered correctly. You should explain any challenges you experienced and how you solved them.

Project 3: Leaky Bucket Filtering for Rate Limiting (a.k.a. Traffic Shaping)

In this project you'll be designing a traffic shaping algorithm and demonstrating it in a custom-built application. Traffic shaping is the act of throttling traffic over a network connection to meet pre-specified rules. In this case we wish to limit overall traffic throughput and bandwidth with a leaky bucket algorithm (see: http://en.wikipedia.org/wiki/Leaky_bucket).

Leaky Bucket

When companies or enterprises throttle a user's traffic, the most common way to do this is through the leaky bucket algorithm. The leaky bucket buffers user traffic in order to reduce the throughput, and if the leaky bucket buffer becomes full, it may drop user traffic packets. The leaky bucket can also be used to "smooth" bursty traffic by delaying portions of it so that the resulting traffic stream is spread out over more time.

Leaky Bucket Implementation

For this project you will implement a client and server using sockets (you can build on top of existing code, e.g., from previous assignments). Once the client establishes a connection to the server, data will only be sent from the server to the client.

The server will implement two types of traffic sources: constant bit rate, and bursty. The constant bit rate source sends small packets containing 800 bytes of traffic are sent every 100ms, mimicking streaming audio. The bursty source enters a loop where it waits for 15 seconds (sending nothing) and then sends 120k bits of data immediately, in one burst.

The leaky bucket should be implemented at the server by writing a wrapper around the TCP Socket class. The server and client should be multi-threaded. When the server application is launched, any parameters required for the leaky bucket should either be read from a configuration file, or the user should be prompted for these parameters. When the client application is started, it should accept either a configuration file or command line input indicating how many connections to open to the server, and what type of traffic to use on each connection (constant bit rate or bursty). Then the client will open multiple parallel connections to the server. On each connection, it will send a request which includes which type of traffic source to use, and whether or not to activate the leaky bucket. (The format of the request packet is to be designed by the project group.) The server will continue sending traffic until the connection is closed by the client.

After you implement your client and server, you will conduct experiments using Wireshark to capture the traffic between client and server. Your experiments should illustrate the effects that the leaky bucket has on both types of traffic sources.

For Demo 1:

For demonstration 1, you should describe the way you will implement the leaky bucket filter, the two traffic sources, and you should also describe the protocol you will use to signal between the client and server. You do *not* need to have all of these implemented for the first demonstration, but we recommend starting your implementation as soon as possible.

For Demo 2:

For the second demo, you should demonstrate the entire system, including using Wireshark to illustrate the effect that the leaky bucket has on each traffic source.