CS 352 Spring 2015
Programming Project Part 3

1. Overview:

For part 2 of the project, your team will continue to improve the socket library you have developed in part 2. For part, you must make your socket library work as part of an client and server that use encryption. The new client and server, called client_crypto and server_crypto, work by having the client send an HTTP-style string requesting a file from the server. However, after the server sends a nonce back to the client, all traffic is encrypted. For this part of the project, the client and server use the Sodium library, which provides simple, portable cryptographic functions in C. More information about the sodium library can be found at: http://www.libsodium.org/

As part of the project part 1 and part 2, you will be given a number of files. You can also find them in the sakai site under "Resources" -> "Project resources" -> "Part 3" .

For part 3 of the project, you will only need to make a *single connection* work over a *single port* for a *single thread* in a *bi-directional manner.* The goal is to correctly implement a go-back-N protocol for one connection, for example, when sending a single file between a client and server.

1. `Makefile` : you must use a makefile that creates the client and server programs. You can extend this makefile but “make all”, “make client”, “make server” and “make clean” must operate as defined in this file. Your library should be called: sock352lib.o. If you want to extend the makefile, you can create a .a or .so file, but make sure the make commands end up building a client and server.

2. **`sock352.h`** : You may not alter this file. The file defines the interface for the cs 352 socket library that you will implement.

3. **`client_crypto.c`** : This is the cryptographic client source code. file. You may not alter the code for this file. It must compile and link against your code.

4. **`server_crypto.c`** : This is the cryptographic server file You may not alter the code for this file. It must compile and link against your code.

5. **`client.c, client2.c`**: These are the old clients for part 1 and part 2, which sends a request for a file from a server.

6. **`server.c,server2.c:`** These are the old servers for part 1 and part 2. It waits for a file-name to be sent by client2, and then responds with the binary of the file. If there is an error obtaining a file, a zero file-length is returned.

7-12: **`client, client2, client_crypto, server, server2, server_crypto:`** These are binaries you can run to test your client and server. These binaries have been built with the instructors/TA's version of a library that implements the sock352 protocol. Your client should run with this binaries server, and your server should be able to run against this client.

Your library must implement the following functions as defined in the `sock352.h` file:

```
int sock352_init(int udp_port);
int sock352_init2(int remote_UDP_port,int local_UDP_port);
int sock352_init3(int remote_UDP_port,int local_UDP_port, char *environment_p[]);
int sock352_socket(int domain, int type, int protocol);
int sock352_bind(int fd, sockaddr_sock352_t *addr, socklen_t len);
int sock352_connect(int fd, sockaddr_sock352_t *addr, socklen_t len);
int sock352_listen(int fd, int n);
int sock352_accept(int _fd, sockaddr_sock352_t *addr, int *len);
int sock352_close(int fd);
int sock352_read(int fd, void *buf, int count);
int sock352_write(int fd, void *buf, int count);
```

These function map to the existing C-library functions for sockets. See chapter 4, pages 95-120 of Stevens et. al. for the definitions of these functions. The one exception is the `sock352_initX` family of calls. The init calls take two kinds of parameters. The first are UDP ports that the rest of the CS 352 RDP library will use for communication between hosts. The second is the environment pointer passed from the C `main()` function. The environment variables are used in the instructor/TA library to control various outputs of the library. In particular, you can set the environment variable SOCK352_DEBUG_LEVEL to 5 to get increased debugging output from the binary client and server programs. Recall to set an environment variable from the command line shell do “export <VARIABLE>=value”

2. The 352 RDP v1 protocol:

Recall as in TCP, 352 RDP v1 maps the abstraction of a logical byte stream onto a model of an unreliable packet network. 352 RDP v1 thus closely follows TCP for the underlying packet protocol. A connection has 3 phases: Set-up, data transfer, and termination. 352 RDP v1 uses a much simpler timeout strategy than TCP for handling lost packets.

Packet structure:

The CS 352 RDP v1 packet as defined as a C-structure, as can be found in the sock352.h file:

```
/* a CS 352 RDP protocol packet header */
struct __attribute__ ((__packed__)) sock352_pkt_hdr {
      uint8_t version;         /* version number */
      uint8_t flags;           /* for connection set up, tear-down, control */
      uint8_t opt_ptr;         /* option type between the header and payload */
      uint8_t protocol;        /* higher-level protocol */
      uint16_t header_len;     /* length of the header */
      uint16_t checksum;       /* checksum of the packet */
      uint32_t source_port;    /* source port */
      uint32_t dest_port;      /* destination port */
      uint64_t sequence_no;    /* sequence number */
      uint64_t ack_no;         /* acknowledgement number */
      uint32_t window;         /* receiver advertised window in bytes*/
      uint32_t payload_len;    /* length of the payload */
};
```

```
typedef struct sock352_pkt_hdr sock352_pkt_hdr_t; /* typedef shortcut */
```

Note that uintX_t is an X-bit unsigned integer., as defined in `<sys/types.h>`. At the packet level, all these fields are defined to be in network byte-order (big-endian, most significant byte first).

For part 3, in the packet, the `version` field should be set of 0x1. The `protocol`, `opt_ptr`, `source_port` and `dest_port` fields should all be set to zero. Future versions of the protocol will add port spaces and options. The `header_len` field will always be set to the size of the header, i.e., `sizeof(sock352_pkt_hdr_t)`.

An address for the CS 352 RDP is slightly different from the normal socket address, as found in the sock352.h file. The main difference is the addition of a port layer on top of the UDP port space, as seen in the `cs352_port` field. This will be used in later versions of the protocol.

```
/* Structure describing a CS 352 socket address.  */
struct sockaddr_sock352 {
      __SOCKADDR_COMMON (sin_);
      uint32_t cs352_port; /* CS 352 socket port number */
      in_port_t sin_port;  /* UDP Port number.  */
      struct in_addr sin_addr; /* Internet address.  */

      /* Pad to size of `struct sockaddr'.  */
      unsigned char sin_zero[sizeof(struct sockaddr) -
      __SOCKADDR_COMMON_SIZE - sizeof(uint32_t) - sizeof(in_port_t)
                  - sizeof(struct in_addr)];
};
typedef struct sockaddr_sock352 sockaddr_sock352_t;  /* add type shortcut */
```

Connection Set-up:
352 RDP follows the same connection management protocol as TCP. See Chapter 3.5.6, pages 252-258 of Kurose and Ross for a more detailed description. The bit flags needed are set in the `flags` field of the packet header. The exact bit definitions of the flags are defined in the sock352.h file.

The client initiates a connection by sending a packet with the SYN bit set in the `flags` field, picking a random sequence number, and setting the `sequence_no` field to this number. If no connection is currently open, the server responds with both the SYN and ACK bits set, picks a random number for it's `sequence_no` field and sets the ack_no field to the client's incoming `sequence_no+1`. If there is an existing connection, the server responds with the `sequence_no+1`, but the RST flag set.

Data exchange:
352 RDP follows a simplified Go-Back-N protocol for data exchange, as described in section Kurose and Ross., Chapter 3.4.3, pages 218-223 and extended to TCP style byte streams as described in Chapter 3.5.2, pages 233-238.

When the client sends data, if it is larger than the maximum UDP packet size (64K bytes, minus the size of the sock352 header), it is first broken up into segments, that is, parts of the application byte-stream, of up to 64K. If the client makes a call smaller than 64K, then the data is sent in a single UDP packet of that size, with the `payload_len` field set appropriately. Segments are acknowledged as the last segment received in-order (that is, go-back-N). Data is delivered to the higher level application in-order based on the `read()` calls made. If insufficient data exists for a `read()` call, partial data can be returned and the number of bytes set in the call's return value.

Not that just like TCP, the ACK field is set for each data packet.

For CS 352 RDP version 1, for part 2 the client and server can ignore the window field. In this case, the window can be ignored.

Timeouts and retransmissions:
352 RDP v1 uses a single timer model of timeouts and re-transmission, similar to TCP in that there should be a *single timer per connection*, although each segment has a logical *timeout.* The timeout for a segment is 0.2 seconds. That is, if a packet has not been acknowledged after 0.2 seconds it should be re-transmitted, and the logical timeout would be set again set to 0.2 seconds in the future for that segment. The timeout used for a connection should be the timeout of the oldest segment.

There are two strategies for implementing timeouts. One approaches uses Unix signals and other uses a separate thread. These will be covered in class and recitation.

Connection termination:
Connection termination will follow a similar algorithm as TCP, although simpified. In this model, each side closes it's send side separtately, see pages 255-256 of Kurose and Ross and pages 39-40 of Stevens. In version 1, it is OK for the client to end the connection with a FIN bit set when it both gets the last ACK and `close` has been called. That is, `close` cannot terminate until the last ACK is received from the server. The sever can terminate the connection under the same confitions.

If the socket receives an FIN from the other side, and it's data buffer is empty, the socket can be closed after a timeout of 5 seconds.

3. Grading:

Functionality: 80%
Style: 20%

Functionality:
We will run the client programs linked to our library (called the 'course clients)' against the server programs linked against your library (the 'student servers'), and the clients linked to your library (the 'student clients') against the server source code linked to our library ('course servers'). We will send a file and see if the checksum on the client and server match the correct checksums. See the source code for more details. The size of the file may range from a few bytes to many megabytes. There will be a total of 4 tests, as below, and each test is worth 16% of the total grade:

(1) student client2, student server2, in-order packets.
(2) student client_crypto, student server_crypto, in-order packets.
(3) student client_crypto, class server_crypto, in-order packets.
(4) student client_crypto, course server_crypto, random 20% packets dropped by the course library.

Style:

Style points are given by the instructor and TA after reading the code. Style is subjective, but will be graded on a scale from 1-5 where 1 is incomprehensible code and 5 means it is perfectly clear what the programmer intended.

4. What to hand in

You must hand in a single archived file, either zip, tar, gzipped tar, bzipped tar or WinRAR (.zip, .tar, .tgz, .rar) that contains: (1) a Makefile, as in part 1, (2) the client2.c and client_crypto.c source code, (3) the server2.c and server_crypto.c source code, (4) the sock352.h header file, and (4) all the files for your library source code. Your files must build and compile the client2, server2, client_crypto and server_crypto binaries.

Your archive file must include a file called "README.TXT" that includes the names of the project partners for the project!

5. Extra resources

For this project, you will need to keep lists and potentially a hash table. The uthash and utlist libraries are simple, easy to use C-ibraries for these purposes. The documentation for uthash is at https://troydhanson.github.io/uthash/ and for utlist is at https://troydhanson.github.io/uthash/utlist.html If you use these libraries, please include them in the source-code files you turn in.

The sodium security library documentation can be found here: http://download.libsodium.org/doc/

If you wish to use other 3rd party source code in your project, cryptographic you must clear them with the instructor first.