Programming Project Part 2

1. Overview:

For part 2 of the project, your team will continue to improve the socket library you have developed in part 1. For this part, you will add encryption to the messages sent by your socket library. The client and server will give optional arguments to connect and accept methods to specify that encryption should be used. Messages will be encrypted using the NACL (sodium) library's public/private encryption methods. A single nonce will be used for the connection. The socket library will hold a set of keys in a keychain, which is mapping between host,port pairs and public/private keys. Section 2 describes keychains in more detail.

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

1. **`client2.c`** : This is the original client source code. file. You may not alter the code for this file. It must run using your `sock352.py` file.

2. **`server2.c`** : This is the original server file You may not alter the code for this file. It must run using your your sock352.py file.

3. **`sock352.py:`** This is a new library for part 2. You must fill in the methods defined in this file, as below.

4. **`examplenacl.py:`** This files will both generate and print a public/private key pair, as well as demonstrates how to use the sodium (nacl) library.

5. **`Project1Solutions.py:`** This is the solution to project 1.

Your library must implement the following methods as defined in the `sock352.py` file:

```
def init(UDPportTx,UDPportRx):
def readKeyChain(filename):
def __init__(self):
    def bind(self,address):
    def connect(self,*args):
    def listen(self,backlog):
    def accept(self,*args):
    def close(self):
    def send(self,buffer):
    def recv(self,nbytes):
    def readKeyChain(filename):
```

2. Keychains:

For your socket library, each destination host and port address will be assigned a key that is used to encrypt or decrypt a packet. A public key is used for outbound communication, and a private key for

inbound communication. The method `readKeyChain(filename)` will be provided to you for this purpose.

This is an example keychain file:

```
# this is an example keychain file for the CS 352 socket assignment #
lines with a # in the first word are comments
# the keys are labeled by if they are public, private, and the host and
# destination ports for each key.
# a '*' is a wildcard that can be used for all hosts and ports
# private keys are used to decrypt incomming packets and public keys are
# used to encrypt outbound packets
private    *      *   53fbcbb3b76e173f8241408b1f3dd8f9bf0d2a9f84d3db8fee2f38d0f2429729
public localhost 8888 78c7227cf5c637fbc2066070b6fa2662b5c94bbac6fcb1ba5d77ecd61f718574
public localhost 9999 78c7227cf5c637fbc2066070b6fa2662b5c94bbac6fcb1ba5d77ecd61f718574
```

3. 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:

< ----------32 Bits ---------- >			
Version	Flags	Option	Protocol
Header Length		Packet Checksum	
Source Port			
Destination Port			
Sequence Number			
Acknowledgement Number			
Receiver's Window			
Payload Length			

The flags field is defined as:

< ----------8 Bits ---------- >							
			Has Option	RESET	ACK	FIN	SYN

Connection Set Up:

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.

Connection additions for part 2: Encryption

In order to specify that a given connection is encrypted, additional arguments are given to the socket's `connect()`and `accept()` methods. These are constants passed into the methods using Python's variable arguments structure:

```
connect((host,port), sock352.ENCRYPT) accept(sock352.ENCRYPT)
```

The method signature to us variable arguments in Python is:

```
def connect(self, *args): if
     (len(args) >= 1):
          # do something elif
     (len(args) >= 2):
          #  check constant, add encryption
```

During connection set-up these function will consult the global variables `publicKeys` and `global privateKeys` to see if there is a matching host and port. These tables are populated in the `readKeyChain()`method. These hash tables store the key as a `(host,port)` pair and the value as the key.

Also during the connection set up, the nonce should be created, the keys found, and the `Box` objection created.

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 bytestream, 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 inorder based on the `recv()` calls made.

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.

Data exchange additions for part 2: Encryption

If the options field in the header is set to binary 01, then the payload of packet will be encrypted. The client will create one nonce to be used for the connection.

Recall that to encrypt data for a connection, the payload only needs to be encrypted using code similar to:

```
encrypted_payload= socket_box.encrypt(payload, nonce)
```

and decryption is similar:

```
plaintext = sock_box.decrypt(encrypted_payload)
```

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 simplified. 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 conditions.

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%
40% - Part 1 working (i.e. no encryption)
40% - Part 2 working (i.e. encryption)

Style: 20%

Functionality:

We will run the client2.py program linked to our library (called the 'course client)' against the server2.py program linked against your library (the 'student server'), and the client2.py linked to your library (the 'student client') against the server2.py linked to our library ('course server'). We will send a file and see if the checksum on the client and server match the correct checksums. The client2.py

program opens a single file, sends to the server, and then both exit. See the source code for more details.

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) README.TXT file with your team members, (2) your sock352.py, and (3) any other files your library needs to work.

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

5. Extra resources

A file called `examplenacl.py` provides a way to generate keys