CS 452 – Secure Inter-process Communication (IPC)

Due: 24-Nov-2020

Overview

The objective of this assignment is to gain experience in application level programming with TCP stream sockets and implement a secure **`sendfile`** utility between a client and a server over our phoenix virtual machines (VMs). You must write your code in C++ and use wrapper classes to exploit C's low-level capabilities.

Deliverables

Your submission to the W drive must include a zipped folder (<lastname><1st name>_lab2.zip) that contains the following:

- sendfile code base
- makefile
- README

The README file must include a short header containing your name, username, and the assignment title. It should also include a short description of your code, tasks accomplished, and how to compile, execute, and interpret the output of your programs. Any questions posed in this lab write-up should contain answers your README file.

Basics of `sendfile`

Your **`sendfile`** program is a simple networking program that connects to a remote server and sends an encrypted file over the network to the remote server. The `sendfile` program will start a client in one machine and a server at the other and can be viewed as a data transport over-the-network utility. The range of port numbers that will be open is between 9000 and 9999. Here is an example of some of its usages:

`Sendfile` as a client:

The most common usage of `sendfile` is as a client; that is, as a program that connects to a remote server. It sends a secure file to the server and the server sends them to `stdout`. When sendfile is run, the following prompts must be issued to the user:

a. Connect to IP address
b. Port number
c. File to be sent
d. Packet size (in KB)
e. Encryption key

`sendfile` as a server:

`sendfile` also functions as a simple server that will listen for a connection and write all received data to stdout. The following prompts must be issued to the user:

a. Connect to IP address
b. Port number
c. Save to file (default is stdout)
d. Encryption key

Required format of a session between client and server

IMPORTANT: Failure to follow the exact format shown below for your output will result in a 10% penalty.

Assuming that we connect from one of our VMs to another, transferring a file called sample.txt. The file size is 1G:

Client (when verbose flag is set on phoenix1.cs.uwec.edu	**Server** (on phoenix2.cs.uwec.edu)
$ run client	$ run server
Connect to IP address: 10.35.195.47	Connect to IP address: 10.35.195.47
Port #: 9285	Port#: 9825
File to be sent: sample.txt	Save file to: sample.out (default: stdout)
Pkt size: 32	
Enter encryption key: usa123	Enter encryption key: usa123
Sent encrypted packet# 0 – encrypted as 6f37 … 5b8f	Rec packet# 0 – encrypted as 6f37… 5b8f
Sent encrypted packet# 1 – encrypted as 453f … 9cf2	Rec packet# 1 – encrypted as 453f… 9cf2
:	:
:	:
Send Success!	Receive success!
MD5:	MD5:
7d41700dbc59cd95c3076a47eb168e6c	7d41700dbc59cd95c3076a47eb168e6c

The MD5 hash is a digital signature utility that produces a 128-bit unique number that is used to authenticate the integrity of the files you have transferred over, i.e., the file at the source must have the

Required format

The MD5 hash is a digital signature utility that produces a 128-bit unique number that is used to authenticate the integrity of the files you have transferred over, i.e., the file at the source must have the same md5 hash value as the one at the destination.

To use the md5 hash on our linux machines:

```
$ md5sum sample.txt
  7d41700dbc59cd95c3076a47eb168e6c sample.txt
```

Encryption

Before sending the file over, each side must know the encryption/decryption key. These are prompted and entered before sending the file over (see session dialog above). The entire file is broken up into packets based on the packet size specified in the command line, i.e., each packet cannot be over that size. A large file will obviously require the sender to send a large number of packets each time.

Each packet is encrypted using a simple exclusive or. E.g.,
Plaintext = hello world
Key = fox
Cipher test = Plaintext ^ Key (^ = XOR)

```
h e l l o   w o r l d
f o x f o x f o x f o
? ? ? ? ? ? ? ? ? ?
```

In hex = 5f 82 7a 46 …… (this should be printed in session dialog**)**

Note: When printing the encrypted hex representation, do not print the entire hex string but only the first 4 hexes and the last 4 hexes since the string may be long due to a large packet size)

File Programming Preliminaries and Socket Programming

Below are some brief descriptions of the requisite C functions needed for this lab. First, file I/O is discussed, particularly file streams, and following, the basics of socket programming is discussed, including C-stlyle pseudo-code examples.

File Streams

- **Opening a file**: You are probably already familiar with the basic file opening procededure open(), which returns a file descriptor, an int. Additionally, there are other ways for manipulating files in C using a file pointer.Consider fopen() below:

```
FILE * fopen(const car * filename, const car * restrict mode)
```

fopen() opens a file named filename with the appropriate mode (e.g., “r” for reading, “w” for writing, “r+” for reading and writing, and etc.). The important part to consider is that a file pointer is returned rather than file descriptor. A file pointer allows you to interpret files as streams of bits with a read head pointed to some part of the file.

For example, if you open a file for reading, the read head will be pointed to the beginning of the file, and if you open a file for append, the read head is pointed to the end of the file.

- **Reading and Writing**: Reading and writing from a file pointer is very similar to that of a file descriptor:

```
size_t fread (void * ptr, size_t size, size_t nitems, FILE * stream); size_t
fwrite(void * ptr, size_t size, size_t nitems, FILE * stream);
```

Like before data is read from or written to a buffer, ptr, but the amount is described as nitems each of size length. This is very useful when reading chunks of data, but you can always set size to 1 and nitems to the number of bytes you wish to read and write, i.e., read nitems each 1 byte in size.

- **Converting to File Descriptor**: Finally, I should note that you can always convert a file pointer to a file descriptor and vice versa using either fdopen() and fileno().

Socket Programming API

You will be using standard stream sockets, TCP connections, and not raw sockets.

- **Opening a socket for streaming:** To open a stream socket, you will use the following function call.

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

socket() takes a socket domain, e.g., internet socket AF INET, the type of the socket, SOCK STREAM indicate a stream session or TCP, and a protocol, which is 0 for SOCK STREAM over IP. socket() returns a socket file descriptor, which is just an integer.

- **Connecting a socket:** In previous labs, you were using connection-less sockets where you just send data to a particularly destination without establishing a connection *a priori*. With stream sockets you must first connect with the destination before you can begin transmitting data. To connect a socket, use the connect() system call:

```
int connect(int socket, cost struct sockaddr *address, socklen_t address_len)
```

which takes a socket file descriptor, a pointer to a socket address, and the length of that address. The return value of connect() indicates a succesful or failed connection: You should refer to the manual for more details on error conditions.

Regarding socket addressing: you'll probably want to use the sockaddr in form of sockaddr, which is the same size and has the following structure members:

```
struct sockaddr_in{
        u_char  sin_len;
        u_char  sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

And an in addr_is a uint32, or a 32 bit number, like an int. You can convert string representation of internet addresses to in addr's using either getaddrinfo() or gethostbyname(). Sample code is available on the message board.

Note that if your computer has mutltiple interface, you must first bind() your socket to one of the interfaces using the bind() system call. It has the following function definition:

```
int bind(int socket, struct sockaddr *address, socklen_t address_len)
```

where the address indicates which of the servers interfaces, identified by IP address, this socket should use for listening (or sending).

- **Listening and Accepting a Connection:** On the server end, a socket must be set such that it can accept incoming connections. This occurs in two parts, first it requires a call to listen(), and second, a call to accept(). The function definition for listen() is as follows:

```
int listen(int socket, int backlog)
```

Of course, listen() first argument is the socket that the will be listened on. The second argument, backlog indicates the number of *queued* or *backlogged* incoming connections that can be pending waiting on an accept() call before a connection refused message is sent to the connecting client.

The accept() function is the key server side mechanism of socket programming. Let's start by inspecting its function definition:

```
int accept(int socket, struct sockaddr * address, socklent_t * address_len)
```

Essentially, given a socket that is listening to incoming connections, accept **will block** until a client connects, filling in the address of the client in address and the length of the address in address len.

The return value of accept is very important: It returns a *new socket file descriptor* for the newly accepted connection. The information about this socket is encoded in address, and all further communication with this client occurs over the new socket file descriptor. Don't forget to close the socket when done communicating with the client.

- **Reading and Writing:** Reading and writing from an stream socket occurs very much like reading and writing from any standard file descriptor. There are a number of functions to choose from, I suggest that you use the standard read() and write() system calls. Writing is rather simple:

```
ssize_t write(int filedes, void * buf, size_t nbyte)
```

which, given a file descriptor (the socket) and a buffer buf, write() will write nbyte's to the destination described in the file descriptor and return the number of bytes written.

Reading from a socket is slightly more complicated because you cannot be certain how much data is going to be sent ahead of time. First consider the function definition of read():

```
ssize_t read(int filedes, void * buf, size_t nbyte)
```

Similar to write(), it will read from the give file descriptor (the socket), and place up to nbyte into the buffer pointed to by buf, returing the number of bytes read. However, consider the case where the remote side of the socket has written more than the buffer size of bytes. In such cases, you must place the read() in a loop to clear the line. Here is some sample code that does that:

```
while(read(sockfd, buf, BUF_SIZE)){

 //do something with data read so far

}
```

That is, the loop will continue until the amount read is zero, which indicates that there is no more data to read. But be careful, subsequent calls to read() when there is no data on the line will block until there is something to read.

- **Closing a Socket:** To close a socket, you simply use the standard file descriptor close() function:

```
int close(int filedes)
```

Putting it all together

Below are examples in skeletal psuedocode for a server and a client :

Client:

```
int sockfd;
struct socckaddr_in sockaddr;
char  data[BUF_SIZE];

//open the socket
sockfd = socket(AF_INET, SOCK_STREAM, 0);

//set socket address and connect
if( connect(sockfd, &sockaddr, sizeof(sturct ockaddr_in) < 0){
perror("connect");
exit(1);
}

//send data
write(sockfd, data, BUF_SIZE);

//close the socket
close(sockfd);
```

Server:

```
int serve_sock, client_sock, client_addr_size;
struct sockaddr_in serv_addr, client_addr; char
data[BUF_SIZE];

//open the socket
sockfd = socket(AF_INET, SOCK_STREAM, 0);

//optionally bind() the sock
bind(sockfd, serv_addr, sizeof(struct sockaddr_in));

//set listen to up to 5 queued connections
listen(sockfd, 5);

//could put the accept procedure in a loop to handle multiple clients

//accept a client connection
client_sock = accept(sockfd, &client_addr, &cleint_addr_len);

while(read(client_sock, data, BUF_SIZE)){
//Do something with data
//close the connection
close(client_sock);

}
```

Lab Submission

1. Zip or tar folder and put it in the W drive. Name your folder: <lastname<1st name>_lab1.zip.
2. Email me the folder as an attachment.