Using PI to Exchange PGP Encrypted Files in a B2B Scenario

Applies to:

SAP Net Weaver Process Integration 7.1 (SAP PI 7.1). For more information, visit the SOA Management homepage.

Summary

This document talks about creating and using a custom adapter module to PGP encrypt a plain txt file through SAP PI 7.1 using third part libraries (DIDI SOFT).

Author: Amit Srivastava, Anshul Chowdhary

Company: MNC

Created on: 27 July 2011

Author Bio

Amit Srivastava is working as a Consultant on SAP XI/PI. He began his career on Nov-2007 and since then he has been working on SAP XI/PI. His area of expertise is SAP XI/PI, JAVA.

Anshul Chowdhary is working as a Technical Consultant. He began his career on JULY-2006 and has an experience of around 1 year on DOT NET. He started working on SAP XI/PI from December-2007 and is hooked to the technology.

Table of Contents

Why this design?

This integration design provides a secure, traceable and seamless process for transferring sensitive information files in B2B kind of scenarios. EG: Sensitive files to be sent to Banks from SAP etc.

By using this design during these transmissions, all steps involved (refer Fig: 2) in sending such files are logged, acknowledgements are provided for status of the transfer to the source system (EG: SAP) and E-mail alerts can also be sent for errors.

Technical Realization

System Architecture diagram:

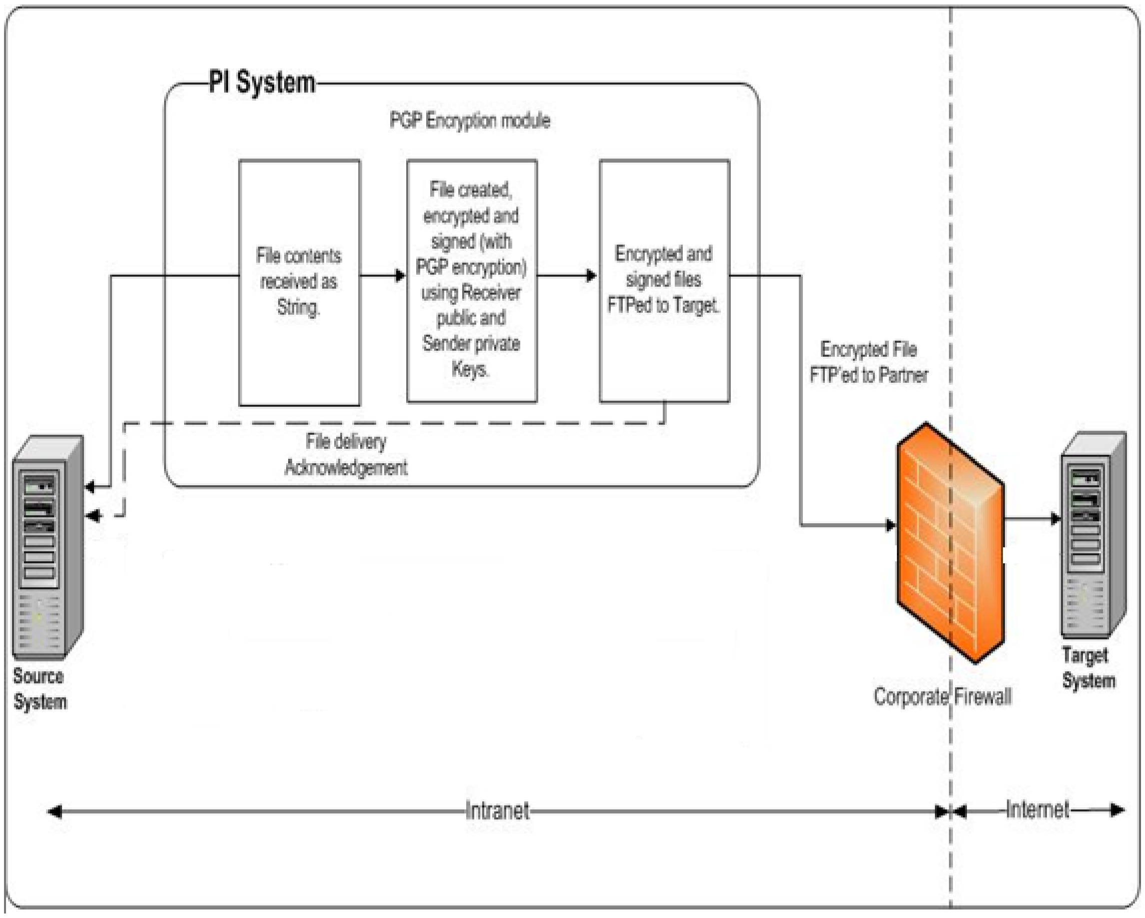

PI System
PGP Encryption module
File contents received as String.
File created, encrypted and signed (with PGP encryption) using Receiver public and Sender private Keys.
Encrypted and signed files FTPed to Target.
Encrypted File FTP'ed to Partner
File delivery Acknowledgement
Source System
Corporate Firewall
Target System
Intranet
Internet

Fig 1: System Architecture.

Process steps:

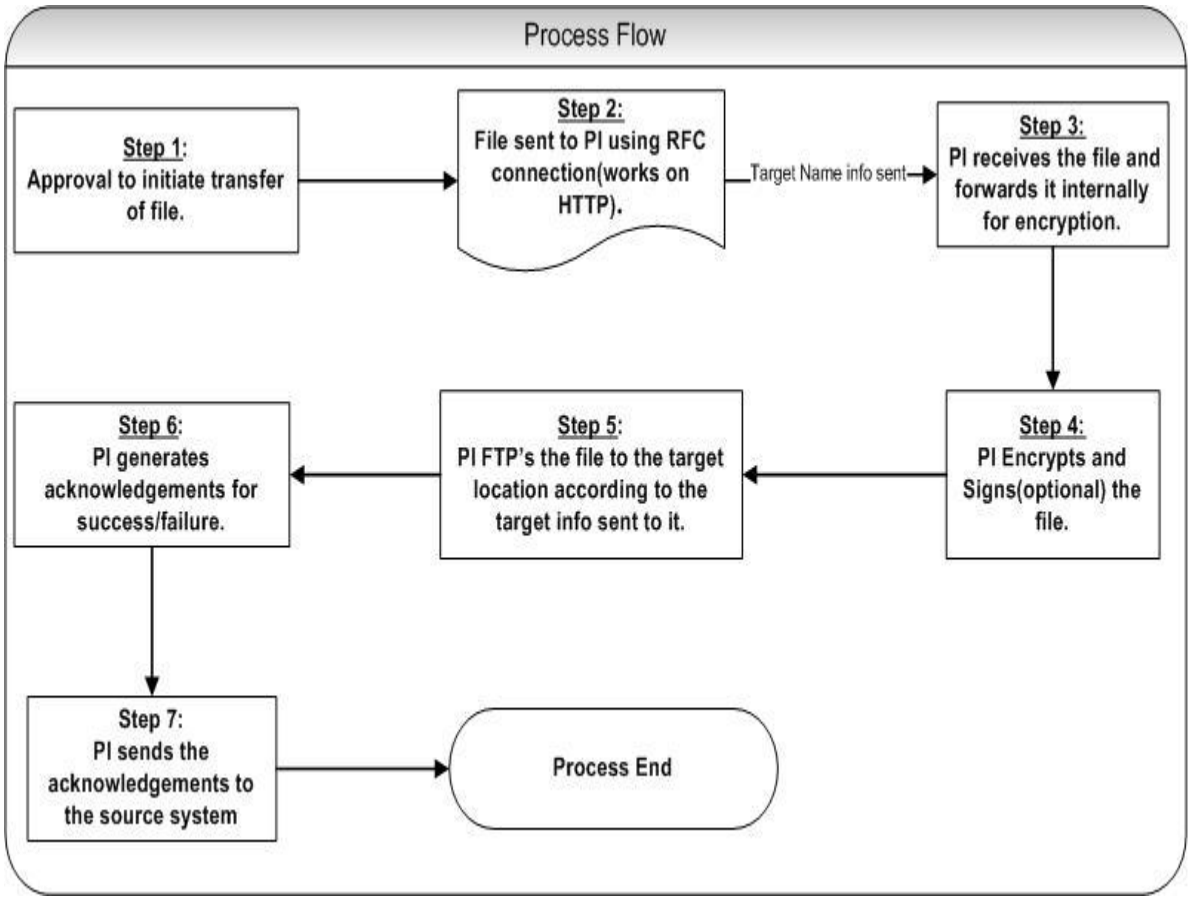

Process Flow
Step 1: Approval to initiate transfer of file.
Step 2: File sent to PI using RFC connection(works on HTTP).
Target Name info sent
Step 3: PI receives the file and forwards it internally for encryption.
Step 4: PI Encrypts and Signs(optional) the file.
Step 5: PI FTP's the file to the target location according to the target info sent to it.
Step 6: PI generates acknowledgements for success/failure.
Step 7: PI sends the acknowledgements to the source system
Process End

Fig 2: Process Flow.

Process design:

The data to be encrypted is received in PI through a Proxy in the structure shown below. The data thus received is converted into a file using the SimpleXML2Plain post the Encryption module is called to encrypt the resulting payload.

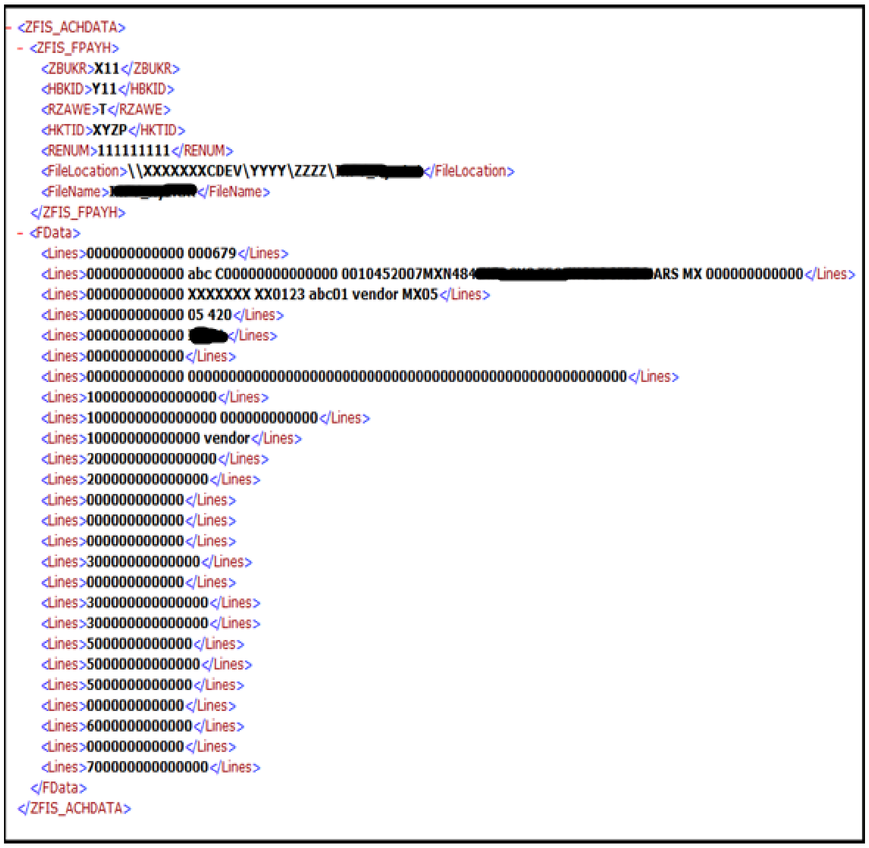
Generic Structure of File data sent to PI:

Data from the Files to be encrypted are either sent to PI or mapped and created in PI in the following structure (which also has other metadata for routing and updating status required later in the process):

```
    <ZFIS_ACHDATA>

          <ZFIS_FPAYH>

             <ZBUKR/>

             <HBKID/>

             <RZAWE/>

             <HKTID/>

             <ZBNKS/>
```

```
              <ZBNKY/>

              <ZBNKL/>

              <SRTF1/>

              <SRTF2/>

              <SRTF3/>

              <RENUM/>

              <FileName/>

           </ZFIS_FPAYH>

           <FData>

              <Lines/>

           </FData>

        </ZFIS_ACHDATA>
```

Sample payload for one such file received by PI (in our scenario) is as shown below:



```
- <ZFIS_ACHDATA>
  - <ZFIS_FPAYH>
      <ZBUKR>X11</ZBUKR>
      <HBKID>Y11</HBKID>
      <RZAWE>T</RZAWE>
      <HKTID>XYZP</HKTID>
      <RENUM>111111111</RENUM>
      <FileLocation>\\XXXXXXXCDEV\YYYY\ZZZZ\[redacted]</FileLocation>
      <FileName>[redacted]</FileName>
    </ZFIS_FPAYH>
  - <FData>
      <Lines>000000000000 000679</Lines>
      <Lines>000000000000 abc C00000000000000 0010452007MXN484[redacted]ARS MX 000000000000</Lines>
      <Lines>000000000000 XXXXXXX XX0123 abc01 vendor MX05</Lines>
      <Lines>000000000000 05 420</Lines>
      <Lines>000000000000 [redacted]</Lines>
      <Lines>000000000000</Lines>
      <Lines>000000000000 0000000000000000000000000000000000000000000000000000000</Lines>
      <Lines>1000000000000000</Lines>
      <Lines>1000000000000000 000000000000</Lines>
      <Lines>10000000000000 vendor</Lines>
      <Lines>2000000000000000</Lines>
      <Lines>200000000000000</Lines>
      <Lines>000000000000</Lines>
      <Lines>000000000000</Lines>
      <Lines>000000000000</Lines>
      <Lines>30000000000000</Lines>
      <Lines>000000000000</Lines>
      <Lines>300000000000000</Lines>
      <Lines>300000000000000</Lines>
      <Lines>5000000000000</Lines>
      <Lines>50000000000000</Lines>
      <Lines>5000000000000</Lines>
      <Lines>000000000000</Lines>
      <Lines>6000000000000</Lines>
      <Lines>000000000000</Lines>
      <Lines>700000000000000</Lines>
    </FData>
  </ZFIS_ACHDATA>
```

Fig 3: Sample payload from ECC to PI.

File Structure Creation

The file is generated in PI by reading the contents of the each node <Lines> in <FData>.

File routing:

The encrypted files generated are routed to file (ftp) channels. The channel has a Target specific encryption details (like Target public keys, private keys for signing etc as shown below) and FTP details.

File Encryption and digitally signing:

The files are encrypted in appropriate modules using custom adapter modules for PGP encryption. These modules can be configured for using Target specific Public keys for encryption. Finally the files are signed using an appropriate private key.

Module Design

Encryption Module Design

The module to encrypt and sign the file using PGP encryption is created using standard PI module framework, which accepts a byte stream and encrypts the same generating another byte stream for subsequent modules. We are using DIDISOFT API for encrypting the files.

For more information about the different DIDIDOFT encryption methods please refer the link mentioned below in Appendix section.

The property of the module is as follows:

Input:

PI payload to encrypt as byte stream.

Output:

Encrypted PI payload as byte stream.

Configurable parameter:

1) ReceiverPublicKey (encUserId)

2) SenderPrivateKey (signUserId)

3) PrivateKeyPassword (privateKey password)

4) KeystoreLocation (input KeyStore Location, containing public and private keys)

5) KeystorePassword (keystore passphrase to access keystore)

Other Dependencies:s

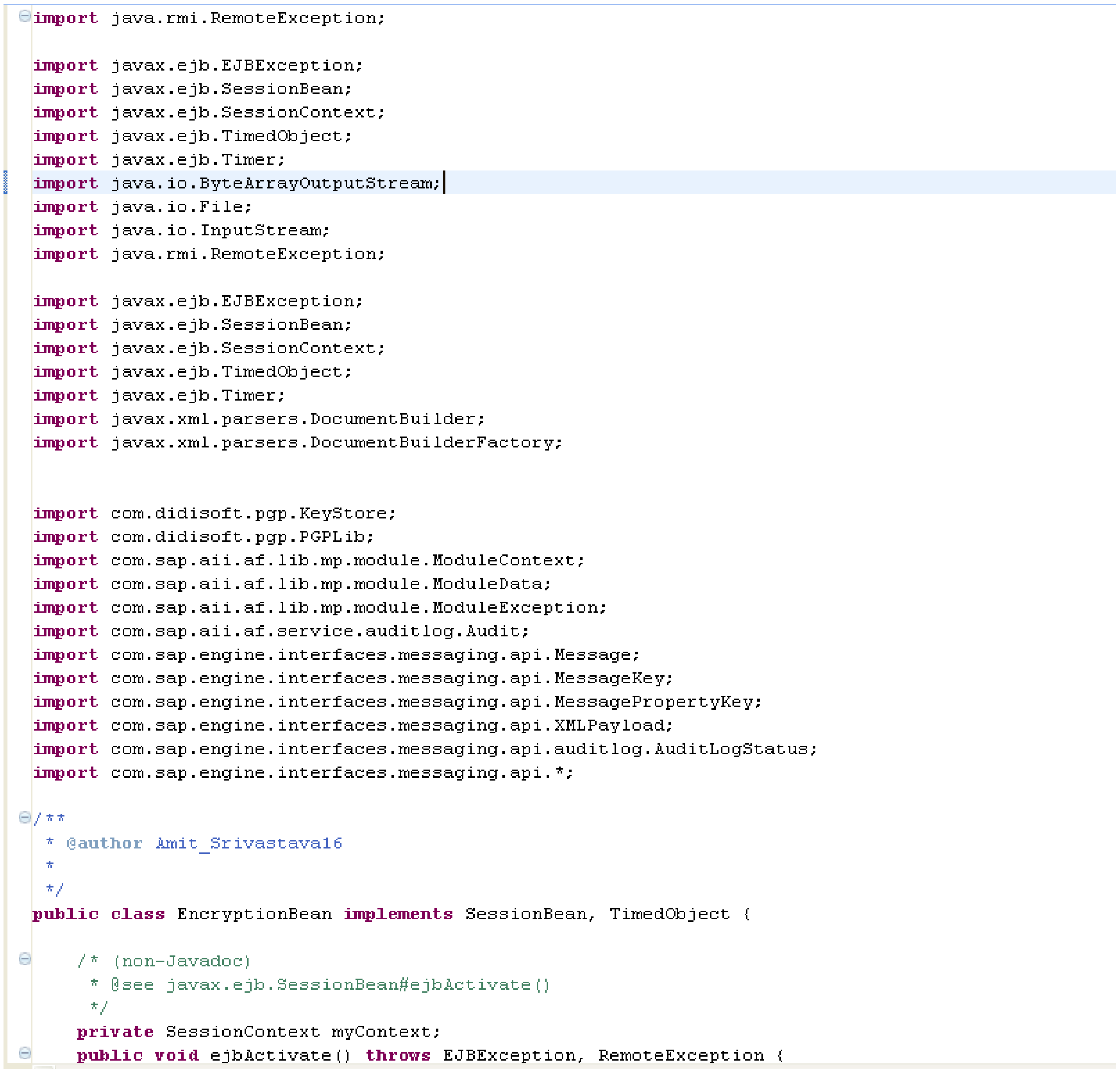
1. Keystore having the Public and Private keys.

Functionality:

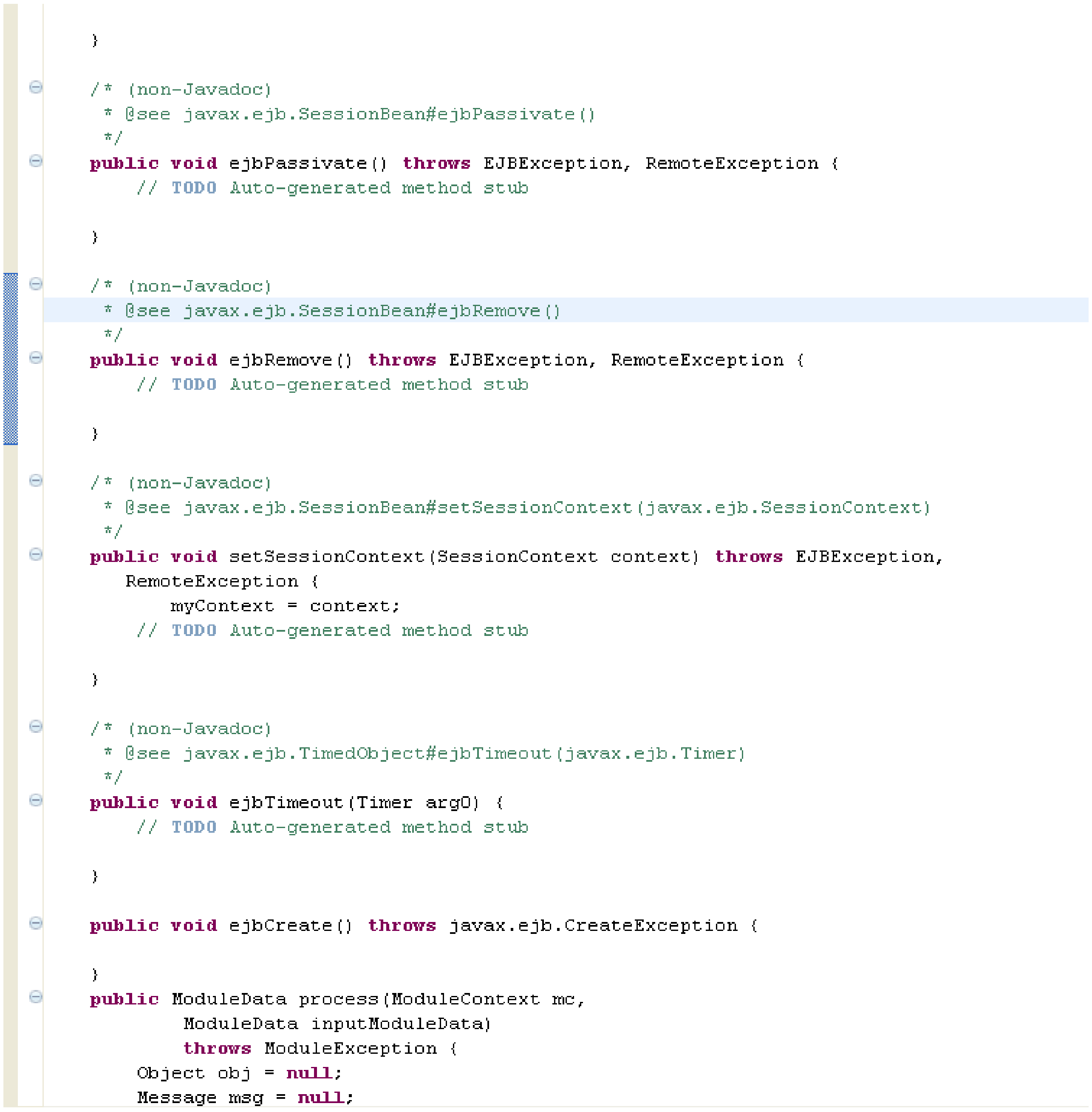
The module should encrypt the data received using the Public key mentioned in the parameter <ReceiverPublicKey> the key for which having a similar name should be available in the keystore stored in a file location on the PI server. The keystore used here is a file with '.keystore' extension. The keystore contains trusted certificates and combinations of private keys with their corresponding certificates. A passphrase is required to access information from this keystore.

Post the encryption, the module should digitally sign the encrypted file using the Private Key mentioned in parameter <SenderPrivateKey> which again should be available with the same name mentioned here in the same keystore as mentioned above.

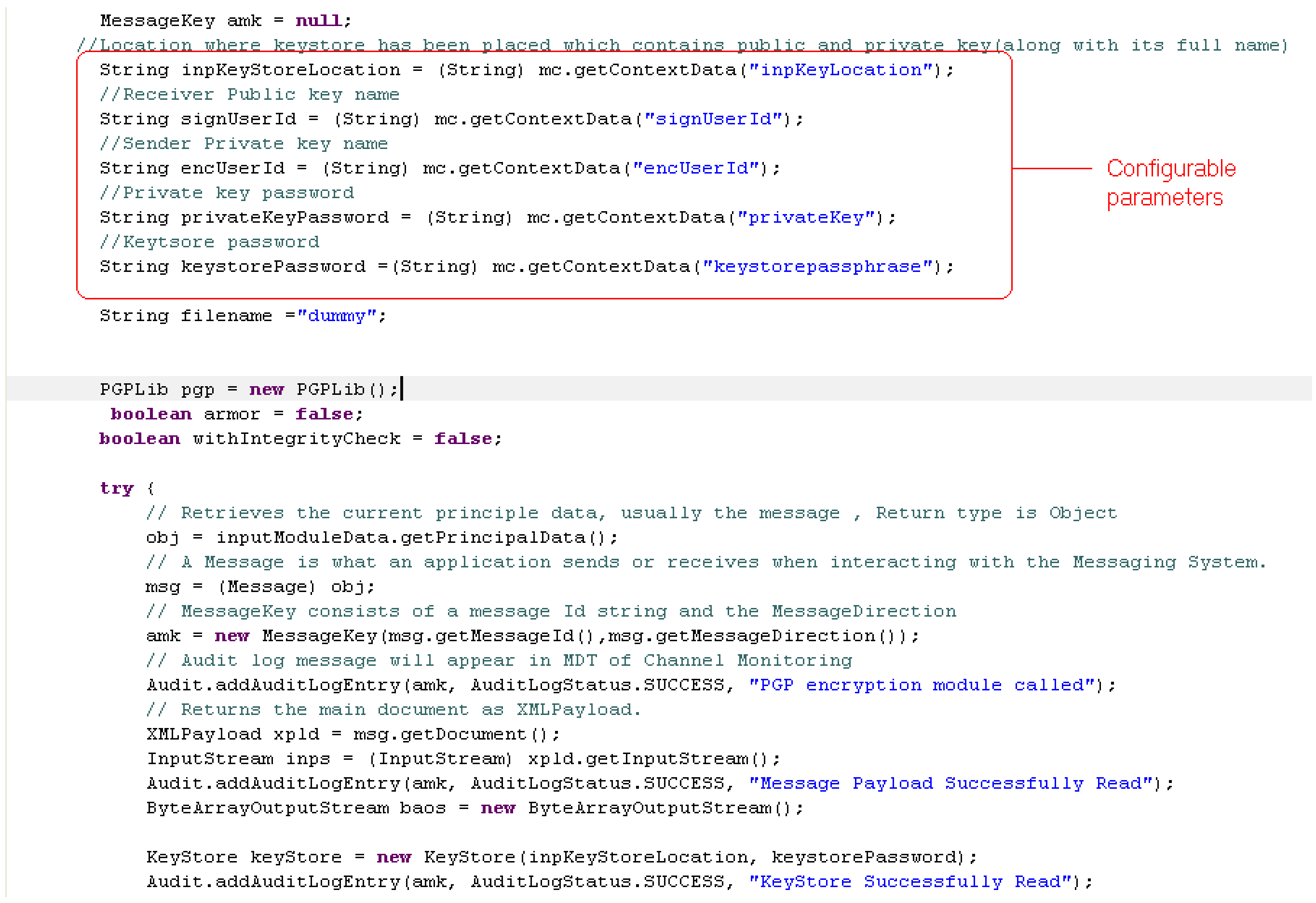
Code: Encryption code



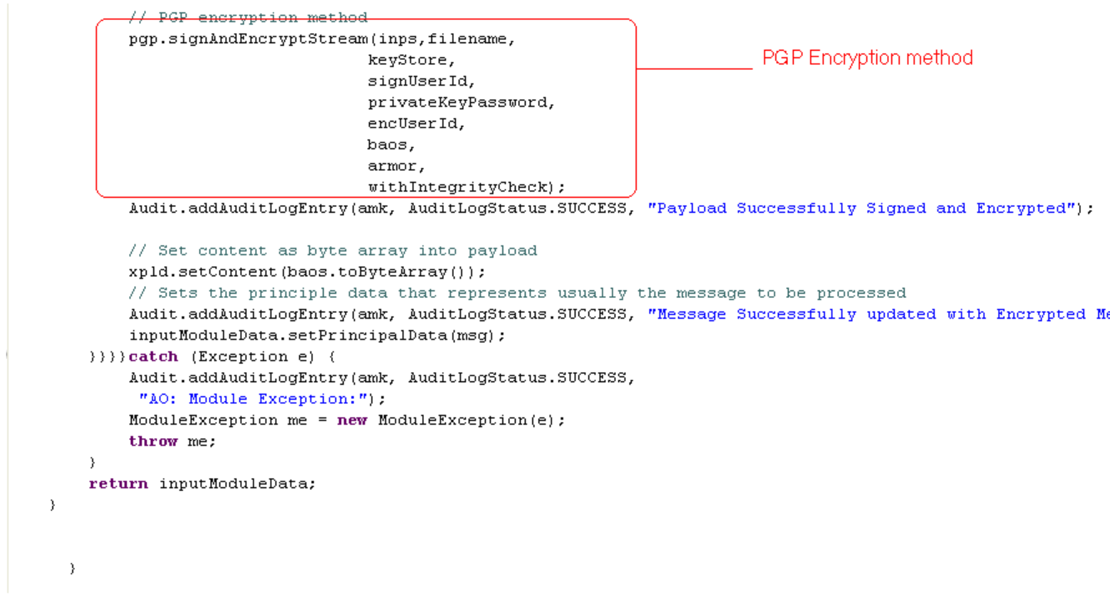
```
import java.rmi.RemoteException;

import javax.ejb.EJBException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.ejb.TimedObject;
import javax.ejb.Timer;
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.InputStream;
import java.rmi.RemoteException;

import javax.ejb.EJBException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.ejb.TimedObject;
import javax.ejb.Timer;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;


import com.didisoft.pgp.KeyStore;
import com.didisoft.pgp.PGPLib;
import com.sap.aii.af.lib.mp.module.ModuleContext;
import com.sap.aii.af.lib.mp.module.ModuleData;
import com.sap.aii.af.lib.mp.module.ModuleException;
import com.sap.aii.af.service.auditlog.Audit;
import com.sap.engine.interfaces.messaging.api.Message;
import com.sap.engine.interfaces.messaging.api.MessageKey;
import com.sap.engine.interfaces.messaging.api.MessagePropertyKey;
import com.sap.engine.interfaces.messaging.api.XMLPayload;
import com.sap.engine.interfaces.messaging.api.auditlog.AuditLogStatus;
import com.sap.engine.interfaces.messaging.api.*;

/**
 * @author Amit_Srivastava16
 *
 */
public class EncryptionBean implements SessionBean, TimedObject {

    /* (non-Javadoc)
     * @see javax.ejb.SessionBean#ejbActivate()
     */
    private SessionContext myContext;
    public void ejbActivate() throws EJBException, RemoteException {
```



```
    }

    /* (non-Javadoc)
     * @see javax.ejb.SessionBean#ejbPassivate()
     */
    public void ejbPassivate() throws EJBException, RemoteException {
        // TODO Auto-generated method stub

    }

    /* (non-Javadoc)
     * @see javax.ejb.SessionBean#ejbRemove()
     */
    public void ejbRemove() throws EJBException, RemoteException {
        // TODO Auto-generated method stub

    }

    /* (non-Javadoc)
     * @see javax.ejb.SessionBean#setSessionContext(javax.ejb.SessionContext)
     */
    public void setSessionContext(SessionContext context) throws EJBException,
       RemoteException {
           myContext = context;
        // TODO Auto-generated method stub

    }

    /* (non-Javadoc)
     * @see javax.ejb.TimedObject#ejbTimeout(javax.ejb.Timer)
     */
    public void ejbTimeout(Timer arg0) {
        // TODO Auto-generated method stub

    }

    public void ejbCreate() throws javax.ejb.CreateException {

    }
    public ModuleData process(ModuleContext mc,
            ModuleData inputModuleData)
            throws ModuleException {
        Object obj = null;
        Message msg = null;
```

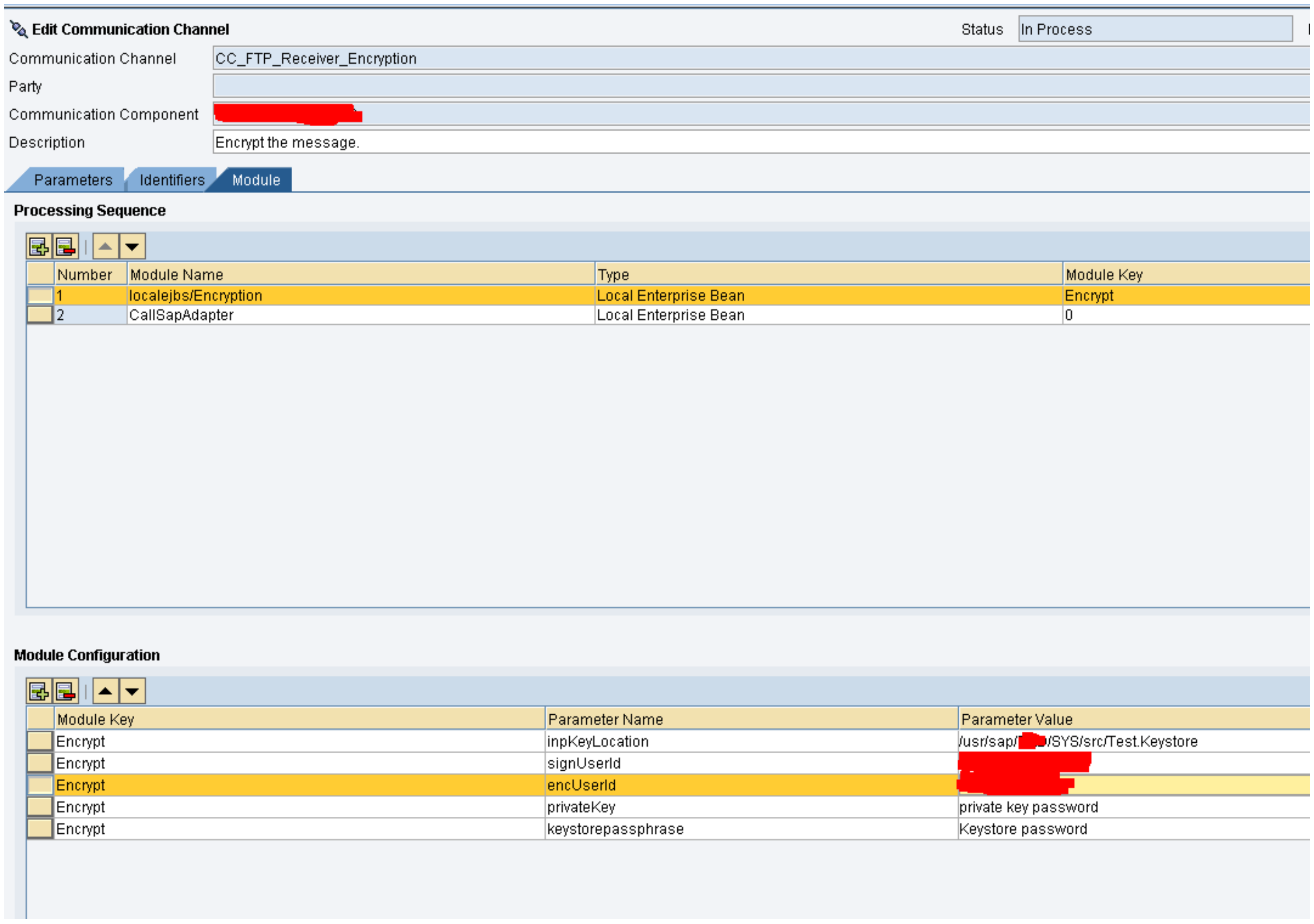


```
        MessageKey amk = null;
      //Location where keystore has been placed which contains public and private key(along with its full name)
        String inpKeyStoreLocation = (String) mc.getContextData("inpKeyLocation");
        //Receiver Public key name
        String signUserId = (String) mc.getContextData("signUserId");
        //Sender Private key name
        String encUserId = (String) mc.getContextData("encUserId");
        //Private key password
        String privateKeyPassword = (String) mc.getContextData("privateKey");
        //Keytsore password
        String keystorePassword =(String) mc.getContextData("keystorepassphrase");

        String filename ="dummy";


        PGPLib pgp = new PGPLib();
         boolean armor = false;
        boolean withIntegrityCheck = false;

        try {
            // Retrieves the current principle data, usually the message , Return type is Object
            obj = inputModuleData.getPrincipalData();
            // A Message is what an application sends or receives when interacting with the Messaging System.
            msg = (Message) obj;
            // MessageKey consists of a message Id string and the MessageDirection
            amk = new MessageKey(msg.getMessageId(),msg.getMessageDirection());
            // Audit log message will appear in MDT of Channel Monitoring
            Audit.addAuditLogEntry(amk, AuditLogStatus.SUCCESS, "PGP encryption module called");
            // Returns the main document as XMLPayload.
            XMLPayload xpld = msg.getDocument();
            InputStream inps = (InputStream) xpld.getInputStream();
            Audit.addAuditLogEntry(amk, AuditLogStatus.SUCCESS, "Message Payload Successfully Read");
            ByteArrayOutputStream baos = new ByteArrayOutputStream();

            KeyStore keyStore = new KeyStore(inpKeyStoreLocation, keystorePassword);
            Audit.addAuditLogEntry(amk, AuditLogStatus.SUCCESS, "KeyStore Successfully Read");
```



```
            // PGP encryption method
            pgp.signAndEncryptStream(inps,filename,
                                    keyStore,
                                    signUserId,
                                    privateKeyPassword,
                                    encUserId,
                                    baos,
                                    armor,
                                    withIntegrityCheck);
            Audit.addAuditLogEntry(amk, AuditLogStatus.SUCCESS, "Payload Successfully Signed and Encrypted");

            // Set content as byte array into payload
            xpld.setContent(baos.toByteArray());
            // Sets the principle data that represents usually the message to be processed
            Audit.addAuditLogEntry(amk, AuditLogStatus.SUCCESS, "Message Successfully updated with Encrypted Message");
            inputModuleData.setPrincipalData(msg);
        }}}}catch (Exception e) {
            Audit.addAuditLogEntry(amk, AuditLogStatus.SUCCESS,
             "AO: Module Exception:");
            ModuleException me = new ModuleException(e);
            throw me;
        }
        return inputModuleData;
    }


     }
```

Receiver Communication Channel Configuration:



Status	In Process
Communication Channel	CC_FTP_Receiver_Encryption
Party	
Communication Component	[redacted]
Description	Encrypt the message.

Processing Sequence

Number	Module Name	Type	Module Key
1	localejbs/Encryption	Local Enterprise Bean	Encrypt
2	CallSapAdapter	Local Enterprise Bean	0

Module Configuration

Module Key	Parameter Name	Parameter Value
Encrypt	inpKeyLocation	/usr/sap/[redacted]/SYS/src/Test.Keystore
Encrypt	signUserId	[redacted]
Encrypt	encUserId	[redacted]
Encrypt	privateKey	private key password
Encrypt	keystorepassphrase	Keystore password

Appendix

Appendix 1a

Working of PGP:

OpenPGP is a non-proprietary protocol for encrypting email using public key cryptography. It is based on PGP as originally developed by Phil Zimmermann. The OpenPGP protocol defines standard formats for encrypted messages, signatures, and certificates for exchanging public keys.

OpenPGP uses a combination of symmetric and asymmetric encryption to secure messages in an effective way and is used widely in the industry.

PGP is most securely used with a combination of encryption and digital signature to verify the authenticity of the intended sender.

More on OpenPGP.

Encryption:

In this encryption method the plain text is encrypted using symmetric key which is different for each session and is called the session key. The session key is further encrypted using a public key and bundled with the encrypted payload as the encrypted message.

However in a PKI kind of environment where there are several holders of the Public key of a particular key pair any one can spoof another user and send a message which the receiver has no way of identifying, hence a digital signature is also used for additional security along with the encryption.

Digital Signature:

Digital signatures enable the recipient of information to verify the authenticity of the information's origin, and also verify that the information is intact. Thus, public key digital signatures provide authentication and data integrity. A digital signature also provides non-repudiation, which means that it prevents the sender from claiming that he or she did not actually send the information. The encrypted message signed with the Private Key of the sender ensures that the authenticity of the message is preserved, considering that the Private Key is possessed by none other than the Sender.

Related Content

http://www.didisoft.com/

http://www.pgpi.org/

For more information, visit the SOA Management homepage.

Disclaimer and Liability Notice