1) Diffie-Hellman Key Exchange algorithm.

Diffie-Hellman key exchange offers the best of both worlds -- it uses public key techniques to allow the exchange of a private encryption key. Let's take look at how the protocol works, from the perspective of Client and Server, the two users who wish to establish secure communications. We can assume that Client and Server know nothing about each other but are in contact.

Here are the nine steps of the process:

1. Communicating in the clear, Client and Server agree on two large positive integers, p and g.
2. Client randomly chooses another large positive integer, X_A, which is smaller than p. X_A will serve as Client's private key.
3. Server similarly chooses his own private key, X_B.
4. Client computes her public key, Y_A, using the formula $Y_A = (g\text{^}X_A) \bmod p$.
5. Server similarly computes his public key, Y_B, using the formula $Y_B = (g\text{^}X_B) \bmod p$.
6. Client and Server exchange public keys over the insecure circuit.
7. Client computes the shared secret key, k, using the formula $k = (Y_B\ \text{^}X_A) \bmod p$.
8. Server computes the same shared secret key, k, using the formula $k = (Y_A\ \text{^}X_B) \bmod p$.
9. Client and Server communicate using the symmetric algorithm of their choice and the shared secret key, k, which was never transmitted over the insecure circuit.

Functionality of the program:

- Client program ask for input of two number p and g. p is checked for prime number, if yes then it is accepted as is else next prime number is selected as the value of p

```
private static BigInteger getNextPrime(String ans) {
        BigInteger test = new BigInteger(ans);
        while (!test.isProbablePrime(99))
                test = test.add(one);
        return test;
}
```

- Secret key for client and server side is hardcoded in the program.

```
//Client Side
BigInteger a = BigInteger.valueOf(233);

//Server Side
s = BigInteger.valueOf(100);
```

- Client and server compute their public values using the following code

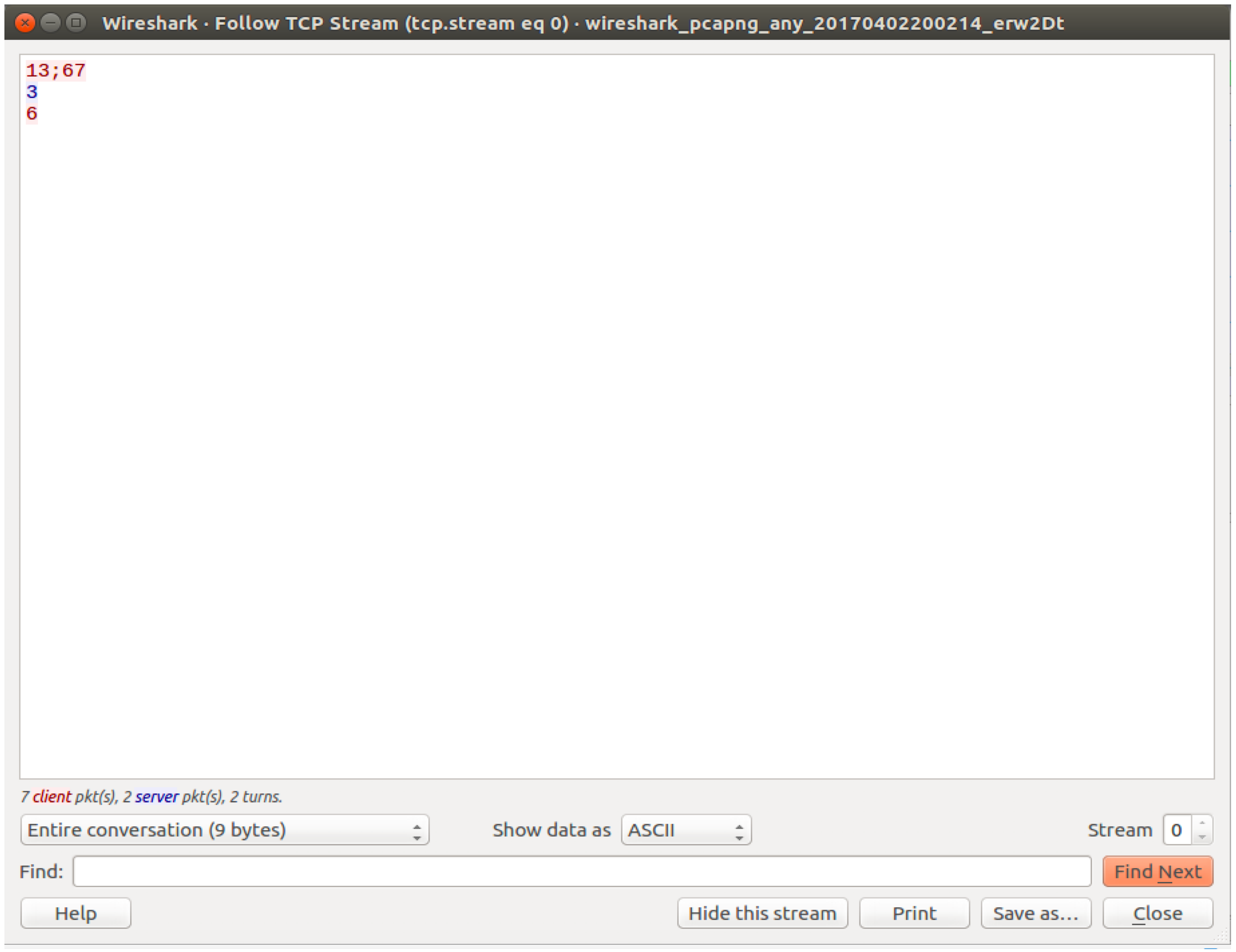
```
//Client side
BigInteger resulta = g.modPow(a, p);
//reslta will be shared with server

//Server side
results = g.modPow(s, p);
//results will be shared with client
```

  later both share their computed values over socket with each other.

- Once client and server receives computed values they will try to compute shared secret which will be used later in interaction for encryption and decryption.

```
//Client side shared secret computation
BigInteger KeyClientCalculates = resultb.modPow(a,p);
//Server side shared secret computation
BigInteger KeyServerCalculates = resulta.modPow(b,p);
```

Data from wireshark:


Wireshark · Follow TCP Stream (tcp.stream eq 0) · wireshark_pcapng_any_20170402200214_erw2Dt

```
13;67
3
6
```

7 client pkt(s), 2 server pkt(s), 2 turns.

First line indicates the values of p and g.
Second line is server computed public key.
Third line is client computed public key.

Computed Shared secret at each end is: 9 which is not exchanged over network.

2) RSA

RSA involves a public key and a private key. The mathematics behind RSA algorithm and code demonstration is as follows:

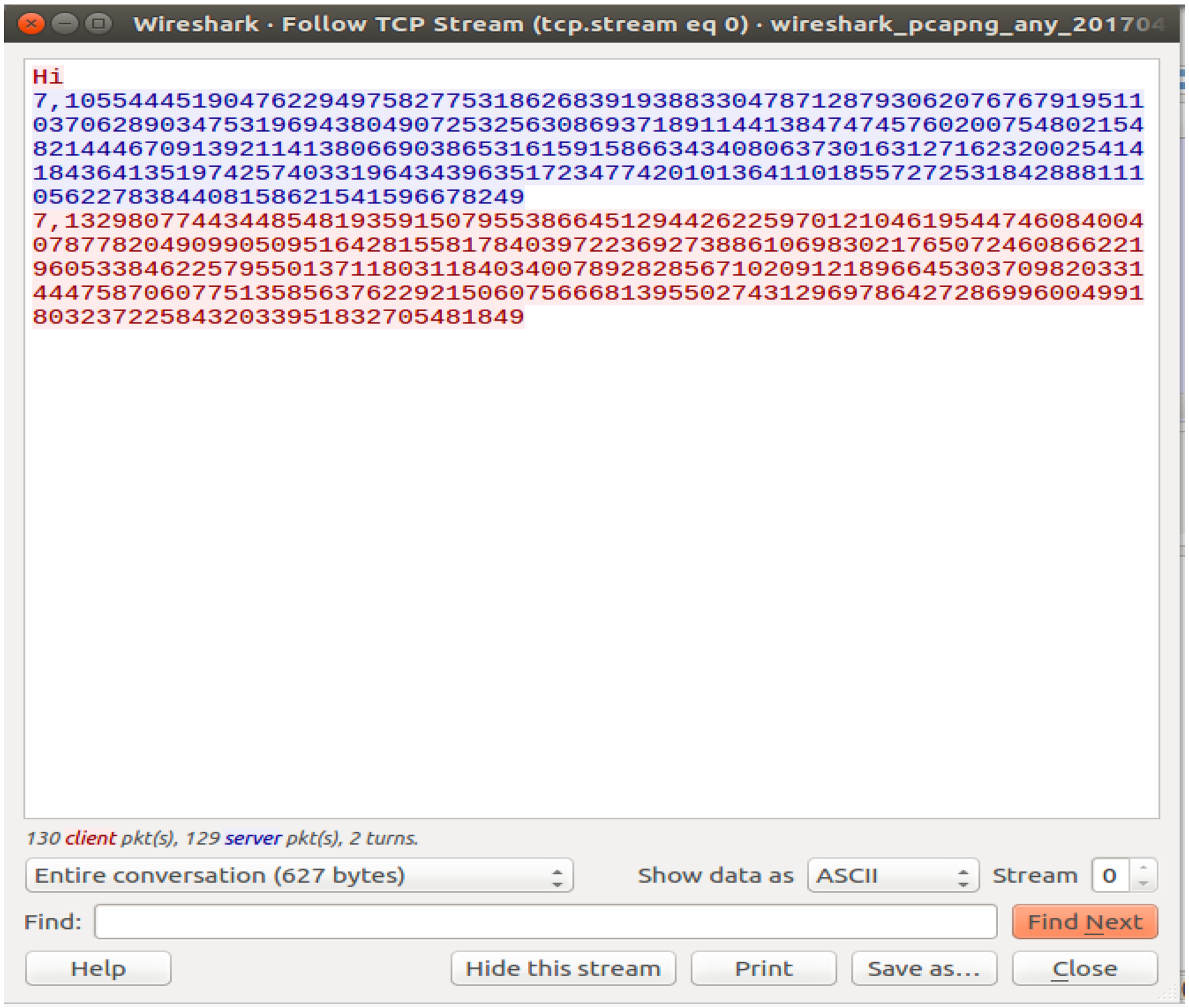
1. Generate 2 large prime number p, q.
2. After step 1 n is computed which is equal to pq.(n = pq)
   This n is used as the modulus for both the public and private keys.
3. Then phi(n) is computed. (phi(n) = (p – 1)(q – 1)).

4. Then e is selected relative to prime number phi(n).
   $1 < e < \text{phi}(n)$
5. D can be determined as
   $e * d \equiv 1 \bmod \varphi(n)$

6. The encryption key (e,n) is made public.
7. The decryption key (d,n) is kept private by the client and server both. Client and server both will have their own pair of public and private key.

```
public RSA(int bits) {
            bitlen = bits;
            SecureRandom r = new SecureRandom();
            BigInteger p = new BigInteger(bitlen / 2, 100, r);
            BigInteger q = new BigInteger(bitlen / 2, 100, r);
            n = p.multiply(q);
            BigInteger m =        (p.subtract(BigInteger.ONE)).multiply(q.subtract(BigInteger.ONE));
            e = new BigInteger("3");
            while (m.gcd(e).intValue() > 1) {
                        e = e.add(new BigInteger("2"));
            }
            d = e.modInverse(m);
 }
```

8. Client and server public key exchange


Wireshark · Follow TCP Stream (tcp.stream eq 0) · wireshark_pcapng_any_20170

```
Hi
7,105544451904762294975827753186268391938833047871287930620767679195110370628903475319694380490725325630869371891144138474745760200754802154821444670913921141380669038653161591586634340806373016312716232002541418436413519742574033196434396351723477420101364110185572725318428881110562278384408158621541596678249
7,132980774434485481935915079553866451294426225970121046195447460840040787782049099050951642815581784039722369273886106983021765072460866221960533846225795501371180311840340078928285671020912189664530370982033144475870607751358563762292150607566681395502743129697864272869960049918032372258432033951832705481849
```

130 client pkt(s), 129 server pkt(s), 2 turns.

Entire conversation (627 bytes)

Show data as ASCII

Stream 0

In the application key exchange always starts with HI followed by exchange of client and server public key with each other.

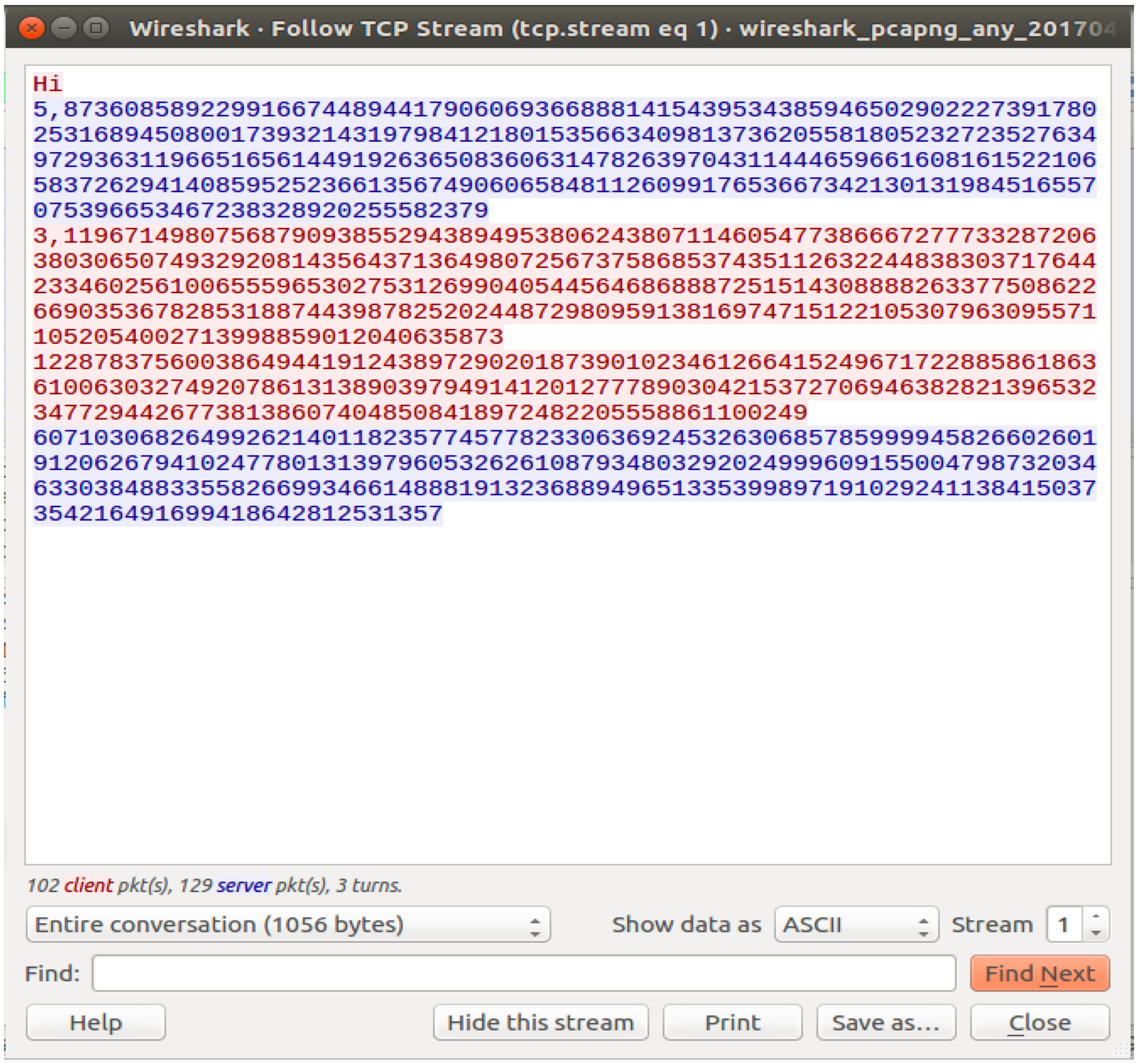
9. The encryption key (e,n) is made public.
   Cipher text can be obtained by c = me mod n

```
public synchronized String encrypt(String message, BigInteger PK, BigInteger    Modu)      {
   return (new BigInteger(message.getBytes())).modPow(PK, Modu).toString();
}
```

10. For decryption key (d,n) is kept as secret by the user.
    Plain text can be obtained by m = cd mod n

```
public synchronized String decrypt(String message, BigInteger PrivatePK, BigInteger Modulu) {
    return new String((new BigInteger(message)).modPow(PrivatePK, Modulu).toByteArray());
}
```

11. WireShark captured packets for text “Network Security”.



Wireshark · Follow TCP Stream (tcp.stream eq 1) · wireshark_pcapng_any_20170

```
Hi
5,87360858922991667448944179060693668881415439534385946502902227391780
25316894508001739321431979841218015356634098137362055818052327235276 34
97293631196651656144919263650836063147826397043114446596616081615221 06
58372629414085952523661356749060658481126099176536673421301319845165 57
0753966534672383289202555823 79
3,11967149807568790938552943894953806243807114605477386667277733287206
38030650749329208143564371364980725673758685374351126322448383037176 44
23346025610065559653027531269904054456468688872515143088882633775086 22
66903536782853188744398782520244872980959138169747151221053079630955 71
1052054002713998859012040635873
12287837560038649441912438972902018739010234612664152496717228858618 63
61006303274920786131389039794914120127778903042153727069463828213965 32
3477294426773813860740485084189724822055588611002 49
60710306826499262140118235774577823306369245326306857859999458266026 01
91206267941024778013139796053262610879348032920249996091550047987320 34
63303848833558266993466148881913236889496513353998971910292411384150 37
354216491699418642812531357
```

102 client pkt(s), 129 server pkt(s), 3 turns.

In the above captured packet conversation is initiated by client saying “Hi” to server, in response to Hi server acknowledges client with its public key and client exchanges its public key with server as well.

Later client sends the encrypted message (“Network Security”) using server public key and server decrypts the message using its private key. If decryption is successful server replies to the client with encrypted message (“I got your reply and here is mine”) using client public key.

```
12.     Total time to encrypt roughly 10000 words took
        142.078724306 Seconds.

13.     Total time to decrypt roughly 10000 words took
        385.599872839 Seconds.
```

3) DES:
Plaintext, is encrypted with an encryption algorithm and an encryption key. The process results in cipher text, which only can be viewed in its original form if it is decrypted with the correct key.

Symmetric-key ciphers use the same secret key for encrypting and decrypting a message or file. While symmetric-key encryption is much faster than asymmetric encryption, the sender must exchange the encryption key with the recipient before he can decrypt it (it can be coupled up with **Diffie-Hellman** key exchange algorithm to establish the secret among them).

In our application key is pre-established secret between client and server.

1) Key is derived from a string and saved in a text file (in encoded way) which is accessible to client and server.
2) Both can read the key from file to encrypt or decrypt the data.
3) Client derives the key and writes the key into the file and save it, which can be used by server to decrypt the data.

```
password = "Passcode";
byte[] salt = new byte[64];
Random rnd = new Random();
rnd.nextBytes(salt);
byte[] data = deriveKey(password, salt, 64);

// BufferedReader inFromServer = new BufferedReader(new
// InputStreamReader(clientSocket.getInputStream()));
System.out.println("Enter the Data to be transmisted to server\n");
sentence = inFromUser.readLine().getBytes();
SecretKey desKey = SecretKeyFactory.getInstance("DES").generateSecret(new DESKeySpec(data));
System.out.println("Secret key " + desKey.toString());
Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding");

FileOutputStream writeStream = new FileOutputStream(new File("key.txt"));
byte[] kb = desKey.getEncoded();
writeStream.write(kb);
writeStream.close();
```

```
public static byte[] deriveKey(String password, byte[] salt, int keyLen) {
	SecretKeyFactory kf = null;
	try {
		kf = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
	} catch (NoSuchAlgorithmException e) {
		// TODO Auto-generated catch block
		e.printStackTrace();
	}
	KeySpec specs = new PBEKeySpec(password.toCharArray(), salt, 1024, keyLen);
	SecretKey key = null;
	try {
		key = kf.generateSecret(specs);
	} catch (InvalidKeySpecException e) {
		// TODO Auto-generated catch block
		e.printStackTrace();
	}
	return key.getEncoded();
}
```

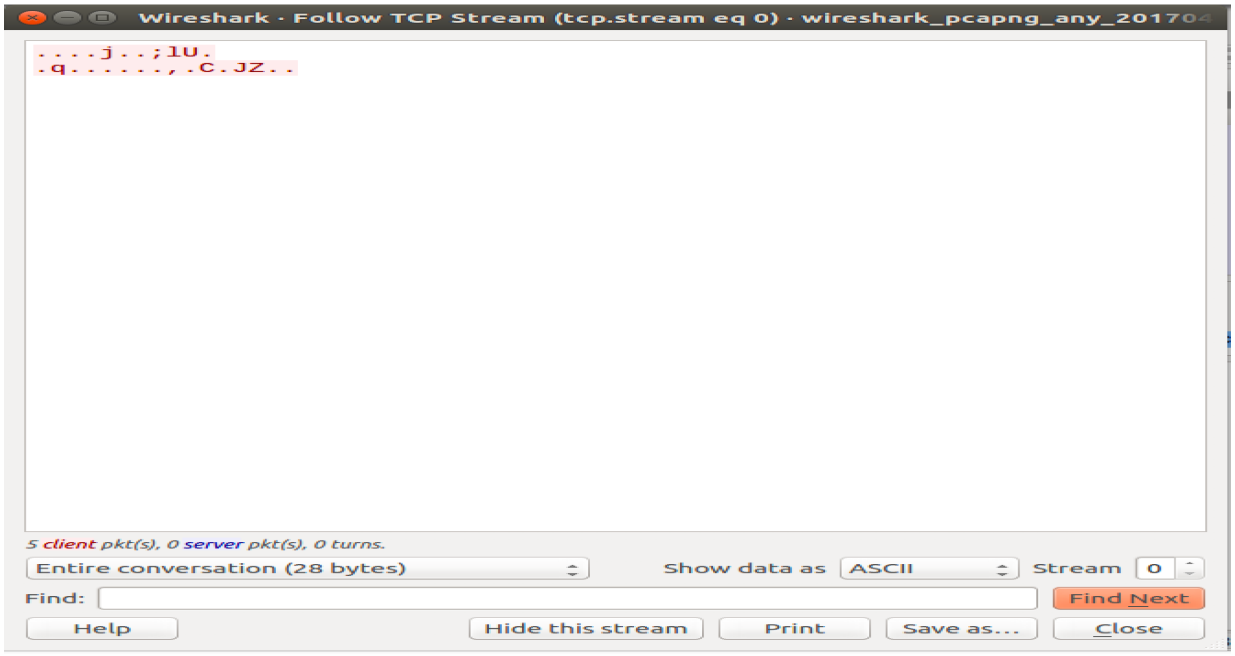
4) Data encryption and sending it to server

```
	cipher.init(Cipher.ENCRYPT_MODE, desKey);
	textEncrypted = cipher.doFinal(sentence);
	outToServer.writeInt(textEncrypted.length);
	outToServer.write(textEncrypted);
```

5) Key reading from a file and data decryption at server side

```
	FileInputStream readFile = new FileInputStream(new File("key.txt"));
	int len1 = readFile.available();
	byte[] data = new byte[len1];
	readFile.read(data);
	KeySpec ks = null;
	SecretKey ky = null;
	SecretKeyFactory kf = null;
	ks = new DESKeySpec(data);
	kf = SecretKeyFactory.getInstance("DES");
	ky = kf.generateSecret(ks);
	SecretKey originalKey = new SecretKeySpec(data, 0, data.length, "DES");
	System.out.println("Read KEy is" + ky);
	Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
	cipher.init(Cipher.DECRYPT_MODE, ky);
		// Decrypt the text
	byte[] textDecrypted = null;
	if (len > 0) {
			byte[] message = new byte[len];
			inFromClient.readFully(message, 0, message.length);
			System.out.println("Text Received " + message);
			textDecrypted = cipher.doFinal(message);
	}
	System.out.println("Text Decryted : " + new String(textDecrypted));
```

6) WireShark captured packet when client sends server message as (“Network Security”).



Wireshark · Follow TCP Stream (tcp.stream eq 0) · wireshark_pcapng_any_20170

```
....j..;1U.
.q.......,.C.JZ..
```

5 client pkt(s), 0 server pkt(s), 0 turns.

Entire conversation (28 bytes) Show data as ASCII Stream 0

7) Time taken to Encrypt 10000 words at client side
0.321017078 seconds.

Time taken to decrypt 10000 word at server side
21.075552073 seconds.

4) **HMAC**:

HMAC is a computed "signature" often sent along with some data. The HMAC is used to verify (authenticate) that the data has not been altered or replaced.

1) At the client side message is hashed with the key and return a fixed size hex string which can be sent along with message to server.

String calculatedHash = calculateHMAC(sentence, "message integrity");

CalculateHMAC is used to get the fixed length hex string which can be sent to server.

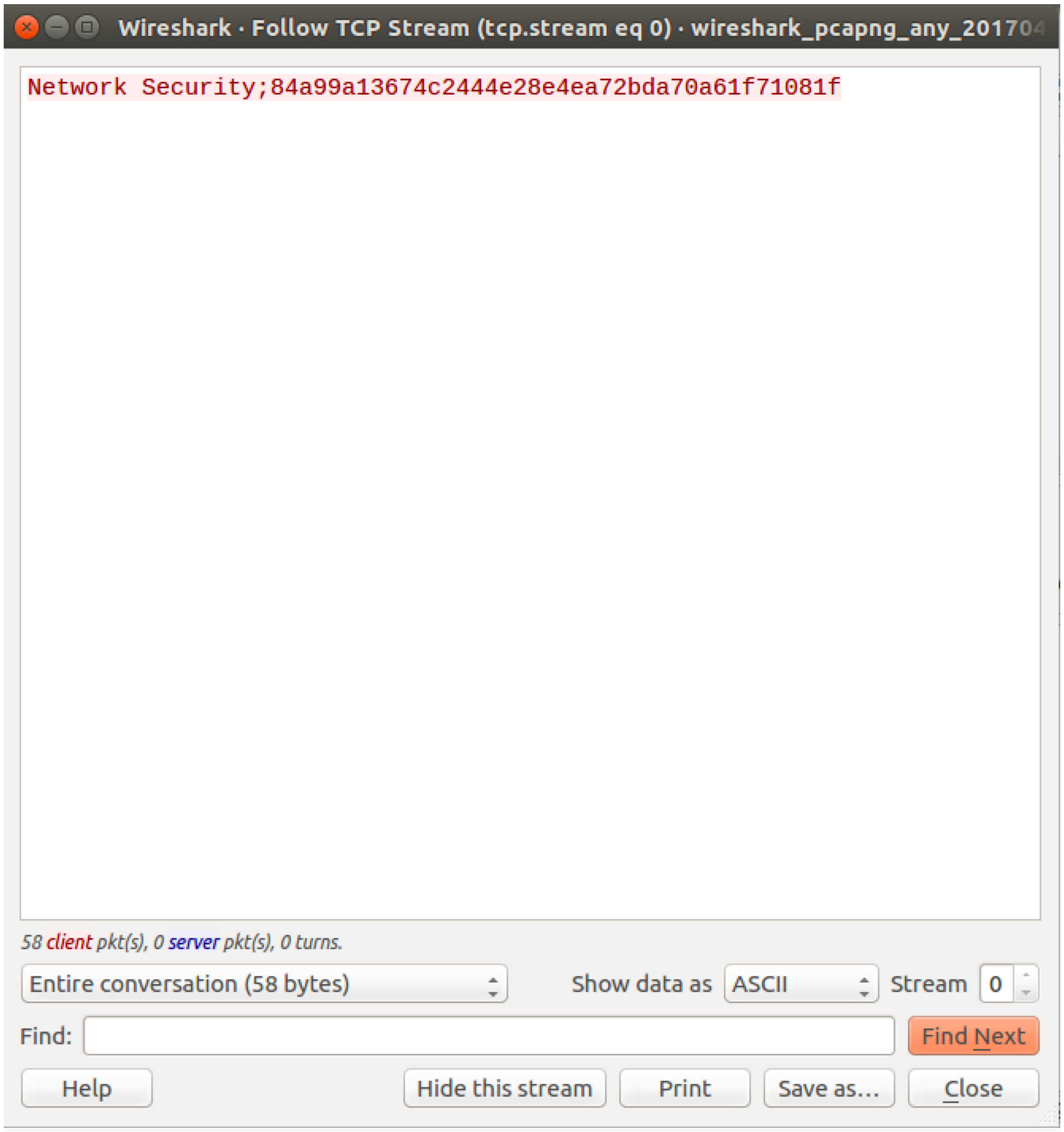
```
public static String calculateHMAC(String message, String key)
                                    throws NoSuchAlgorithmException, InvalidKeyException {
          SecretKeySpec singingkey = new SecretKeySpec(key.getBytes(),
        HMAC_SHA1_ALGORITHM);
        Mac Hmac = Mac.getInstance(HMAC_SHA1_ALGORITHM);
        Hmac.init(singingkey);
        return toHexString(Hmac.doFinal(message.getBytes()));
}

@SuppressWarnings("resource")
private static String toHexString(byte[] bytes) {
        Formatter formatter = new Formatter();
        for (byte b : bytes) {
                formatter.format("%02x", b);
        }
        return formatter.toString();
}
```

2) At the server side server recalculates the HMAC on the message and if it matches to the HMAC sent by client message is accepted else message is rejected.

```
String calculatedHash = calculateHMAC(message, "message
integrity");
if (!hmacForMessage.equals(calculatedHash)) {
      break;
}
```

CalculateHMAC has similar implementation as at client side.

3) Captured packet between client and server with HMAC.

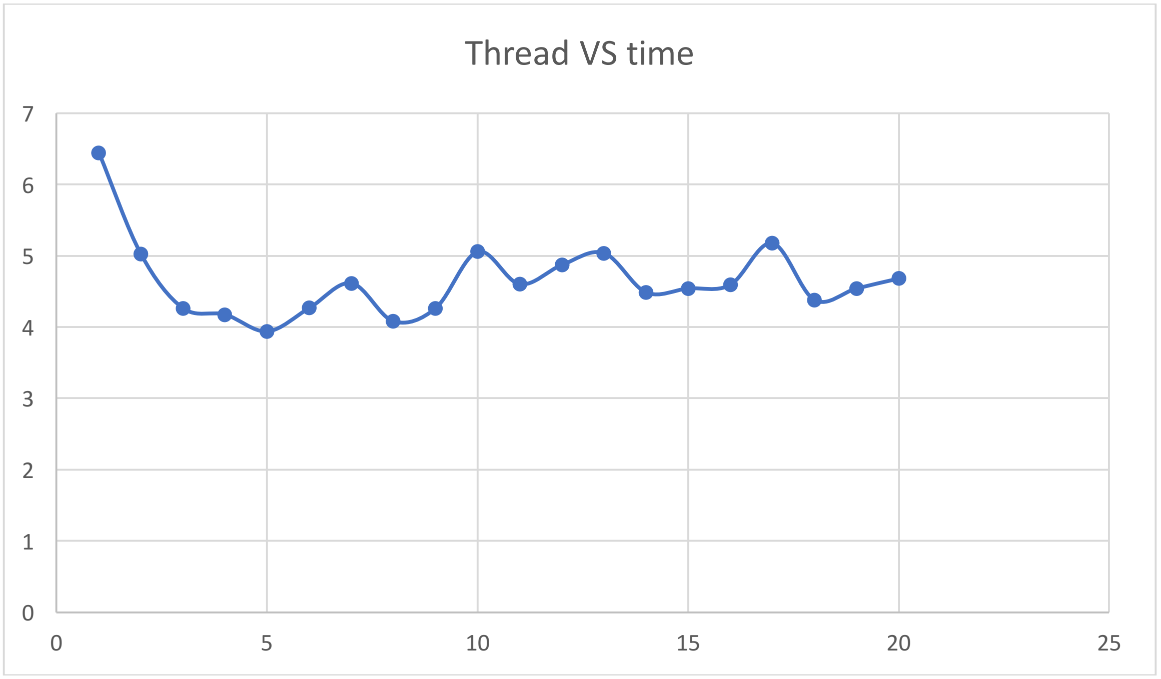

Wireshark · Follow TCP Stream (tcp.stream eq 0) · wireshark_pcapng_any_20170
Entire conversation (58 bytes)
Show data as ASCII
Stream 0
Find:
Find Next
Help
Hide this stream
Print
Save as…
Close

```
Network Security;84a99a13674c2444e28e4ea72bda70a61f71081f
```

58 client pkt(s), 0 server pkt(s), 0 turns.

4) Total time taken for HMAC calculation for 10000 words on client side is
   4.359939439
   Total time taken for HMAC calculation for 10000 words on server side is
   5.51370702

5) Brute Forcing DES:



Assume the scale up to be S64 with 64 threads. Suppose for 1 thread and 20-bits key, the running time is T1; for 64 threads and 26-bits key, the running time is T64 = 177.984 sec.

Which means roughly 377,050 keys/sec are tried by 64 thread (5891.4 key/sec by each thread). Further we can say that roughly 2^18 keys are tried per second.

So, which means time needed to break DES = 2^56/2^18 = 2^38 seconds

Roughly 8716.32 years will be needed to break DES on my system configuration considering 64 threads will be working together.

Utility Explanation

1) Application asks for number of threads and keysize in bits as input.

```
private static void parseArgs(String[] args) {
        if (args.length != 2) {
                System.err.println("Usage: BruteForceDES #threads key_size_in_bits");
        System.exit(0);
}
try {
        numThreads = Integer.parseInt(args[0]);
```

```
            numKeyBits = Long.parseLong(args[1]);
} catch (NumberFormatException e) {
            System.err.println("Usage: BruteForceDES #threads key_size_in_bits");
}
maxkey = ~(0L);
maxkey = maxkey >>> (64 - numKeyBits);
key = new Random().nextLong();
key = key & maxkey;
}

private static void initDecryptThreads() {
  long numKeysPerThread = maxkey / numThreads;
  long numRemainKeys = maxkey % numThreads;
  decryptThreads = new DecryptThread[numThreads];
  threads = new Thread[numThreads];
  long currentKey = 0;
  for (int i = 0; i < numThreads; i++) {
            long num = numKeysPerThread;
            if (numRemainKeys > 0) {
                      num++;
                      numRemainKeys--;
  }
  decryptThreads[i] = new DecryptThread(getSealedText(text), i, currentKey, currentKey + num);
  currentKey += num;
  threads[i] = new Thread(decryptThreads[i]);
}
}
```

2) Currently “Plain text” is hard coded in the application.
3) Application first encrypt the plain text and prints out the random key which was used in encryption
4) Based on number of thread and keysize, each thread is assigned with blocks of keys to test on and decrypt the text.
5) Later each thread calls for dycrypt() and try to recover the plain text.

```
    public static void main(String[] args) {
          BruteForceDES.text = "Network Security?";
          parseArgs(args);
          startTime = System.currentTimeMillis();
          initDecryptThreads();
          decrypt();
    }
   public void run() {
        // System.out.println("Startkey: " + startKey + "EndKey: " + endKey);
        System.out.println("Thread " + threadId + " StartKey: " + startKey + " EndKey: " + endKey);
       for (long i = startKey; i < endKey; i++) {
          this.deccipher.setKey(i);
          String decryptstr = this.deccipher.decrypt(sealedObj);
                    if (decryptstr != null) {
                              System.out.println(
                                        "Thread " + threadId + " found decrypt key " + i +
                                         " producing message: " + decryptstr);
```

```
                        }
                }
              }


             //Decrypt function call
            public String decrypt(SealedObject cipherObj) {
                try {
                        return (String) cipherObj.getObject(the_key);
                } catch (Exception e) {
                        // System.out.println("Failed to decrypt message. " + ". Exception:
                        // " + e.toString() + ". Message: " + e.getMessage()) ;
                }
                return null;
             }

            //Encrypt Function
             public SealedObject encrypt(String plainstr) {
                try {
                        des_cipher.init(Cipher.ENCRYPT_MODE, the_key);
                        return new SealedObject(plainstr, des_cipher);
                } catch (Exception e) {
                        System.out.println("Failed to encrypt message. " + plainstr + ". Exception: " +
e.toString() + ". Message: "
                                        + e.getMessage());
                }
                return null;
        }
```

6) Once plain text is recovered thread prints out its ID, key and plain text.
7) In this way, this can be verified both key is same for encryption and decryption.

6) Brute Force RSA:

For factorizing I used following code:

```
public static String factorise(long l)
{
  double a = Math.floor(Math.sqrt(l));
  while(l/a != Math.round(l/a))
  {
    a--;
  }
  return (long)a + ", " + (long)(l/a);
}
```

for 45 bit the PC approximately took 513 second for factorization.

So, from the above we can derive that
If 45 bits took 513s, then 1024 bits will take approx. 2^1024 / 2^45 = 2^979 second to decrypt the entire algorithm.

<u>Self-evaluation report on project</u>	
Name	Effort (in percentage)
Vaibhav Mishra	100
Prashanth Gopi	0