THE UNIVERSITY OF DAR ES SALAAM

COICT

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

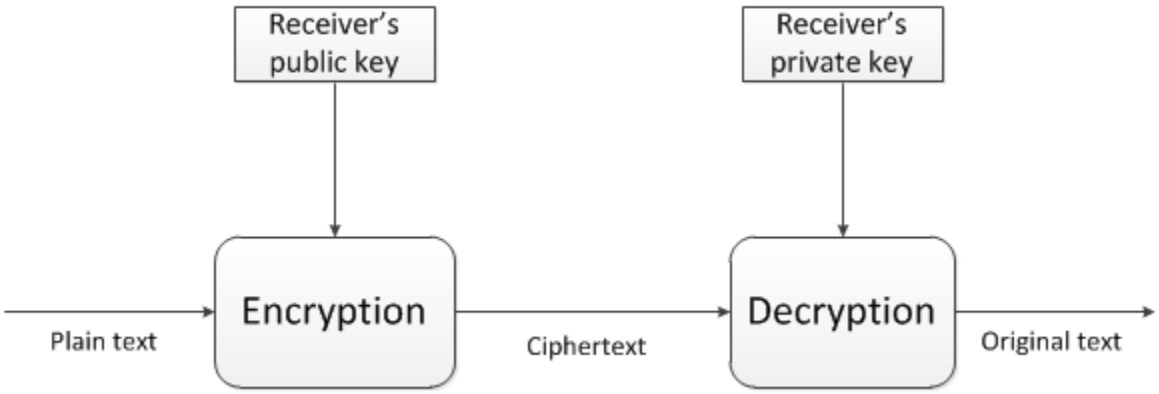
	NAMES	REG NUMBER
1	MREMI, DENIS ISACK	2013-04-02550
2	CHACHA, DORIS DENIS	2013-04-02439
3	SECHA ALBERT C	2013-04-02475
4	MTUI, GODLISTEN R	2012-04-02523
5	MWILILIZA, PETER SIMON	2013-04-02521
6	MUSA FRANK	2013-04-02527
7	FREDRICK, BONIPHACE	2013-04-06732
8	NSULE SHANNON S	2013-04-06882
9	RAHIMU HAMIS SHAMTE	2013-04-07020
10	MGONJA ALEX	2013-04-02536
11	MGENI , KAREBU	2013-04-02541

Qn 2.Security mechanism for Wakwetu

a. Steps for using the asymmetric algorithm encrypting and decrypting messages

   i. We are using RSA with RSA/ECB/PKCS1Padding and UTF-8 encoding.
   ii. Generate the public and private key using class GeneratePrivatePublicMain Where you will specify the location where you want private and public key you wish to be stored.
   iii. Then we use the generated key pairs to encrypt and decrypt messages respectively
   iv. Using class RSACipherTest will use to encrypt and decrypted by passing different parameters of the methods we have supplied.
   v. On sending the message we encrypt it using public key of recipient and sender private key
   vi. On receiving the receiver will decrypt the message using his private key
   vii. The process was successful

b. Flow chart



c. Implementation for asymmetric algorithm

```
package rsa;
/**
 *
 * Program for generating key pairs(Private and Public Key)
 */
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.security.GeneralSecurityException;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;

public final class RSAKeyPair {

   private int keyLength;

   private PrivateKey privateKey;

   private PublicKey publicKey;

   public RSAKeyPair(int keyLength)
         throws GeneralSecurityException {

      this.keyLength = keyLength;
      KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
      keyPairGenerator.initialize(this.keyLength);
      KeyPair keyPair = keyPairGenerator.generateKeyPair();
      privateKey = keyPair.getPrivate();
      publicKey = keyPair.getPublic();
   }

   public final PrivateKey getPrivateKey() {
      return privateKey;
   }

   public final PublicKey getPublicKey() {
      return publicKey;
   }
```

```
   public final void toFileSystem(String privateKeyPathName, String
publicKeyPathName)
          throws IOException {

       FileOutputStream privateKeyOutputStream = null;
       FileOutputStream publicKeyOutputStream = null;

       try {

          File privateKeyFile = new File(privateKeyPathName);
          File publicKeyFile = new File(publicKeyPathName);

          privateKeyOutputStream = new FileOutputStream(privateKeyFile);
          privateKeyOutputStream.write(privateKey.getEncoded());
           publicKeyOutputStream = new FileOutputStream(publicKeyFile);
           publicKeyOutputStream.write(publicKey.getEncoded());

       } catch (Exception ioException) {
          throw ioException;
       } finally {

          try {

             if (privateKeyOutputStream != null) {
                privateKeyOutputStream.close();
             }
             if (publicKeyOutputStream != null) {
                publicKeyOutputStream.close();
             }

          } catch (IOException ioException) {
             throw ioException;
          }
       }
   }
}
package rsa;

/**
```

```
 *
 *   This class will use the RSAKeyPair class to generate public and private key with specific key supplied key eg 2024 or 2078
 */
import java.io.File;
import java.io.FileInputStream;
import java.security.KeyFactory;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.apache.commons.io.IOUtils;
import org.junit.Assert;
import org.junit.Test;

public class RSAKeyPairTest {

   String privateKey = "D:" + File.separator + "babu" + File.separator + "itSecurity" + File.separator + "private" + File.separator;
   String publicKey = "D:" + File.separator + "babu" + File.separator + "itSecurity" + File.separator + "public" + File.separator;
   public static void main(String[] args) {
      try {
         new RSAKeyPairTest().testToFileSystem();
      } catch (Exception ex) {
         Logger.getLogger(RSAKeyPairTest.class.getName()).log(Level.SEVERE, null, ex);
      }
   }

   public void testToFileSystem()
         throws Exception {

      try {

         RSAKeyPair rsaKeyPair = new RSAKeyPair(2048);
         rsaKeyPair.toFileSystem(privateKey, publicKey);

         KeyFactory rsaKeyFactory = KeyFactory.getInstance("RSA");
```

```
            Assert.assertNotNull(rsaKeyPair.getPrivateKey());
            Assert.assertNotNull(rsaKeyPair.getPublicKey());
            Assert.assertEquals(rsaKeyPair.getPrivateKey(),
rsaKeyFactory.generatePrivate(new PKCS8EncodedKeySpec(IOUtils.toByteArray(new
FileInputStream(privateKey)))));
            Assert.assertEquals(rsaKeyPair.getPublicKey(),
rsaKeyFactory.generatePublic(new X509EncodedKeySpec(IOUtils.toByteArray(new
FileInputStream(publicKey)))));

        } catch (Exception exception) {
            Assert.fail("The testToFileSystem() test failed because: " +
exception.getMessage());
        }
    }
}
/*
 * Class utility for decrypting and encrypting messages using stored keys
 */
package rsa;
import org.apache.commons.codec.binary.Base64;
import org.apache.commons.io.IOUtils;

import java.io.FileInputStream;
import java.io.IOException;
import java.security.GeneralSecurityException;
import java.security.KeyFactory;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import javax.crypto.Cipher;

public class RSACipherMain {

    public String encrypt(String rawText, String publicKeyPath, String transformation,
String encoding)
            throws IOException, GeneralSecurityException {

        X509EncodedKeySpec x509EncodedKeySpec = new
X509EncodedKeySpec(IOUtils.toByteArray(new FileInputStream(publicKeyPath)));
```

```
        Cipher cipher = Cipher.getInstance(transformation);
        cipher.init(Cipher.ENCRYPT_MODE,
KeyFactory.getInstance("RSA").generatePublic(x509EncodedKeySpec));

        return Base64.encodeBase64String(cipher.doFinal(rawText.getBytes(encoding)));
    }

    public String encrypt(String rawText, PublicKey publicKey, String transformation,
String encoding)
            throws IOException, GeneralSecurityException {

        Cipher cipher = Cipher.getInstance(transformation);
        cipher.init(Cipher.ENCRYPT_MODE, publicKey);

        return Base64.encodeBase64String(cipher.doFinal(rawText.getBytes(encoding)));
    }

    public String decrypt(String cipherText, String privateKeyPath, String transformation,
String encoding)
            throws IOException, GeneralSecurityException {

        PKCS8EncodedKeySpec pkcs8EncodedKeySpec = new
PKCS8EncodedKeySpec(IOUtils.toByteArray(new FileInputStream(privateKeyPath)));

        Cipher cipher = Cipher.getInstance(transformation);
        cipher.init(Cipher.DECRYPT_MODE,
KeyFactory.getInstance("RSA").generatePrivate(pkcs8EncodedKeySpec));

        return new String(cipher.doFinal(Base64.decodeBase64(cipherText)), encoding);
    }

    public String decrypt(String cipherText, PrivateKey privateKey, String transformation,
String encoding)
            throws IOException, GeneralSecurityException {

        Cipher cipher = Cipher.getInstance(transformation);
        cipher.init(Cipher.DECRYPT_MODE, privateKey);

        return new String(cipher.doFinal(Base64.decodeBase64(cipherText)), encoding);
```

```
    }
}

/*
 * This class will enable wakwetu to send encrypted messages to their recipients and and
decrypt the  received messages for the users where their public keys are known
 */
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package rsa;
import java.io.File;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.junit.Assert;
import org.junit.Test;

public class RSACipherTest {

    private final String transformation = "RSA/ECB/PKCS1Padding";
    private final String encoding = "UTF-8";
    public static void main(String[] args) {
        try {
            new RSACipherTest().testEncryptDecryptWithKeyPairFiles();
        } catch (Exception ex) {
            Logger.getLogger(RSAKeyPairTest.class.getName()).log(Level.SEVERE, null,
ex);
        }
    }
    @Test
    public void testEncryptDecryptWithKeyPairFiles()
            throws Exception {

        try {
            String publicKeyBon = "D:" + File.separator + "babu" + File.separator +
"itSecurity" + File.separator + "public.key";
            String privateKeyBon = "D:" + File.separator + "babu" + File.separator +
"itSecurity" + File.separator + "private.perm";
```

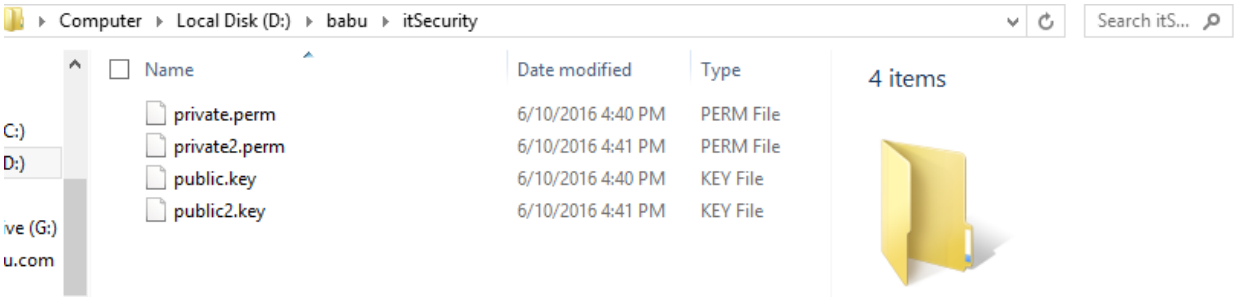
```
        String publicKeyDenis = "D:" + File.separator + "babu" + File.separator +
"itSecurity" + File.separator + "public2.key";
        String privateKeyDenis = "D:" + File.separator + "babu" + File.separator +
"itSecurity" + File.separator + "private2.perm";

        RSAKeyPair rsaKeyPair = new RSAKeyPair(2048);
        rsaKeyPair.toFileSystem(privateKeyDenis, publicKeyDenis);

        RSACipher rsaCipher = new RSACipher();
        String messageText = "Shannon Dorice.";
        System.out.println("Text message:" + messageText);
        String encrypted = rsaCipher.encrypt(messageText, publicKeyBon,
transformation, encoding);
        String decrypted = rsaCipher.decrypt(encrypted, privateKeyBon, transformation,
encoding);
        System.out.println("Encrypted :" + encrypted);
        System.out.println("Decrypted :" + decrypted);
        Assert.assertEquals(decrypted, messageText);

      } catch (Exception exception) {
        Assert.fail("The testEncryptDecryptWithKeyPairFiles() test failed because: " +
exception.getMessage());
      }
   }
}
```

Sample Output



Text message:Shannon Dorice.

Encrypted
:j5Vh46K+MainfWzlvxHeUPI1OkvpPvVSqkVqVCUaB/5g+q2RU/94NPLmBwLcoS1E96N
YjYw9y3dIcIKtvWxkIdYeLlOa3r2W5sNfiQyc67yAfcx3K48d9hhmYQQxIIAXMza20tgNq
HQQLIuoBTtcxUgXaj0scim3fCpFQ0gExTysLAcYMDCtzUSWLrzSzR2Y1N52N19fEQms
Gb1OXmq14kh3PoU0nJv03sOxl6eHK51GiY5f7PKGIAFRqYKbQzxXYftEbW+K6nYBS
Koo5GYwKvvW9BIQEun7KQa/4SV+1asuZ7mMU5JdZn04t15kbSH/FrXIBtTGtwB47LX
rBAk9Pw==

Decrypted :Shannon Dorice.

BUILD SUCCESSFUL (total time: 7 seconds)

4. Security policies for Git Hub
    i. GitHub Security

    We know your code is extremely important to you and your business, and we're very protective of it. After all, GitHub's code is hosted on GitHub.

    ii. Physical Security
        a. Data center access limited to data center technicians and approved GitHub staff
        b. Biometric scanning for controlled data center access
        c. Security camera monitoring at all data center locations
        d. 24x7 onsite staff provides additional protection against unauthorized entry
        e. Unmarked facilities to help maintain low profile
        f. Physical security audited by an independent firm

    iii. System Security
        a. System installation using hardened, patched OS
        b. Dedicated firewall and VPN services to help block unauthorized system access
        c. Distributed Denial of Service (DDoS) mitigation services powered by industry-leading solutions.

    iv. Operational Security
        a. Our primary data center operations are regularly audited by independent firms against an ISAE 3000/AT 101 Type 2 Examination standard
        b. Systems access logged and tracked for auditing purposes

c. Secure document-destruction policies for all sensitive information
d. Fully documented change-management procedures

v. Software Security.

We employ a team of 24/7/365 server specialists at GitHub to keep our software and its dependencies up to date eliminating potential security vulnerabilities. We employ a wide range of monitoring solutions for preventing and eliminating attacks to the site.

vi. Communications

a. All private data exchanged with GitHub is always transmitted over SSL (which is why your dashboard is served over HTTPS, for instance). All pushing and pulling of private data is done over SSH authenticated with keys, or over HTTPS using your GitHub username and password.
b. The SSH login credentials used to push and pull can not be used to access a shell or the filesystem. All users are virtual and have no user account on our machines.

vii. File system and backups

a. Every piece of hardware we use has an identical copy ready and waiting for an immediate hot-swap in case of hardware or software failure. Every line of code we store is saved on a minimum of three different servers, including an off-site backup.
b. We do not retroactively remove repositories from backups when deleted by the user, as we may need to restore the repository for the user if it was removed accidentally.
c. We do not encrypt repositories on disk because it would not be any more secure: the website and git back-end would need to decrypt the repositories on demand, slowing down response times.

d. Any user with shell access to the file system would have access to the decryption routine, thus negating any security it provides. Therefore, we focus on making our machines and network as secure as possible.

viii. Employee access

a. No GitHub employees ever access private repositories unless required to for support reasons. Staff working directly in the file store access the compressed Git database, your code is never present as plaintext files like it would be in a local clone.
b. Support staff may sign into your account to access settings related to your support issue. In rare cases staff may need to pull a clone of your code, this will only be done with your consent.
c. Support staff does not have direct access to clone any repository, they will need to temporarily attach their SSH key to your account to pull a clone. When working a support issue we do our best to respect your privacy as much as possible, we only access the files and settings needed to resolve your issue. All cloned repositories are deleted as soon as the support issue has been resolved.

ix. Maintaining security

a. We protect your login from brute force attacks with rate limiting. All passwords are filtered from all our logs and are one-way encrypted in the database using bcrypt. Login information is always sent over SSL.
b. We also allow you to use two-factor authentication, or 2FA, as an additional security measure when accessing your GitHub account. Enabling 2FA adds security to your account by requiring both your password as well as access to a security code on your phone to access your account.
c. We have full time security staff to help identify and prevent new attack vectors. We always test new features in order to rule out potential attacks, such as XSS-protecting wikis, and ensuring that Pages cannot access cookies.

d. We also maintain relationships with reputable security firms to perform regular penetration tests and ongoing audits of GitHub and its code.
e. We're extremely concerned and active about security, but we're aware that many companies are not comfortable hosting code outside their firewall. For these companies we offer GitHub Enterprise, a version of GitHub that can be installed to a server within the company's network.

x. Credit card safety

When you sign up for a paid account on GitHub, we do not store any of your card information on our servers. It's handed off to Braintree Payment Solutions, a company dedicated to storing your sensitive data on PCI-Compliant servers.

Qn 3. Steps for sending digital signed message using email client.

The protocol used to encrypt emails is called PGP (Pretty Good Privacy).

i. We are going to use thunderbird tool sending and receiving signed and encrypted message emails. Mozilla Thunderbird is a free, open source, cross-platform email, news, and chat client developed by the Mozilla Foundation.
ii. Search for Thunderbird on Google search tool and Install it.
iii. To use PGP within Thunderbird, you must first install:

a. GnuPG : (GNU Privacy Guard): a free software implementation of PGP
b. Enigmail: a Thunderbird add-on

These two applications also provide the capability to digitally sign messages.

iv. Install PGP and Enigmail.
    a. To install GnuPG, download appropriate package from the GnuPG binaries page. Follow the installation instructions provided for your particular package.
    b. We use windows for our particular installation hence we downloaded the GNUPG installer for windows.
    c. Install the gpg4win-2.3.1.exe package you can use any preferred version.
    d. To install Enigmail:
        1. In Thunderbird, select Tools | Add-ons
        2. Use the search bar in the top right corner to search for Enigmail.

3. Use the search bar in the top right corner to search for Enigmail.
4. Select Enigmail from the search results and follow the instructions to install the add-on.

v. Creating PGP keys

You can create your public/private keys by following the following steps:

a. On the Thunderbird menu bar, click OpenPGP and select Setup Wizard.
b. Select Yes, I would like the wizard to get me started as shown in the image below.
c. Click Next to proceed.
d. The wizard will ask whether you want to sign all outgoing messages or whether you want to configure different rules for different recipients. Message recipients do not need to use digital signatures or PGP to read a digitally signed message. Select Yes, I want to sign all of my email and click Next to proceed.
e. Next, the wizard asks if you want to encrypt all your emails. You should not select this option unless you have the public keys for all the people that you expect to send messages to. Select No, I will create per-recipient rules for those who send me their public keys and click Next to proceed.
f. The wizard asks if it can change some of your mail formatting settings to better work with PGP. It is a good choice to answer Yes here. Click Next to proceed.
g. Select the email account for which you want to create the keys. You need to enter a password in the ‘Passphrase’ text box which is used to protect your private key. This password is used to decrypt messages, so don't forget it. The password should be at least 8 characters long and not use any dictionary words. Enter this password twice and click next to proceed.
h. The next screen displays the preferences you configured. If you are satisfied, click Next to proceed.
i. When the process of creating your keys is completed, click Next to proceed.
j. The wizard will ask if you want to create a ‘Revocation certificate’ which you would use if the security of your key pair was compromised and you needed to

inform others that it is no longer valid. If you want to create the file click on Generate Certificate and follow the steps on the subsequent screens. Otherwise, click Skip.

k. The wizard finally informs you that it has completed the process. Click Finish to exit the wizard.

vi. Sending and receiving public keys

Send public key via email, to receive encrypted messages from other people, you must first send them your public key.

a. Compose the message.
b. Select OpenPGP from the Thunderbird menu bar and select Attach My Public Key.
c. Send the email as usual.

vii. Receiving a public key via email, to send encrypted messages to other people, you must receive and store their public key:

a. Open the message that contains the public key.
b. At the bottom of the window, double click on the attachment that ends in '.asc'. (This file contains the public key.)
c. Thunderbird automatically recognizes that this is a PGP key. A dialog box appears, prompting you to ‘Import’ or ‘View’ the key. Click Import to import the key.

viii. Sending a digitally signed and / or encrypted email

a. Compose the message as usual.
b. To digitally sign a message, select OpenPGP from the Thunderbird menu and enable the Sign Message option. To encrypt a message, enable the Encrypt Message option. The system may ask you to enter your Passphrase before encrypting the message.
c. If your email address is associated with a PGP key, the message will be encrypted with that key.

d. If the email address is not associated with a PGP key, you will be prompted to select a key from a list.

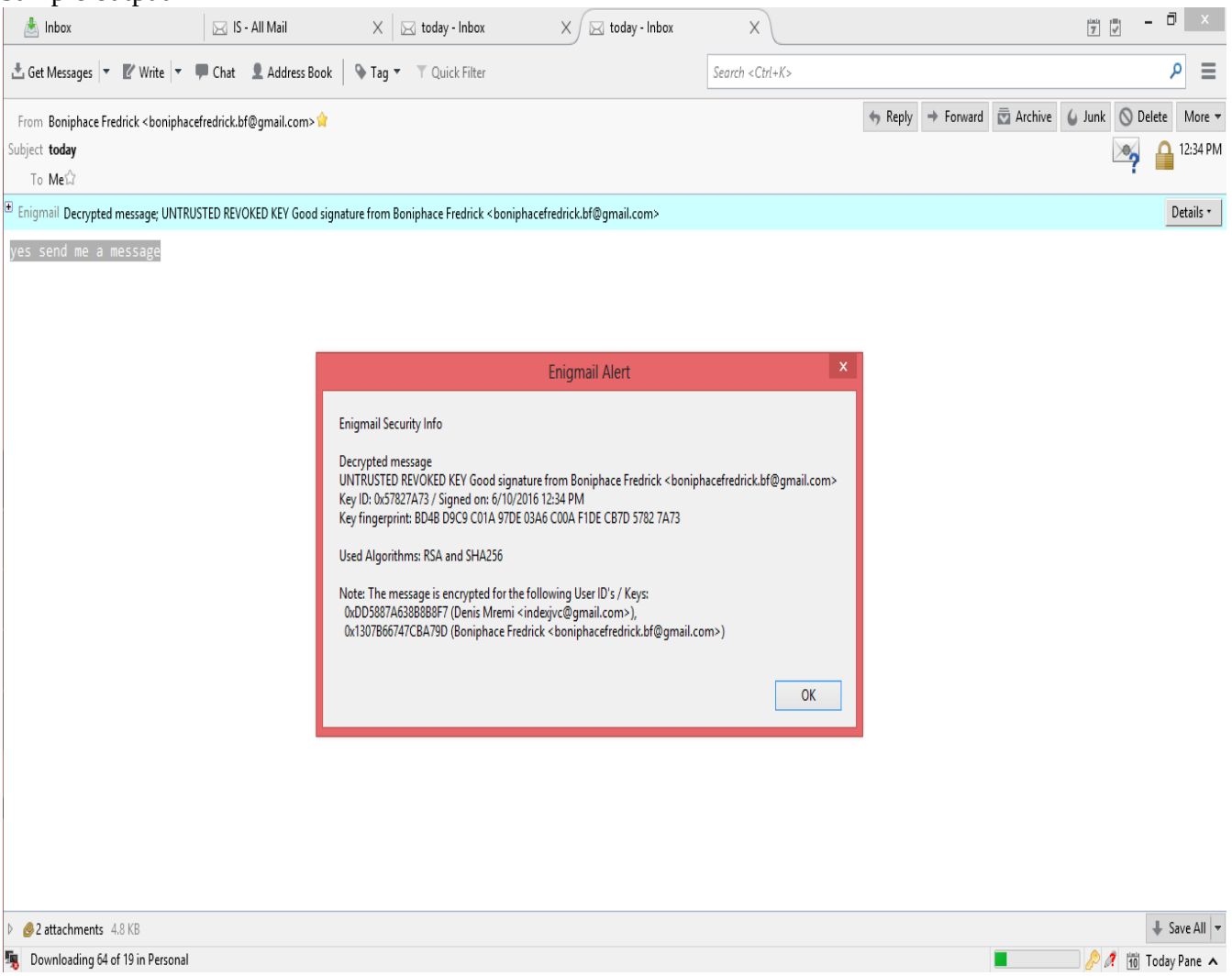
e. Send the message as usual.

ix. Reading a digitally signed and / or encrypted email

a. When you receive an encrypted message, Thunderbird will ask you to enter your secret passphrase to decrypt the message. To determine whether or not the incoming message has been signed or digitally encrypted you need to look at the information bar above the message body.

b. If Thunderbird recognizes the signature, a green bar (as shown below) appears above the message.

x. Sample output



The certificate that we used for sending email to recipient email address we use sender private key and for reading encrypted signed email messages we use public key of sender .The sender and receiver must somehow communicate to verify the public key if is really coming from them.