UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

INSTITUTO DE MATEMÁTICA

CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Ian de Castro de Miranda

Criptografia Digital: Implementação do OpenPGP

RIO DE JANEIRO

2018

Ian de Castro de Miranda

Criptografia Digital: Implementação do OpenPGP

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Luis Menasché Schechter

RIO DE JANEIRO

2018

CIP - Catalogação na Publicação

M672c

Miranda, Ian de Castro de
Criptografia digital: implementação do OpenPGP / Ian de Castro de Miranda. -- Rio de Janeiro, 2018.
63 f.

Orientador: Luis Menasché Schechter.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Matemática, Bacharel em Ciência da Computação, 2018.

1. OpenPGP. 2. Criptografia. 3. Assinatura Digital. 4. RSA. I. Schechter, Luis Menasché, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a).

Ian de Castro de Miranda

Criptografia Digital: Implementação do OpenPGP

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em _____ de _____ de _____.

BANCA EXAMINADORA:

_____
Prof. Luis Menasché Schechter, D.Sc. (UFRJ)

_____
Prof. Severino Collier Coutinho, Ph.D. (Universidade de Leeds)

_____
Prof. Hugo de Holanda Cunha Nobrega, Ph.D. (Universidade de Amsterdã)

RESUMO

Esse trabalho inicia-se explicando a criptografia e a assinatura digital que são a base do protocolo OpenPGP, em específico o protocolo RSA que é um dos primeiros protocolos de chave assimétrica.

Após isto, explica-se o que é o protocolo OpenPGP e como ele funciona. Para isto, mostramos como se lê e se cria as mensagens OpenPGP e chaves transferíveis, explicando o significado dos pacotes que compõem as mensagens OpenPGP e a ordem que eles têm que aparecer para ser considerados como uma mensagem OpenPGP válida, permitindo aos leitores compreenderem o funcionamento do OpenPGP.

Palavras-chave: OpenPGP. Criptografia Digital. Assinatura digital. RSA.

ABSTRACT

This work begins by explaining the encryption and digital signature that are the basis of the OpenPGP protocol, in particular the RSA protocol that is one of the first asymmetric key protocols.

After this, it explains what the OpenPGP protocol is and how it works, showing how to read and create OpenPGP messages and transferable keys and explaining the meaning of the packages that make up the OpenPGP messages and the order they have to appear to be considered a valid OpenPGP message, thus allowing readers to understand how OpenPGP works.

Palavras-chave: OpenPGP. Digital cryptography. Digital signatures. RSA.

LISTA DE SIGLAS

PGP – Pretty Good Privacy

GPG – GNU Privacy Guard

MPI – Multiprecision Integers

S2K – String to Key

SUMÁRIO

1 INTRODUÇÃO

Neste trabalho, discutimos e implementamos o protocolo OpenPGP, um protocolo desenvolvido para comunicação por e-mail criptografado usando chaves públicas de criptografia.

A implementação foi realizada usando a linguagem Python e tem as principais funcionalidades do protocolo, que são criptografia, assinatura digital, transferência de chaves públicas e privadas de criptografia, além de outra funcionalidades como compressão, geração de chave de criptografia e armor (transformação dos dados para a base64 e adição de algumas informações extras na mensagem).

A implementação é compatível com outros softwares que implementam o OpenPGP. Utilizando-se o GPG, foi validado que o código implementado conseguia ler e escrever mensagens no padrão OpenPGP. O GPG foi escolhido para os testes por ser um software gratuito que implementa o OpenPGP.

Neste trabalho, apresentamos uma explicação sobre criptografia e assinatura digital e outros conceitos usados no OpenPGP para depois explicarmos como funciona o protocolo, como ele gera as mensagens e como elas são lidas.

1.1 HISTÓRIA DO OPENPGP

O OpenPGP é um protocolo não proprietário usado para criptografar a comunicação por e-mail usando criptografia de chave pública.

Este protocolo define formatos padrão para mensagens, assinaturas e certificados criptografados para troca de chaves públicas.

A partir de 1997, o Grupo de Trabalho OpenPGP foi formado na IETF (Internet Engineering Task Force) para definir este padrão baseado no software PGP, que é proprietário desde que foi desenvolvido por Phil Zimmermann em 1991.

O PGP e, posteriormente, o OpenPGP se tornou o padrão para quase todos os e-mails criptografados do mundo.

O OpenPGP pode ser implementado por qualquer empresa sem pagar taxas de licenciamento.

O PGP foi alvo de uma investigação criminal de três anos, porque o governo dos EUA considerou que as restrições de exportação dos EUA para software criptográfico foram violadas quando o PGP se espalhou por todo o mundo após sua publicação como freeware.

Em 1996, após o governo desistir do caso, Zimmermann fundou PGP Inc.

Em dezembro de 1997, essa empresa e sua propriedade intelectual foram adquiridas pela NAI (Network Associates Inc), que continuou a possuir e desenvolver produtos PGP para fins comerciais e freeware. Em 2002, a NAI interrompeu a desenvolvimento e vendas do PGP e vendeu os direitos a uma nova empresa, a PGP Corporation.

O OpenPGP é a versão de padrões abertos do protocolo de criptografia PGP da NAI. O OpenPGP Working Group está buscando a qualificação do OpenPGP como um padrão da internet, conforme definido pelo IETF. Cada versão distinta de uma especificação relacionada a padrões da internet é publicada como parte da série de documentos RFC (Request for Comments).

O OpenPGP está na Internet Standards Track e está em desenvolvimento ativo. Muitos clientes de e-mail fornecem segurança de e-mail compatível com OpenPGP, conforme descrito na RFC 3156. A especificação atual é RFC 4880 (de Novembro de 2007), o sucessor da RFC 2440. RFC 4880 especifica um conjunto de algoritmos necessários, consistindo em criptografia ElGamal, DSA, Triple DES e SHA-1. Além destes algoritmos, o padrão recomenda o RSA conforme descrito no PKCS # 1 v1.5 para criptografia e assinatura, bem como o AES-128, o CAST-128 e o IDEA. Além destes, muitos outros algoritmos são suportados. O padrão foi estendido para apoiar a codificação Camellia pela RFC 5581 em 2009 e a criptografia baseada em curva elíptica (ECDSA, ECDH) pela RFC 6637 em 2012. O suporte da EdDSA será adicionado pelo draft-koch-eddsa-for-openpgp-00 proposta em 2014.

Aqui estão algumas aplicações de e-mail que usam o protocolo OpenPGP

- Windows
    - eM Client
    - Outlook:
        - gpg4o
        - Gpg4win
        - p≡p

    - Thunderbird: Enigmail
- Mac OS
    - Apple Mail: GPGTools
    - Mutt
    - Thunderbird: Enigmail
- Android
    - K-9 Mail: OpenKeychain
    - p≡p
    - R2Mail2
- iOS
    - iPGMail
- Linux
    - Evolution: Seahorse
    - KMail: Kleopatra
    - Mutt
    - Thunderbird: Enigmail
- Browser Plugins
    - Mailvelope
    - FlowCrypt (Gmail)
    - Psono

2 CRIPTOGRAFIA E ASSINATURA DIGITAL

2.1 CRIPTOGRAFIA

A Criptografia é uma técnica que existe há milhares de anos para evitar que uma informação possa ser lida por qualquer pessoa. A idéia central é que uma mensagem seja alterada de forma que somente um grupo específico de pessoas saibam como reverter a alteração. Existem vários métodos de criptografia, desde os mais simples como a Cifra de substituição monoalfabética (trocar cada letra do alfabeto por uma outra) até os métodos mais complexos como o RSA.

Podemos classificar a criptografia em dois tipos: criptografia de chave simétrica e criptografia de chave assimétrica.

2.1.1 Criptografia de chave assimétrica

A criptografia de chave assimétrica é uma forma de criptografia em que a chave usada para decriptografar a mensagem não pode ser calculada usando a chave para criptografar a mensagem. Ela possui a vantagem de poder tornar a chave de encriptação pública sem que haja risco de que outras pessoas possam usar isso para desencriptar a mensagem, permitindo assim que qualquer um possa mandar uma mensagem ao dono da chave sem que outros possam acessar o conteúdo desta mensagem. Por outro lado, ela possui a desvantagem de ser mais lenta para encriptar e desencriptar a mensagem do que a criptografia de chave simétrica. Temos como exemplo o RSA, em que a chave privada é o par de números (n, d) e a chave pública é o par de números (n, e), sendo que ‘n’ é o produto de dois primos distintos ‘p’ e ‘q’ e ‘d’ multiplicado por ‘e’ é congruente a 1 módulo (p-1)*(q-1). A dificuldade de se obter a chave privada está na necessidade da fatoração de ‘n’ para calcular ‘d’. O problema da fatoração de inteiros possui alta complexidade computacional (a fatoração é muito lenta), se tornando inviável fatorar um número muito grande.

2.1.2 Criptografia de chave simétrica

A criptografia de chave simétrica é uma forma de criptografia em que a chave para decriptografar a mensagem pode ser diretamente obtida com a chave de criptografia da

mensagem. Então, para se manter a mensagem segura, é necessário garantir que apenas o destinatário e o remetente possuem a chave. Desta forma, para cada um que queira mandar uma mensagem, o destinatário deve criar uma chave diferente, sendo necessária a existência de um canal seguro para enviar esta chave de criptografia. Temos como exemplo a Cifra de substituição monoalfabética, em que a chave de criptografia é uma tabela que mapeia cada letra do alfabeto para outra letra e a chave de descriptografia é a tabela inversa. Podemos ver que facilmente conseguimos obter a chave de descriptografia a partir da chave de criptografia, temos apenas que observar que, se uma letra alfa vai para beta na chave de criptografia, então na chave de descriptografia a letra beta vai para alfa.

2.1.3 Criptografia RSA

A criptografia RSA é um sistema de criptografia assimétrica que usa o fato de ser difícil fatorar um número inteiro grande para garantir que não se possa obter a chave privada usando a chave pública.

A geração da chave funciona da seguinte forma:

1. Gere 2 números primos grandes **P** e **Q**.
2. Calcule **N = P*Q**
3. Calcule **C = (P - 1)*(Q - 1)**
4. Escolha um número **E** menor que **C** tal que **E** e **C** sejam primos entre si.
5. Calcule **D** como o inverso multiplicativo de **E** módulo **C**.

Para encriptar uma mensagem iremos inicialmente transformá-la em um vetor de números entre **2** e **N - 2**. Para cada um desses números, iremos fazer a exponenciação a **E** módulo **N**, obtendo, com isso, um vetor de números encriptados.

Para desencriptar a mensagem, iremos pegar cada número deste vetor e exponenciá-los a **D** módulo **N**. Em seguida, utilizamos os números obtidos e remontamos a mensagem original.

2.2 ASSINATURA DIGITAL

A assinatura digital tem o objetivo de certificar que uma mensagem realmente foi escrita por quem disse que a escreveu. De certo modo, é parecida com a criptografia de chave assimétrica, havendo uma chave para assinar a mensagem e uma chave para autenticar a

mensagem. Neste caso, a chave de assinatura é a chave privada e a chave de autenticação é a chave pública. E, diferentemente da chave pública de criptografia, que pode ser enviada de qualquer forma, a chave pública de assinatura digital tem que ser enviada de alguma forma segura. Isto tem o objetivo de garantir que uma terceira parte não possa alterar a chave enviada para uma de sua propriedade, pois isto permitiria que esta terceira parte fingisse ser o remetente original.

2.2.1 Assinatura RSA

A assinatura RSA utiliza uma técnica parecida à criptografia RSA, sendo que a chaves pública e privada são as mesmas mudando apenas a forma como vão ser usadas.

Como as chaves da assinatura RSA são as mesmas da criptografia RSA, elas são calculadas da mesma forma que foi explicada anteriormente.

Como a finalidade da assinatura não é proteger a mensagem e sim garantir a origem, então não precisamos assinar a mensagem toda, mas apenas algo que produza o mesmo resultado. Sendo assim, neste caso iremos assinar um hash (resumo) da mensagem, pois, se o hash for bem feito, não se consegue gerar intencionalmente outra mensagem com o mesmo hash.

Para assinar uma mensagem **M** iremos inicialmente calcular **H = hash(M)** e a assinatura da mensagem será **H** elevado a **D** módulo **N**. Com isso, teremos um número que representará a assinatura da mensagem.

Para autenticar a mensagem (**M, A)** recebida, sendo **M** o conteúdo da mensagem e **A** a assinatura, iremos calcular **H = hash(M)**, e **H2 = (A** elevado a **E** módulo **N)**.

Se **H** e **H2** forem iguais, então a mensagem foi autenticada com sucesso, pois este cálculo garante que quem gerou a assinatura realmente possuía a chave privada.

3 CONCEITOS IMPORTANTES PARA O OPENPGP

Neste capítulo, são apresentados alguns conceitos usados no OpenPGP.

3.1 BASE64

A Base64 é uma forma de codificar um arquivo binário para texto. Dependendo de onde vai ser utilizada, sua implementação pode ser feita de algumas formas diferentes. No caso do OpenPGP, inicia-se separando o arquivo binário em blocos de 6 bits que vão representar um número entre 0 e 63. Cada bloco vai ser convertido para um caracter definido pela seguinte tabela:

Valor	Caracter		Valor	Caracter		Valor	Caracter		Valor	Caracter
0	A		16	Q		32	g		48	w
1	B		17	R		33	h		49	x
2	C		18	S		34	i		50	y
3	D		19	T		35	j		51	z
4	E		20	U		36	k		52	0
5	F		21	V		37	l		53	1
6	G		22	W		38	m		54	2
7	H		23	X		39	n		55	3
8	I		24	Y		40	o		56	4
9	J		25	Z		41	p		57	5
10	K		26	a		42	q		58	6
11	L		27	b		43	r		59	7
12	M		28	c		44	s		60	8
13	N		29	d		45	t		61	9
14	O		30	e		46	u		62	+
15	P		31	f		47	v		63	/

O último bloco pode não conter 6 bits, então existem 2 casos especiais. Quando o último bloco tem apenas 4 bits, serão concatenados 2 bits zeros a direita para poder fazer a conversão. Além disso, será adicionado 1 símbolo de = (igual) ao final do arquivo codificado. Já quando o último bloco tem apenas 2 bits, serão concatenados 4 bits zeros a direita para poder fazer a conversão. Além disso, serão adicionados 2 símbolos de = (igual) ao final do arquivo codificado. Não irá existir o caso de restar apenas 1, 3 ou 5 bits pois, como a entrada é um arquivo em bytes, então irá sempre ter um múltiplo de 8 bits. Assim, ao ir selecionando blocos de 6 bits, só poderá resta 2 ou 4 bits no último bloco. A saída será então representada em linhas, com cada linha contendo no máximo 76 caracteres.

3.2 MULTIPRECISION INTEGERS

Usado para representar inteiros grandes, um Multiprecision Integer (MPI) consiste de 2 partes: o tamanho e o corpo. O tamanho é a quantidade de bits do corpo iniciando pelo primeiro bit 1, sendo escrito em dois octetos[1]. Já o corpo é uma cadeia de octetos representando o número. Por exemplo, o número 510 será representado como 0x00 0x09 0x01 0xFE, onde os dois primeiros octetos representam o tamanho e os demais representam o corpo, pois 510 possui 9 bits e sua representação em hexadecimal é 0x1FE. Escrever como 0x00 0x10 0x01 0xFE é errado, pois mesmo que na representação do 0x1FE se esteja usando 2 octetos, ou seja, 16 bits, a contagem deve iniciar do bit mais significativo que não seja zero, por isso colocar o tamanho como 0x10 é errado.

3.3 STRING TO KEY

Este é um método para gerar uma chave a partir de uma frase, sendo abreviado como S2K. No OpenPGP, se especifica 3 tipos de S2K: Simple S2K, Salted S2K e Iterated and Salted S2K. A idéia básica de todos os métodos é fazer um hash da frase para gerar a chave.

[1] sequência de 8 bits, nomenclatura usada pelo protocolo OpenPGP

3.3.1 Simple S2K

Neste método, inicia-se fazendo o hash da entrada. Se o tamanho do hash for maior ou igual que o tamanho desejado para a chave, então os caracteres mais à esquerda do hash são usados como chave. Caso contrário, será adicionado um octeto de zeros na esquerda da frase, e será gerado um novo hash, que será concatenado à direita com o anterior. Enquanto o tamanho dos hashes concatenados não chegar ao tamanho desejado da chave, será repetido o processo, ou seja, adicionar mais 1 octeto de zeros na frente da chave, calcular o novo hash e concatenar este novo hash. Uma vez que se ultrapasse o tamanho da chave, os caracteres mais à direita que ultrapassem o tamanho desejado serão descartados e o que restar será a chave.

Supondo que se queira uma chave com 16 bytes de tamanho e a função hash gera uma sequência de 6 bytes, então se aplicará a função hash na frase, gerando os primeiros 6 bytes, depois irá se aplicar a função hash no octeto zero concatenado com a frase, gerando os próximos 6 bytes. Como ainda faltam 4 bytes, então se aplicará o hash de novo após se concatenar mais um octeto de zeros à esquerda da frase, tendo-se então concatenados ao total 2 octetos de zeros à frase original, e os bytes gerados pelo hash serão concatenados com os 12 anteriores que já tínhamos, tendo então 18 bytes. Como queríamos uma uma chave de 16 bytes, os dois últimos bytes a direita serão descartados.

3.3.2 Salted S2K

O Salted S2K é bem similar ao Simple S2K, mas, antes de se fazer os passos do Simple S2K, será adicionado um ‘sal’ no início da frase. Esse sal é uma sequência de 8 octetos gerados aleatoriamente (gerado na criação da chave, ele será armazenado para as próximas vezes que se gerar a chave) que serve para dificultar ataques de dicionário.

3.3.3 Iterated and Salted S2K

O Iterated and Salted S2K incrementa ainda mais a dificuldade de ataques de dicionário em relação ao Salted S2K. Neste método, além de adicionar o sal na frase, há um parâmetro contador dizendo quantos caracteres serão lidos pela função hash. Ou seja, serão concatenadas várias instâncias do sal + frase até que o tamanho seja maior que o contador,

sendo que serão descartados os últimos caracteres para que o tamanho seja exatamente o do contador.

Suponha que minha frase tenha 12 caracteres (isto é, 12 bytes) e que o contador seja 96. Como o sal tem tamanho de 8 bytes (8 octetos), ao concatenar o sal mais a frase teremos tamanho de 20 bytes. Então, serão concatenadas 5 instâncias do sal + frase e, após descartarmos os 4 últimos bytes, teremos 96 bytes para serem lidos pela função de hash. Após este passo, será usada esta sequência de bytes para aplicar os passos do simple S2K.

3.4 ID DA CHAVE E FINGERPRINTS

O ID da chave e Fingerprints são formas compactas de identificar uma chave. Atualmente, o OpenPGP aceita duas versões para o ID da chave e fingerprints, as versões 3 e 4.

Na versão 3, só estava disponível o RSA como método de criptografia e assinatura digital. Nesta versão, o ID da chave consiste dos 64 bits menos significativos do módulo público da chave RSA e o fingerprint é formado aplicando hash MD5 no MPI (sem o tamanho) do módulo público N seguido pelo expoente E. Tanto a chave da versão 3 quanto o hash MD5 são obsoletos. Assim, não se deve gerar novas chaves na versão 3, mas se pode aceitar estas chaves.

Na versão 4, o fingerprint é formado pelos 160-bits do hash SHA-1 aplicado no octeto 0x99 seguido por dois octetos contendo o tamanho do corpo do pacote da chave pública e pelo corpo do pacote da chave pública desde a versão até os MPIs. O ID da chave são os 64 bits menos significativos do fingerprint.

Observe-se que duas chaves diferentes podem possuir o mesmo ID de chave ou até mesmo, com uma probabilidade bem menor, possuir o mesmo fingerprint e as aplicações têm que estar preparadas para aceitarem isso.

O ID da chave e fingerprint de uma subchave é calculado da mesma forma que a de uma chave, incluindo o fato de iniciar com o octeto 0x99, mesmo que isso não forme um pacote válido para uma subchave.

4 PROTOCOLO OPENPGP

Neste capítulo, vou explicar os componentes do protocolo e como se lê uma mensagem OpenPGP recebida.

4.1 PACOTE OPENPGP

O Pacote OpenPGP é formado por duas partes: o cabeçalho e o corpo. No cabeçalho, há a tag do pacote e o tamanho do corpo, sendo que a tag é um número que indica o tipo do pacote. Já no corpo, há os dados do pacote, sendo que o significado vai ser dado pela tag do pacote, ou seja, para cada tipo de pacote diferente a forma do corpo também será diferente, não sendo possível ler o corpo do pacote sem saber qual é a sua tag.

4.2 TIPOS DE PACOTES POR TAG

4.2.1 Reservado (Tag 0)

Isto não é um pacote, é apenas uma indicação de que nenhum pacote pode ter a tag 0.

4.2.2 Pacote de Sessão de Chave Pública Encriptada (Tag 1)

O pacote de Sessão de Chave pública Encriptada serve para guardar a chave de sessão que vai desencriptar a mensagem. Contém a chave de sessão criptografada e suas especificações: o id da chave que a criptografou e o algoritmo de chave assimétrica usado para criptografar.

4.2.3 Pacote de Assinatura (Tag 2)

O pacote de Assinatura serve para validar a mensagem. Esse pacote contém uma assinatura e suas especificações: o tipo de assinatura, o algoritmo usado para assinar e o hash usado, além de, opcionalmente, algumas outras informações que ficam em subpacotes.

4.2.4 Pacote de Sessão de Chave Simétrica Encriptada (Tag 3)

O pacote de Sessão de Chave Simétrica Encriptada serve para guardar ou gerar a chave de sessão que vai desencriptar a mensagem. Esse pacote contém o algoritmo simétrico usado para a encriptação, a especificação do S2K usado e pode ou não conter uma chave de sessão criptografada. Caso não contenha a chave de sessão criptografada, o S2K vai ser usado para gerar a chave de sessão. Já caso contenha a chave de sessão criptografada, o S2K vai ser usado para gerar uma chave para decriptografar a chave de sessão.

4.2.5 Pacote de Assinatura de Um Passo (Tag 4)

O pacote de Assinatura de Um Passo serve para guardar algumas informações para permitir que o destinatário comece a calcular qualquer hash necessário para autenticar a mensagem, permitindo a assinatura estar no final da mensagem e o destinatário calcular a totalidade da mensagem assinada em um passo. Esse pacote contém o tipo de assinatura usado, o algoritmo de hash usado, o algoritmo de chave pública usado, o id da chave de quem assinou e se vai ter outro *pacote de Assinatura de Um Passo***.**

4.2.6 Pacote de Chave Privada (Tag 5)

O pacote de Chave Privada serve para guardar uma chave privada. Esse pacote inicia contendo as informações da sua chave pública correspondente e depois vem a parte privada da chave. A parte privada da chave pode ou não estar criptografada, se estiver vai conter também as informações de como decriptografar a chave privada, e no final vai ter um hash da parte secreta da chave que vai ser criptografada junto da chave.

A parte privada da chave vai depender do algoritmo de chave assimétrica que vai estar especificado junto da parte pública da chave. Caso o algoritmo de chave assimétrica seja o RSA então vai conter 4 números escritos em MPI na parte privada da chave, o expoente secreto **D** do RSA, o valor primo secreto **P** do RSA, o valor primo secreto **Q(P < Q)** do RSA e o valor de **U** que é o inverso de **P** modulo **Q**.

4.2.7 Pacote de Chave Pública (Tag 6)

O pacote de Chave Pública serve para guardar uma chave pública. Esse pacote contém a hora em que a chave foi criada, o algoritmo de chave pública usado e os valores que dependem do algoritmo de chave pública usado. Caso seja o RSA, então tem 2 valores escritos em MPI, o módulo público **N** do RSA e o expoente público **E** do RSA.

4.2.8 Pacote de Subchave Privada (Tag 7)

O pacote de Subchave Privada serve para guardar uma subchave privada. Seu formato é igual ao de uma chave privada.

4.2.9 Pacote de Dado Comprimido (Tag 8)

O pacote de Dado Comprimido serve para guardar a mensagem de forma comprimida. Esse pacote contém o algoritmo usado para comprimir e a mensagem OpenPGP comprimida. Ao fazer a descompressão, vai ser obtido uma mensagem OpenPGP válida.

4.2.10 Pacote de Dado Encriptado Simetricamente (Tag 9)

O pacote de Dado Encriptado Simetricamente serve para guardar a mensagem de forma encriptada. Esse pacote contém apenas a mensagem encriptada. Para desencriptar a mensagem será usado a chave de sessão que terá vindo antes em um *pacote de Sessão de Chave Pública Encriptada* ou em um *pacote de Sessão de Chave Simétrica Encriptada*.

4.2.11 Pacote de Marcação (Tag 10)

O pacote de Marcação serve para indicar que a mensagem usa algumas funções não compatíveis com versões mais antigas do OpenPGP. O pacote consiste de apenas 3 octetos, 0x50, 0x47, 0x50, que representa PGP em UTF-8.

4.2.12 Pacote de Dado Literal (Tag 11)

O pacote de Dado Literal serve para armazenar mensagem. O pacote vai conter uma indicação se o dado é texto ou binário, o nome do arquivo, uma data de criação ou alteração do arquivo e o dado do arquivo.

4.2.13 Pacote de Confiança (Tag 12)

O pacote de Confiança serve para armazenar dados para lembrar o usuário certas especificações. Esse pacote não é para ser colocado em mensagens que vão ser transmitidas para outros usuários.

4.2.14 Pacote de ID de Usuário (Tag 13)

O pacote de ID de Usuário serve para armazenar o id do dono da chave. Usualmente contém o nome e e-mail do dono da chave, mas não há restrições sobre o conteúdo desse pacote.

4.2.15 Pacote de Subchave Pública (Tag 14)

O pacote de Subchave Pública serve para guardar uma subchave pública. Seu formato é igual ao de uma chave pública.

4.2.16 Pacote de Atributo de Usuário (Tag 17)

O pacote de Atributo de Usuário serve para conter algumas outras informações sobre o dono da chave que não estão no *pacote de ID de Usuário*. Esse pacote vai conter um ou mais subpacotes. Atualmente, o único tipo de subpacote válido para esse pacote é o subpacote de imagem.

4.2.17 Pacote de Dado com Proteção de Integridade Encriptado Simetricamente (Tag 18)

O pacote de Dado com Proteção de Integridade Encriptado Simetricamente é uma variação do *pacote de Dado Encriptado Simetricamente*, que aborda o problema de detecção de modificação em dados encriptados. Ele é usado em combinação com o *pacote de Detecção de Modificação*. Este pacote contém apenas a mensagem encriptada. Para desencriptar a mensagem, será usada a chave de sessão que terá vindo antes em um *pacote de Sessão de Chave Pública Encriptada* ou em um *pacote de Sessão de Chave Simétrica Encriptada*. Após a decriptação, esse pacote vai conter uma mensagem OpenPGP cujo último pacote vai ser o *pacote de Detecção de Modificação*.

4.2.18 Pacote de Detecção de Modificação (Tag 19)

O pacote de Detecção de Modificação serve para validar se houve alguma alteração no dado encriptado. Esse pacote contém o hash SHA-1 do restante do dado encriptado que veio antes no *pacote de Dado com Proteção de Integridade Encriptado Simetricamente*. Esse pacote sempre é usado em conjunto com o *pacote de Dado com Proteção de Integridade Encriptado Simetricamente* e será o último pacote encriptado pelo *pacote de Dado com Proteção de Integridade Encriptado Simetricamente*, não podendo aparecer em outros lugares.

4.2.19 Valores experimentais ou privados (Tags 60 até 63)

Diferentemente da tag 0 as tags 60 até 63 são pacotes, mas esses valores são para testar pacotes que ainda não estão definidos na versão atual do OpenPGP.

4.3 Objetos OpenPGP

Objetos OpenPGP são: Chaves públicas transferíveis, chaves privadas transferíveis, mensagens OpenPGP e assinaturas desanexadas. E cada um desses objetos é formando concatenando pacotes OpenPGP em uma sequência que segue algumas regras, ou seja, não é

qualquer sequência de pacotes que irá gerar um objeto OpenPGP válido. A seguir, mostrarei quais são as sequências de pacotes que geram objetos OpenPGP válidos.

4.3.1 Chave pública transferível

A chave pública transferível é usada para poder enviar chaves e subchaves públicas para outras pessoas. As chaves públicas são usadas para que outras pessoas possam validar as assinaturas feitas pelo dono da chave, e as subchaves públicas são usadas para criar mensagens criptografadas que se quer mandar ao dono da subchave. Uma chave pública transferível é válida se possui a seguinte sequência de pacotes:

1. Um *pacote de Chave Pública*.
2. Zero ou mais assinaturas de revogação.
3. Um ou mais *pacotes de ID de Usuário*
4. Após cada *pacote de ID de Usuário*, zero ou mais *pacotes de Assinatura(certificação).*
5. Zero ou mais *pacotes de Atributos de Usuário*.
6. Após cada *pacote de Atributo de Usuário*, zero ou mais pacotes de *Assinatura(certificação).*
7. Zero ou mais *pacotes de Subchave Pública*.
8. Após cada *pacote de Subchave*, um *pacote de Assinatura* e opcionalmente após isso uma revogação.

4.3.2 Chave privada transferível

A chave privada transferível é usada para poder enviar suas chaves e subchaves privadas para outras máquinas, podendo assim usar as suas chaves de múltiplas máquinas diferentes, sendo que para poder usar a chave privada será necessário ter a senha da chave. As chaves privadas são usadas para que se possa criar assinaturas, e as subchaves privadas são usadas para desencriptar mensagens recebida de outras pessoas. O formato de uma chave privada transferível válida é igual ao de uma chave pública transferível, substituindo apenas os *pacotes de Chave Pública* e *pacotes de Subchave Pública* por *pacotes de Chave Privada* e *pacotes de Subchave Privada*.

4.3.2.1 Senha OpenPGP

Nos *pacotes de Chave Privada* e *pacotes de Subchave Privada* a parte secreta das chaves e subchaves podem ser criptografadas através de uma criptografia de chave simétrica, então mesmo que se tenha acesso ao pacote com chave privada não se poderá ler o conteúdo do pacote sem ter a senha.

Como a senha é algo que o usuário escolhe, então não é apropriada para ser usada como chave simétrica para criptografar o pacote. Desta forma, é usado o S2K na senha para gerar uma sequência de bytes para aí sim ser usado como chave simétrica para criptografar as chaves privadas e subchaves privadas.

4.3.3 Mensagem OpenPGP

A mensagem OpenPGP é uma mensagem no qual foi aplicado o protocolo OpenPGP. A mensagem pode ter sido assinada, criptografada, comprimida ou ter se mantido na forma literal. Podemos descrever uma mensagem OpenPGP pela seguinte regra:

- ➔ **Mensagem OpenPGP**:
  - ◆ **Mensagem encriptada**.
  - ◆ **Mensagem assinada**.
  - ◆ **Mensagem comprimida**.
  - ◆ **Mensagem literal**.
- ➔ **Mensagem literal**:
  - ◆ *Pacote de Dado Literal*.
- ➔ **Mensagem encriptada**:
  - ◆ Zero ou mais **Chaves de Sessão Encriptadas**, seguido por um **Dado Encriptado.**
- ➔ **Mensagem assinada**:
  - ◆ *Pacote de Assinatura* seguido por uma **Mensagem OpenPGP**.
  - ◆ *Pacote de Assinatura de Um Passo* seguido por uma **Mensagem OpenPGP** seguido por *pacote de Assinatura*
- ➔ **Mensagem comprimida**:
  - ◆ *Pacote de Dado Comprimido*.
- ➔ **Chave de Sessão Encriptada**:

- ◆ *Pacote de Sessão de Chave Pública Encriptada*.
- ◆ *Pacote de Sessão de Chave Simétrica Encriptada*.

➔ **Dado Encriptado**:

- ◆ *Pacote de Dado Encriptado Simetricamente*.
- ◆ *Pacote de Dado com Proteção de Integridade Encriptado Simetricamente*.

E além disso, ao desencriptar o *pacote de Dado Encriptado Simetricamente* ou o *pacote de Dado com Proteção de Integridade Encriptado Simetricamente*, ou ainda ao descomprimir o *pacote de Dado Comprimido*, deve-se obter uma **Mensagem OpenPGP** válida.

4.3.4 Assinatura desanexada

A assinatura desanexada é um *pacote de Assinatura* armazenado em um arquivo diferente do arquivo a ser assinado, permitindo assim enviar apenas a assinatura por um canal seguro para garantir que o arquivo assinado não sofreu alterações.

4.4 ARMOR OPENPGP

Ao criar um objeto OpenPGP, se obtém uma sequência de octetos. No entanto, alguns sistemas só permitem o uso de blocos consistindo de caracteres imprimíveis de sete bits, então o armor do OpenPGP vem para fornecer um serviço de codificação da mensagem para a Base64.

Além de converter para a base64, esse serviço vai colocar um cabeçalho, anotações, uma validação e um rodapé sobre a mensagem. Cada uma dessas partes vai ser descritas abaixo.

4.4.1 Cabeçalho

O cabeçalho vai estar na primeira linha e vai depender do tipo de mensagem. É importante que o cabeçalho venha com os 5 traços antes e depois e em uma linha completa, ou seja, não deve haver nenhum caractere na mesma linha antes ou depois do cabeçalho. Os cabeçalhos para cada tipo de mensagem são:

- Para chaves públicas: -----BEGIN PGP PUBLIC KEY BLOCK-----

- Para chaves privadas: -----BEGIN PGP PRIVATE KEY BLOCK-----
- Para mensagem: -----BEGIN PGP MESSAGE-----
- Para mensagem com varias partes, onde a mensagem possui Y partes e a parte atual é a parte X: -----BEGIN PGP MESSAGE, PART X/Y-----
- Para mensagem com varias partes, onde não se conhece o número de partes e a parte atual é a parte X: -----BEGIN PGP MESSAGE, PART X-----
- Para assinatura desanexada, assinaturas MIME/OpenPGP ou assinatura em texto puro: -----BEGIN PGP SIGNATURE-----

4.4.2 Anotações

As anotações virão a partir da segunda linha e irão terminar quando vier uma linha em branco (pode não haver anotações e, com isso, a segunda linha ser em branco) e estarão no formato '<chave>: <valor>' (sem aspas), onde <chave> vai ser Version, Comment, MessageID, Hash ou Charset. Como existem alguns métodos de transporte sensíveis ao tamanho da linha, nenhuma linha poderá ter mais de 76 caracteres. Além disso, como o <valor> pode ter qualquer tamanho, então ele pode ter que se separar em várias linhas, porém todas as linhas terão que começar com o '<chave>:' correspondente.

Caso venha uma <chave> desconhecida esta deverá ser reportada para o usuário, mas o OpenPGP deverá continuar a processar a mensagem. O <valor> depende da <chave> e o conteúdo do <valor> vai ser o descrito abaixo.

- Version: a versão usada para codificar a mensagem.
- Comment: Um comentário qualquer feito pelo usuário.
- MessageID: 32 caracteres imprimíveis, o MessageID tem que ser o mesmo para todas as partes de mensagens com várias partes, e tem que tentar ser diferente do MessageID de outras mensagens. O MessageID só tem que aparecer para mensagens de várias partes e deve ser calculado a partir da mensagem final de forma determinística.
- Hash: uma lista separada por vírgulas contendo os algoritmos hash usados. É usado apenas para assinatura em texto puro.
- Charset: Uma descrição do conjunto de caracteres no qual o texto original está. O OpenPGP define por padrão os textos como estando em UTF-8.

4.4.3 Dado

Após a linha em branco que termina as anotações, vem o Objeto OpenPGP convertido para a Base64.

4.4.4 Validação

Após o dado, vem uma validação para garantir que não houve uma alteração não intencional no dado. É usada a Verificação Cíclica de Redundância de 24 bits convertida para a Base64 e precedida por '=' (símbolo de igual).

4.4.5 Rodapé

Na última linha após a validação, vem o rodapé, que é igual ao cabeçalho substituindo apenas o 'BEGIN' por 'END'. Por exemplo, se no cabeçalho veio '-----BEGIN PGP MESSAGE-----', então no rodapé vai vir '-----END PGP MESSAGE-----'.

4.5 CRIPTOGRAFIA OPENPGP

No OpenPGP, haverá uma mistura entre criptografia de chave simétrica com criptografia de chave assimétrica. A mensagem será criptografada usando uma chave simétrica, chamada de chave de sessão, para se aproveitar do fato de ser mais rápida a encriptação e desencriptação da mensagem, e a chave de sessão será transmitida após ser criptografada usando uma criptografia assimétrica, fazendo os seguintes passos.

- O remetente cria a mensagem.
- O remetente cria uma chave de sessão aleatória(para cada mensagem mandata será criada uma nova chave de sessão).
- O remetente usa a chave de sessão para criptografar a mensagem que se quer enviar.
- O remetente usa a chave pública do destinatário para encriptar a chave de sessão.
- O remetente envia a mensagem e a chave de sessão que foram encriptadas.
- O destinatário usa a sua chave privada para desencriptar a chave de sessão recebida.
- O destinatário usa a chave de sessão para desencriptar a mensagem.

4.6 Assinatura digital OpenPGP

A assinatura digital OpenPGP possui o intuito de certificar o remetente, garantindo que terceiros não possam mandar mensagens em nome dele. Ela possui um conceito semelhante ao da criptografia de chave assimétrica: o remetente possui uma chave pública que é divulgada a todos, e uma chave privada que apenas ele conhece. Nesta situação, o remetente usa a chave privada para assinar a mensagem, que poderá ser certificada usando a chave pública.

- O remetente cria a mensagem
- O remetente faz um hash da mensagem a ser enviada.
- O remetente usa a sua própria chave privada para aplicar uma assinatura no hash da mensagem.
- O remetente envia a mensagem e a assinatura do hash.
- O destinatário válida a assinatura comparando o hash que ele gerou usando a mensagem recebida, com o hash calculado aplicando a chave pública na assinatura do hash recebido.

4.7 Lendo OpenPGP

Um objeto OpenPGP pode vir na forma binária ou com Armor. Para poder diferenciar, é preciso olhar o primeiro byte da mensagem: caso seja 0x2D (que é o traço), então a mensagem veio com armor. É possível termos certeza que não é um objeto OpenPGP sem armor, pois o primeiro bit de um pacote OpenPGP é sempre 1. Neste caso, a primeira coisa que o programa que implementamos faz é converter para binário retirando o armor.

Tendo o objeto OpenPGP em binário, o programa pode iniciar a leitura Como o objeto OpenPGP é a concatenação de vários pacotes em uma ordem específica, vão haver casos em que um pacote vai precisar da informação de um pacote anterior, pois sabemos que certos pacotes sempre vem antes de outros. Como o programa não sabe o número de pacotes a priori, então ele continua lendo o próximo pacote enquanto não chegar ao fim do arquivo.

Dado um pacote, os primeiros octetos são o cabeçalho do pacote. No primeiro octeto de cada pacote, o primeiro bit é sempre 1 e o segundo bit indica se o pacote está descrito em um formato novo, ou no formato antigo. Os outros bits do cabeçalho vão depender se o pacote está no formato novo ou antigo, mas eles vão conter o tipo do pacote e o tamanho do corpo do

pacote. Após o cabeçalho, virá o corpo do pacote com o tamanho que foi descrito no cabeçalho e a leitura do corpo vai depender do tipo de pacote.

4.7.1 Formato antigo

Sabe-se que está no formato antigo quando o segundo bit do primeiro octeto do pacote é 0. Nesse caso, o terceiro ao sexto bits vão formar um número de 4 bits (entre 0 e 15) que indica o tipo de pacote, e os 2 próximos bits vão indicar o tipo de tamanho do pacote.

- Tipo de tamanho 00_2: o próximo octeto é o tamanho do corpo pacote.
- Tipo de tamanho 01_2: os 2 próximos octetos são o tamanho do corpo do pacote.
- Tipo de tamanho 10_2: os 4 próximos octetos são o tamanho do corpo do pacote.
- Tipo de tamanho 11_2: o pacote é de tamanho indeterminado, a implementação que deve determinar o tamanho do pacote, se o pacote for um arquivo então o pacote vai até o final do arquivo.

4.7.2 Formato novo

Sabe-se que está no formato novo quando o segundo bit do primeiro octeto do pacote é 1. Nesse caso, o terceiro ao oitavo bits vão formar um número de 6 bits (entre 0 e 63) que indica o tipo de pacote. Note-se que o formato novo aceita mais tipos de pacotes, então certas mensagem só podem ser escritas usando o formato novo. O tamanho do pacote vai ser dado pelos próximos octetos.

4.7.2.1 Tamanho em 1-octeto

O tamanho em 1-octeto é reconhecido pelo fato do primeiro octeto ser menor que 192. Ele vai ser usado quando o corpo do pacote tiver menos que 192 octetos, e esse octeto indica o tamanho do corpo do pacote.

4.7.2.2 Tamanho em 2-octetos

O tamanho em 2-octetos é reconhecido pelo fato do primeiro octeto ser maior ou igual a 192 e menor que 224. Ele suporta pacotes cujo corpo tenha tamanho entre 192 e 8383

octetos. O tamanho do corpo do pacote é dado pela seguinte fórmula (((primeiro octeto - 192) << 8) + segundo octeto + 192).

4.7.2.3 Tamanho em 5-octetos

O tamanho em 5-octetos é reconhecido pelo fato do primeiro octeto ser 255. Ele suporta pacotes cujo corpo tenha tamanho até 4.294.967.295 (0xFFFFFFFF) octetos. O tamanho do pacote é dado pela seguinte fórmula ((segundo octeto << 24) | (terceiro octeto << 16) | (quarto octeto << 8) | quinto octeto).

4.7.2.4 Tamanho Parcial

O tamanho parcial é reconhecido pelo fato do primeiro octeto ser maior ou igual a 224 e menor que 255. O tamanho de cada parte do pacote vai ser uma potência de 2 entre 1 e 1.073.741.824 (2 elevado a 30). Após cada ‘tamanho parcial’, vem uma parte do corpo da mensagem de tamanho igual a 2 elevado a (primeiro octeto - 224) e após cada parte da mensagem vem outro pacote de tamanho parcial, com exceção do último, que não pode ser de tamanho parcial.

4.7.3 Exemplos de cabeçalhos

Os 8 primeiros bits estarão em binário e os próximos em hexadecimal:

- 10110100_2 0x2A: é um pacote no formato antigo cujo tipo é 13 e o corpo da mensagem tem 42 octetos.
- 10100001_2 0x0100: é um pacote no formato antigo cujo tipo é 8 e o corpo da mensagem tem 256 octetos.
- 10001010_2 0x0FFFFF01: é um pacote no formato antigo cujo tipo é 2 e o corpo da mensagem tem 268.435.201 octetos.
- 10011011_2: é um pacote no formato antigo cujo tipo é 6 e o corpo da mensagem vai ser o restante do arquivo.
- 11001101_2 0x2A: é um pacote no formato novo cujo tipo é 13 e o corpo da mensagem tem 42 octetos.

- 11001000_2 0xC040: é um pacote no formato novo cujo tipo é 8 e o corpo da mensagem tem 256 octetos.
- 11000010_2 0xFF0FFFFF01: é um pacote no formato novo cujo tipo é 2 e o corpo da mensagem tem 268.435.201 octetos.
- 11010010_2 0xE6 : é um pacote no formato novo cujo tipo é 18 e o corpo da mensagem vai aparecer em partes. A primeira parte vai ter 64 (2 elevado a 6) octetos. Após os próximos 64 octetos, vai vir a próxima parte com seu cabeçalho.

4.8 Primalidade

Para poder implementar o OpenPGP, uma coisa que tinha que ser feita era gerar uma chave RSA válida e, para isso, era necessário gerar números primos. Para gerar um número primo, foi utilizado o seguinte algoritmo:

1. Escolha um número **M** grande de forma aleatória
2. Enquanto **M** for composto, incremente **M** em uma unidade

Então, a cada vez que o número for incrementado, teremos que verificar se M é primo ou composto.

Inicialmente, estava sendo utilizado o algoritmo de primalidade de Lucas, pois esse teste gera um certificado que o número é realmente primo, mas havia o problema de ser necessário conhecer todos os fatores de (**M** - 1), o que poderia demorar um tempo exponencial. Foi feita uma tentativa de contornar esse problema analisando números que (**M** - 1) possua apenas fatores pequenos (menores que um dado **K**). Isto foi suficiente até algum tamanho, mas conforme aumentava-se o número para chegar no tamanho necessário para ser usado no RSA ficava cada vez mais difícil achar números que atendessem à condição. Em uma segunda tentativa, permitia-se que o número tivesse um fator maior que **K**, ou seja, após dividir **M** por todos os fatores menores que **K,** se verifica se o número restante é primo. Isto melhorou bastante os resultados, mas não foi suficiente para conseguirmos números grandes o suficiente que seriam da ordem de 2 elevado a 1024. Ajustando o valor do **K,** o algoritmo só foi capaz de gerar primos da ordem de 2 elevado a 150 em tempo razoável, ficando já bem lento acima disso.

Como não era possível usar o teste de primalidade de Lucas, foi utilizado o teste de primalidade de Miller-Rabin. Este teste não consegue gerar um certificado que o número é primo, mas tem menos de 25% de chance de ser inconclusivo quando a entrada for um

número composto, ou seja, tem mais de 75% de chance de certificar que o número é composto. Desta forma, ao se executar esse teste várias vezes e em todas elas os testes derem inconclusivo, então é plausível se concluir que o número é primo com uma certa confiança(se algum dos testes não tivesse dado inconclusivo, então teria certificado que o número era composto). Como está sendo aplicado o teste 50 vezes, a probabilidade do número ser composto e o algoritmo achar que ele é primo é de 1 em 1267650600228229401496703205376 (2 elevado a 100), sendo assim praticamente garantido que o número é realmente primo.

5 CONCLUSÃO

A criptografia tem sido usada por muitos anos. A necessidade de proteger informações é algo essencial para as pessoas e instituições poderem ter sua privacidade. Mas, por haver pessoas que queiram invadir esta privacidade, também foram surgindo métodos de quebra de criptografia, gerando a necessidade da criação de métodos ainda mais sofisticados de criptografia.

A criação da criptografia de chave assimétrica é um exemplo da evolução da criptografia, pois mudou a forma de lidar com a segurança em relação às chaves de criptografia e descriptografia, pois, enquanto na criptografia de chave simétrica é necessário se proteger a todo custo a chave de criptografia, na criptografia assimétrica qualquer um pode ter a chave de criptografia sem comprometer a segurança da informação.

A criação de assinatura digital veio como um complemento para a criptografia de chave assimétrica. Na criptografia de chave simétrica, apenas quem possui a chave pode criar uma mensagem, então se foi criado um texto criptografado correto, possivelmente a mensagem é verdadeira. Já na criptografia de chave assimétrica, como qualquer um pode ter a chave de criptografia da mensagem, é necessário ter uma forma de autenticar o remetente da mensagem.

O OpenPGP vem fornecer um ferramental para se usar métodos de criptografia e assinatura digital em comunicações feitas via e-mail pela rede, sem que seja preciso de uma rede exclusiva segura. Quando o protocolo é corretamente implementado não é possível quebrar a criptografia e nem criar uma assinatura em nome de terceiros, assim ajudando as pessoas e instituições a poderem mandar mensagens sem terem que se preocupar com alguém lendo essa informação por ter interceptada a mensagem passando pela internet ou que alguém crie uma mensagem em seu nome.

Quanto à parte da implementação, criar um arquivo para testar a aplicação foi muito útil, pois como o código foi sofrendo várias alterações enquanto ia sendo desenvolvido, erros acabavam sendo gerado em uma parte do código que funcionava anteriormente. A detecção rápida desses erros facilitou muito a correção, pois era possível saber que ele havia sido gerado pela alteração que tinha acabado de ser feita. O uso de um repositório git também ajudou, pois a cada commit podia-se ver claramente as alterações que tinham sido feitas, além de ajudar a verificar que os arquivos que tinham sidos gerados corretamente pela

aplicação não sofreram mudanças, o que provavelmente indicaria um erro de lógica na aplicação.

REFERÊNCIAS

Callas, J., Donnerhacke, L., Finney, H., Shaw, D., Thayer, F.: RFC 4880 - OpenPGP Message Format. http://tools.ietf.org/html/rfc4880 (2007)

Página Web `History - OpenPGP`. URL: https://www.openpgp.org/about/history/. Acessada pela última vez em 18/11/2018.

APÊNDICES

APÊNDICE A – Implementação do OpenPGP

A implementação do OpenPGP é formado por 6 arquivos.

- Util.py: Contem algumas funções genéricas usadas no projéto
- S2KOpenPGP.py: Trata da criação, leitura e processamento do S2K.
- RSAOpenPGP.py: Trata da criação, leitura e processamento do RSA.
- myOpenPGP.py: Consiste da classe principal da aplicação.
- OpenPGPExceptions.py: Define as exceções geradas pela aplicação.
- testMyOpenPGP.py: Contem vários testes para verificar a corretude da aplicação.

File Name: Util.py

```
import binascii
import random
import time
import hashlib
myRandInt = random.SystemRandom().randint
localTest = False
if localTest:
	random.seed(0)
	myRandInt = random.randint

def powMod(b, e, mod):
	o = 1
	a = b
	while e > 0:
		if e%2 == 1:
			o = (o * a) % mod
		a = (a*a)%mod
		e //= 2
	return o

def invMod(b, mod):
	x, y, g = gcdE(b, mod)
	return (x+mod)%mod if g == 1 else 0

def gcdE(a, b):
	x0, y0 = 1, 0
	x1, y1 = 0, 1
	while b != 0:
		x0, x1 = x1, x0 - x1*(a//b)
		y0, y1 = y1, y0 - y1*(a//b)
		a, b = b, a%b
	return x0, y0, a

def blockSize(algo):
	# 9.2.  Symmetric-Key Algorithms
	if algo == 7 or algo == 8 or algo == 9:
		return 16
	else:
		raise OpenPGPException('9.2. Symmetric-Key Algorithms: ' + algo)

def toint(str256):
	return reduce(lambda x,y:x*256+ord(y), str256, 0)

def SampleChecksum(data):
	return reduce(lambda x,y:(x+ord(y))%65536, data, 0)

def leMPI(data, p):
	length = (toint(data[p: p + 2]) + 7) / 8
	p += 2
	mpi = data[p: p + length]
	p += length
	return (p, mpi)

def toMPI(data):
	length = len(data)*8
	p = 0
	while data[p] == chr(0):
		p += 1
		length -= 8
	aux = 128
	while aux > ord(data[p]):
		aux /= 2
```

```
                length -= 1
        return int2str256(length, 2) + data[p:]

def int2str256(longInt, length):
        if length == 0:
                s = "{0:x}".format(longInt)
                if len(s) % 2 == 1:
                        s = '0' + s
        else:
                s = "{0:0{1}x}".format(longInt, length*2)
        return binascii.unhexlify(s)

def EME_PKCS1_v1_5_DECODE(EM):
        #13.1.2.  EME-PKCS1-v1_5-DECODE
        p = EM.find(chr(0), 1)
        if p <= 8 or EM[0] != chr(0) or EM[1] != chr(2):
                raise OpenPGPException('>>> EME-PKCS1-v1_5-DECODE decryption error <<<')
        return EM[p+1:]

def EME_PKCS1_v1_5_ENCODE(M, k):
        #13.1.1.  EME-PKCS1-v1_5-ENCODE
        psLen = k - len(M) - 3
        if psLen < 8:
                raise OpenPGPException('>>> EME-PKCS1-v1_5-ENCODE message too long <<<')
        PS = ''.join(chr(myRandInt(1,255)) for i in range(psLen))
        return chr(0) + chr(2) + PS + chr(0) + M

def EMSA_PKCS1_v1_5(M, hashAlgo, length):
        #13.1.3.  EMSA-PKCS1-v1_5
        T = ASN_1_DER(hashAlgo)
        psLen = length - 3 - len(T) - len(M)
        if psLen < 8:
                raise OpenPGPException('>>> EMSA-PKCS1-v1_5 intended encoded message length
too short <<<')
        return chr(0) + chr(1) + chr(0xff)*psLen + chr(0) + T + M

def ASN_1_DER(hashAlgo):
        if hashAlgo == 8:
                return binascii.unhexlify('3031300d060960864801650304020105000420')
        else:
                raise OpenPGPException('5.2.2. Version 3 Signature Packet Format: ' +
hashAlgo)

def TIMENOW():
        if localTest:
                return '\x5a\x49\x96\x21'
        else:
                return int2str256(int(time.time()), 4)

def hashAlgo(algo):
        if algo == 8:
                return hashlib.sha256
        elif algo == 2:
                return hashlib.sha1
        else:
                raise OpenPGPException('9.4. Hash Algorithms: ' + algo)

def display(msg, str256):
        print msg + ':', binascii.hexlify(str256)

def randOctets(n):
        return ''.join(chr(myRandInt(0,255)) for i in range(n))
```

File Name: S2kOpenPGP.py

```
import hashlib
import Util
from OpenPGPExceptions import *

class S2kOpenPGP:
	def read(self, data, p):
		#3.7.1.  String-to-Key (S2K) Specifier Types
		p0 = p
		self.version = ord(data[p])
		p += 1
		if self.version != 0 and self.version != 1 and self.version != 3:
			raise OpenPGPVersionException('S2kOpenPGP', self.version, [0, 1, 3])

		self.hashAlgo = ord(data[p])
		p += 1

		if self.version == 1 or self.version == 3:
			self.salt = data[p: p + 8]
			p += 8

		if self.version == 3:
			coded = ord(data[p])
			self.count = (16 + (coded & 15)) << ((coded >> 4) + 6)
			p += 1

		self.packet = data[p0: p]
		return p

	def generate(self):
		self.version = 3
		self.hashAlgo = 2
		self.salt = Util.randOctets(8)
		coded = 238#0xee
		self.count = 31457280#(16 + (coded & 15)) << ((coded >> 4) + 6)

		self.packet = (chr(self.version)
			+ chr(self.hashAlgo)
			+ self.salt
			+ chr(coded))
		return self

	def makeKey(self, bs, passphrase):
		if self.version != 3:
			raise OpenPGPException('Not Implemented yet makeKey S2K version: ' +
hex(self.version))
		#3.7.1.3.  Iterated and Salted S2K
		comb = self.salt+passphrase
		while len(comb) < self.count:
			comb += comb
		comb = comb[:self.count]

		if self.hashAlgo == 1:
			#MD5
			raise OpenPGPException('Not Implemented s2k MD5')
		elif self.hashAlgo == 2:
			#SHA1
			hd = hashlib.sha1(comb).digest()
		elif self.hashAlgo == 3:
			#RIPE-MD/160
			raise OpenPGPException('Not Implemented s2k RIPE-MD/160')
		elif self.hashAlgo == 8:
			#SHA256
```

```
                        raise OpenPGPException('Not Implemented s2k SHA256')
                elif self.hashAlgo == 9:
                        #SHA384
                        raise OpenPGPException('Not Implemented s2k SHA384')
                elif self.hashAlgo == 10:
                        #SHA512
                        raise OpenPGPException('Not Implemented s2k SHA512')
                elif self.hashAlgo == 11:
                        #SHA224
                        raise OpenPGPException('Not Implemented s2k SHA224')
                else:
                        raise OpenPGPNotValidException('Hash Algorithms', self.hashAlgo, [1,
2, 3, 8, 9, 10, 11])

                if len(hd) < bs:
                        print '''need multiples hashes to make key'''
                        raise OpenPGPException('3.7.1. String-to-Key (S2K) Specifier Types')
                return hd[:bs]
```

File Name: RSAOpenPGP.py

```
from Crypto.Cipher import AES
import hashlib
import Util
from S2kOpenPGP import S2kOpenPGP
from OpenPGPExceptions import *

class RSAOpenPGP:
	def read(self, body):
		#5.5.2.  Public-Key Packet Formats//Tag 6 or Tag 14
		self.version = ord(body[0])
		p = 1
		if self.version == 3:
			raise OpenPGPException('5.5.2. Public-Key Packet Formats: ' +
self.version)
		elif self.version == 4:
			self.dateCreated = body[p: p+4]
			p += 4
			self.publicKeyAlgo = ord(body[p])
			p += 1
			#9.1.  Public-Key Algorithms
			if self.publicKeyAlgo == 1 or self.publicKeyAlgo == 2 or
self.publicKeyAlgo == 3:
				#rsa
				p, self.nStrRSA = Util.leMPI(body, p)
				self.nRSA = Util.toint(self.nStrRSA)
				self.messegeLen = len(self.nStrRSA)
				p, self.eStrRSA = Util.leMPI(body, p)
				self.eRSA = Util.toint(self.eStrRSA)

				self.packet = chr(0x99) + Util.int2str256(p, 2) + body[:p]
				self.fingerPrint = hashlib.sha1(self.packet).digest()
				self.keyId = self.fingerPrint[-8:]

				self.subKeys = []
				self.readed = False

				if p == len(body):
					self.hasSecretData = False
				else:
					self.readSecret(body[p:])
			elif publicKeyAlgo == 16:
				#Elgamal
				p, pDSA = Util.leMPI(body, p)
				p, gDSA = Util.leMPI(body, p)
				p, yDSA = Util.leMPI(body, p)
			elif publicKeyAlgo == 17:
				#DSA
				p, pDSA = Util.leMPI(body, p)
				p, qDSA = Util.leMPI(body, p)
				p, gDSA = Util.leMPI(body, p)
				p, yDSA = Util.leMPI(body, p)
			else:
				raise OpenPGPException('publicKeyAlgo ' + publicKeyAlgo + 'not
suported')
		else:
			raise OpenPGPVersionException('Public key paket', self.version, [3,
4])
		return self

	def readSecret(self, body):
		#5.5.3.  Secret-Key Packet Formats//Tag 5 or Tag 7
		self.hasSecretData = True
```

```
        self.s2kConventions = ord(body[0])
        p = 1
        if self.s2kConventions == 254 or self.s2kConventions == 255:
                self.symEncAlgo = ord(body[p])
                p += 1
                #9.2.  Symmetric-Key Algorithms

                self.s2k = S2kOpenPGP()
                p = self.s2k.read(body, p)

                bs = Util.blockSize(self.symEncAlgo)
                self.IV = body[p: p + bs]
                p += bs

                self.encrData = body[p:]

    def insertSecretData(self, anotherRSA):
        self.hasSecretData = True
        self.s2kConventions = anotherRSA.s2kConventions
        if self.s2kConventions == 254 or self.s2kConventions == 255:
                self.symEncAlgo = anotherRSA.symEncAlgo
                self.s2k = anotherRSA.s2k
                self.IV = anotherRSA.IV
                self.encrData = anotherRSA.encrData

    def generate(self, passphrase):
        self.version = 4
        self.publicKeyAlgo = 1
        self.readed = True
        self.hasSecretData = True
        self.subKeys = []
        self.s2kConventions = 254#0xfe
        self.symEncAlgo = 7
        bs = Util.blockSize(self.symEncAlgo)

        while True:
                self.pRSA = nextPrime(Util.myRandInt((9<<1020), (11<<1020)-1))
                self.qRSA = nextPrime(Util.myRandInt((13<<1020), (15<<1020)-1))
                self.nRSA = self.pRSA*self.qRSA
                self.eRSA = 65537#0x0011010001
                self.uRSA = Util.invMod(self.pRSA, self.qRSA)
                if self.uRSA == 0:
                        continue;
                self.dRSA = Util.invMod(self.eRSA, (self.pRSA-1)*(self.qRSA-1))
                if self.dRSA == 0:
                        continue;
                break

        self.pStrRSA = Util.int2str256(self.pRSA, 0)
        self.qStrRSA = Util.int2str256(self.qRSA, 0)
        self.nStrRSA = Util.int2str256(self.nRSA, 0)
        self.eStrRSA = Util.int2str256(self.eRSA, 0)
        self.dStrRSA = Util.int2str256(self.dRSA, 0)
        self.uStrRSA = Util.int2str256(self.uRSA, 0)

        self.messegeLen = len(self.nStrRSA)
        self.dateCreated = Util.TIMENOW()

        self.s2k = S2kOpenPGP().generate()
        self.IV = Util.randOctets(bs)

        data = (Util.toMPI(self.dStrRSA)
                + Util.toMPI(self.pStrRSA)
                + Util.toMPI(self.qStrRSA)
                + Util.toMPI(self.uStrRSA))
        data += hashlib.sha1(data).digest()
```

```
                symKey = self.s2k.makeKey(bs, passphrase)
                lack = bs - len(data)%bs

                self.encrData = AES.new(symKey, AES.MODE_CFB, self.IV, segment_size =
128).encrypt(data + ' '*lack)[:-lack]

                body = (chr(self.version)
                        + self.dateCreated
                        + chr(self.publicKeyAlgo)
                        + Util.toMPI(self.nStrRSA)
                        + Util.toMPI(self.eStrRSA))
                self.packet = chr(0x99) + Util.int2str256(len(body), 2) + body
                self.fingerPrint = hashlib.sha1(self.packet).digest()
                self.keyId = self.fingerPrint[-8:]
                return self

        def leSecretData(self, passphrase):
                if not self.hasSecretData:
                        raise OpenPGPException('not has the private key for this message')

                bs = Util.blockSize(self.symEncAlgo)
                symKey = self.s2k.makeKey(bs, passphrase)
                lack = bs - len(self.encrData)%bs

                # 9.2.  Symmetric-Key Algorithms
                if self.symEncAlgo == 7:
                        # AES with 128-bit key
                        data = AES.new(symKey, AES.MODE_CFB, self.IV, segment_size =
128).decrypt(self.encrData + ' '*lack)[:-lack]
                else:
                        raise OpenPGPException('9.2. Symmetric-Key Algorithms: ' +
self.symEncAlgo)

                if hashlib.sha1(data[:-20]).digest() != data[-20:]:
                        raise OpenPGPIncorrectException('passphrase', 'sha1',
hashlib.sha1(data[:-20]).digest(), data[-20:])

                p, self.dStrRSA = Util.leMPI(data, 0)
                p, self.pStrRSA = Util.leMPI(data, p)
                p, self.qStrRSA = Util.leMPI(data, p)
                p, self.uStrRSA = Util.leMPI(data, p)

                self.dRSA = Util.toint(self.dStrRSA)
                self.pRSA = Util.toint(self.pStrRSA)
                self.qRSA = Util.toint(self.qStrRSA)
                self.uRSA = Util.toint(self.uStrRSA)
                self.readed = True

        def decodeRSA(self, mRSA, passphrase):
                if not self.readed:
                        self.leSecretData(passphrase)
                MM = Util.powMod(Util.toint(mRSA), self.dRSA, self.nRSA)
                return  Util.int2str256(MM, self.messegeLen)

        def encodeRSA(self, MM):
                mRSA = Util.powMod(Util.toint(MM), self.eRSA, self.nRSA)
                return  Util.int2str256(mRSA, self.messegeLen)

        def unsignRSA(self, MM):
                mRSA = Util.powMod(Util.toint(MM), self.eRSA, self.nRSA)
                return  Util.int2str256(mRSA, self.messegeLen)

        def signRSA(self, mRSA, passphrase):
                if not self.readed:
                        self.leSecretData(passphrase)
```

```
                MM = Util.powMod(Util.toint(mRSA), self.dRSA, self.nRSA)
                return  Util.int2str256(MM, self.messegeLen)

def nextPrime(x):
        while isCompSimple(x) or isCompMiller(x):
                x += 1
        return x

def isCompSimple(n):
        p = [2,3,5,7,11,13,17,19]
        for x in p:
                if n % x == 0:
                        return n != x
        return False

def isCompMiller(n):
        if n < 4:
                return False
        m = n-1
        k = 0
        while m % 2 == 0:
                m //= 2
                k += 1
        for i in range(50):
                b = Util.myRandInt(2, n - 2)
                r = Util.powMod(b, m, n)
                if r == 1:
                        continue
                for i in range(k):
                        if r == n-1:
                                break
                        r = (r*r) % n
                else:
                        return True
        return False
```

File Name: myOpenPGP.py

```
import binascii
from Crypto.Cipher import AES
from datetime import datetime
import hashlib
import base64
import zlib
import time
from getpass import getpass
import os.path
import logging

import Util
from RSAOpenPGP import RSAOpenPGP
from OpenPGPExceptions import *

import testMyOpenPGP

class myOpenPGP:
	def __init__(self):
		self.asymmetricKeys = []
		self.keyId = ''
		self.asymKey = ''
		self.asymSubKey = ''

	def start(self):
		allOptions = ['-a', '--armor', '-i', '--ignore', '-o', '--output', '-c', '--
compress', '-p', '--pass']
		while True:
			try:
				commandInput = raw_input("Enter the command:").strip()
				if commandInput == '':
					continue
				if commandInput[0] == "\"":
					raise OpenPGPException("The command cannot start with
\"(quotation marks)")
				commandInput = commandInput.split("\"")
				commandList = filter(lambda x: x != '',
commandInput[0].split(" "))
				p = 1
				while p < len(commandInput):
					commandList += [commandInput[p]]
					if p+1 >= len(commandInput):
						raise OpenPGPException("Some \"(quotation
marks) was not closed")
					if commandInput[p+1].strip() != "":
						commandList += filter(lambda x: x != '',
commandInput[p+1].split(" "))
					p += 2
				cLen = len(commandList)
				command = commandList[0].lower()

				if command == 'exit':
					break;
				elif command == 'generatekey' or command == '-g':
					algo = raw_input("Algorithm?") if cLen <= 1 else
commandList[1]
					if algo.lower() != 'rsa':
						raise OpenPGPException('Only RSA Avaliable')
					name = raw_input("User Name?") if cLen <= 2 else
commandList[2]
					email = raw_input("User E-mail?") if cLen <= 3 else
commandList[3]
```

```
                                passphrase = getpass("Passphase?") if cLen <= 4 else
commandList[4]
                                if cLen > 5:
                                        raise OpenPGPException('There are more
arguments than expected')

                                self.generateKeyRSA(name + " <" + email + ">",
passphrase)
                        elif command == 'readfile' or command == '-r':
                                fileName = raw_input("File Name?") if cLen <= 1 else
commandList[1]
                                if cLen > 2:
                                        raise OpenPGPException('There are more
arguments than expected')

                                if os.path.isfile(fileName):
                                        self.readFile(open(fileName, "rb").read())
                                else:
                                        raise OpenPGPException(fileName + " not is a
valid file.")
                        elif command == 'help' or command == '-h':
                                if cLen > 1:
                                        print 'Not yet implemented the more detailed
help for command', command
                                        print 'Here is the basic help:',
                                print 'Commands:'
                                print '(GenerateKey | -g) [Cryptosystem [UserName [e-
mail [passphase]]]]'
                                print '(ReadFile | -r) [FileName]'
                                print '(Encrypt | -e) FileName [User] [Options...]'
                                print '(Sign | -s) FileName [User] [Options...]'
                                print '(Export-Key | -pk) [User] [Options...]'
                                print '(Export-Secret-Key | -sk) [User] [Options...]'
                                print '(Help | -h) [command]'
                                print 'Exit'
                                print ''
                                print 'Argumentos:'
                                print 'Cryptosystem: Cryptosystem used to generate the
key, only "rsa" avaliable'
                                print 'UserName: name of the owner of the key'
                                print 'e-mail: e-mail of the owner of the key'
                                print 'passphase: sequence of words to protect from
other people using the key'
                                print 'FileName: File to be processed'
                                print 'User: UserName or e-mail'
                                print 'Options: one or more of these arguments'
                                print '  (--armor | -a): Convet the output to Base64'
                                print '  (--ignore | -i): Do not ask if you want
replace the output file(always replace)'
                                print '  (--output | -o) OutputFile: Set OutputFile as
name to save the generated packet'
                                print '  (--compress | -c) compression: The compression
to be used, only "zip" available'
                                print '  (--pass | -p) passphrase: Passphrase to use
the secret key'
                                print ''
                                print 'Obs: passphrase and user containing spaces need
to be surrounded by \"(quotation marks)'
                        else:
                                if command == 'encrypt' or command == '-e' or command
== 'sign' or command == '-s':
                                        if cLen <= 1:
                                                raise OpenPGPException('Command ' +
command + ' need be folowed by FileName')
                                        inputFile = commandList[1]
                                        if not os.path.isfile(inputFile):
```

```
                                        raise OpenPGPException(inputFile + " not
is a valid file")
                                p = 2
                        elif command == 'export-key' or command == '-pk' or
command == 'export-secret-key' or command == '-sk':
                                p = 1
                        else:
                                raise OpenPGPException('Command ' + command + '
is not valid')

                        user = ''
                        if p < cLen and len(commandList[p]) > 0 and
commandList[p][0] != '-':
                                user = commandList[p]
                                p += 1

                        armor = False
                        outputFile = ''
                        compress = 0
                        passphrase = ''
                        confirm = True
                        while p < cLen:
                                if commandList[p] not in allOptions:
                                        raise OpenPGPException('Option ' +
commandList[p] + ' is not valid')
                                elif commandList[p] == '-a' or commandList[p]
== '--armor':
                                        armor = True
                                        p += 1
                                elif commandList[p] == '-i' or commandList[p]
== '--ignore':
                                        confirm = False
                                        p += 1
                                else:
                                        if p+1 >= cLen:
                                                raise OpenPGPException("Missing
argument after " + commandList[p])
                                        elif commandList[p] == '-o' or
commandList[p] == '--output':
                                                outputFile = commandList[p+1]
                                                if outputFile.endswith('.gpg') or
outputFile.endswith('.asc'):
                                                        outputFile = outputFile[:
-4]
                                        elif commandList[p] == '-c' or
commandList[p] == '--compress':
                                                compress = commandList[p+1]
                                                if compress == "zip":
                                                        compress = 1
                                                elif compress == "zlib":
                                                        compress = 2
                                                elif compress == "bzip2":
                                                        compress = 3
                                                else:
                                                        raise
OpenPGPException('compress must be zip, zlib or bzi2')
                                        elif commandList[p] == '-p' or
commandList[p] == '--pass':
                                                passphrase = commandList[p + 1]
                                        p += 2


                        if command == 'encrypt' or command == '-e':
                                self.encrypt(inputFile, user, outputFile,
armor, compress, confirm)
                        elif command == 'sign' or command == '-s':
```

```
                                        self.signFile(inputFile, user, outputFile,
passphrase, armor, compress, confirm)
                                elif command == 'export-key' or command == '-pk':
                                        self.savePublicKey(user, outputFile,
passphrase, armor, compress, confirm)
                                elif command == 'export-secret-key' or command == '-
sk':
                                        self.savePrivateKey(user, outputFile,
passphrase, armor, compress, confirm)
                        except OpenPGPException as e:
                                print e
                                print 'List of commands:'
                                print '  GenerateKey or -g'
                                print '  ReadFile or -r'
                                print '  Encrypt or -e'
                                print '  Sign or -s'
                                print '  Export-Key or -pk'
                                print '  Export-Secret-Key or -sk'
                                print '  Help or -h'
                                print '  Exit'
                        except Exception as e:
                                print e
                                logging.exception("Something awful happened!")
                        print ''

        def setAsymmetricKeys(self, asymmetricKeys, asymKey = None, asymSubKey = None):
                self.asymmetricKeys = asymmetricKeys
                self.asymKey = asymKey
                self.asymSubKey = asymSubKey
                return self

        def generateKeyRSA(self, userId, passphrase):
                print 'Generating Key RSA for user:', userId
                asymmetricKey = RSAOpenPGP().generate(passphrase)
                asymmetricKey.userId = userId
                asymmetricKey.subKeys.append(RSAOpenPGP().generate(passphrase))
                self.asymmetricKeys.append(asymmetricKey)
                print 'Key RSA generated with success'
                return self

        def write_secretKeyPaket(self, keyIndex = -1, subKeyIndex = None):
                #5.5.3.  Secret-Key Packet Formats//Tag 5 or Tag 7
                return (self.write_publicKeyPaket(keyIndex, subKeyIndex)
                        + chr(self.asymSubKey.s2kConventions)
                        + chr(self.asymSubKey.symEncAlgo)
                        + self.asymSubKey.s2k.packet
                        + self.asymSubKey.IV
                        + self.asymSubKey.encrData)

        def write_publicKeyPaket(self, keyIndex = -1, subKeyIndex = None):
                #5.5.2.  Public-Key Packet Formats//Tag 6 or Tag 14
                self.asymKey = self.asymmetricKeys[keyIndex]
                self.asymSubKey = self.asymKey.subKeys[subKeyIndex] if subKeyIndex != None
else self.asymKey

                return (chr(self.asymSubKey.version)
                        + self.asymSubKey.dateCreated
                        + chr(self.asymSubKey.publicKeyAlgo)
                        + Util.toMPI(self.asymSubKey.nStrRSA)
                        + Util.toMPI(self.asymSubKey.eStrRSA))

        def allKeys(self):
                for asymKey in self.asymmetricKeys:
                        yield asymKey
                        for asymSubKey in asymKey.subKeys:
                                yield asymSubKey
```

```
	def read_Public_Key_Encrypted_Session_Key_Packets(self, p):
		#5.1.  Public-Key Encrypted Session Key Packets (Tag 1)
		version = ord(self.encodedFile[p])
		p += 1
		if version == 3:
			keyId = self.encodedFile[p: p + 8]
			p += 8
			print 'keyId', binascii.hexlify(keyId)
			publicKeyAlgo = ord(self.encodedFile[p])
			p += 1
			#9.1.  Public-Key Algorithms
			if publicKeyAlgo == 1 or publicKeyAlgo == 2 or publicKeyAlgo == 3:
				#rsa
				p, mRSA = Util.leMPI(self.encodedFile, p)


				numTry = 0
				for asymKey in self.allKeys():
					if asymKey.keyId != keyId:
						continue
					numTry = numTry + 1
					MM = asymKey.decodeRSA(mRSA, 'this is a pass')
					MM = Util.EME_PKCS1_v1_5_DECODE(MM)
					if MM == "" or Util.SampleChecksum(MM[1:-2]) !=
Util.toint(MM[-2:]):
						continue
					self.symAlgo = ord(MM[0])
					#print('algo', self.symAlgo)
					self.symKey = MM[1:-2]# key to be used by the
symmetric-key algorithm
					#print('self.symKey', binascii.hexlify(self.symKey))
					break
				else:
					if numTry > 0:
						raise OpenPGPException('>>> checksum of
symmetric-key does not match <<<')
					print 'self.keyId',binascii.hexlify(keyId)
					print '>>> not has the key for this criptografy,
len(self.asymmetricKeys)',len(self.asymmetricKeys)
			elif publicKeyAlgo == 16:
				#Elgamal
				raise OpenPGPException('5.5.2. Public-Key Packet Formats
Elgamal public key: ' + publicKeyAlgo)
			elif publicKeyAlgo == 17:
				#DSA
				raise OpenPGPException('5.5.2. Public-Key Packet Formats DSA
public key: ' + publicKeyAlgo)
			else:
				raise OpenPGPNotValidException('Public Key Algo',
publicKeyAlgo, [1, 2, 3, 16, 17])
		else:
			raise OpenPGPVersionException('Encrypted Session Key', version, [3])
		return p

	def write_Public_Key_Encrypted_Session_Key_Packets(self):
		#5.1.  Public-Key Encrypted Session Key Packets (Tag 1)
		version = chr(3)

		self.symKey = Util.randOctets(32)
		checkSum = Util.SampleChecksum(self.symKey)
		self.symAlgo = 9

		MM = chr(self.symAlgo) + self.symKey + Util.int2str256(checkSum, 2)
		MM = Util.EME_PKCS1_v1_5_ENCODE(MM, self.asymKey.messegeLen)
		mRSA = self.asymKey.encodeRSA(MM)
```

```
                return (version
                        + self.asymKey.keyId
                        + chr(self.asymKey.publicKeyAlgo)
                        + Util.toMPI(mRSA))

        def read_SymEncryptedIntegrityProtectedDataPacket(self, p, pEnd):
                #5.13.  Sym. Encrypted Integrity Protected Data Packet (Tag 18)
                version = ord(self.encodedFile[p])
                p += 1
                if version == 1:
                        encrData = self.encodedFile[p: pEnd]
                        p = pEnd
                        if self.symAlgo == 7 or self.symAlgo == 8 or self.symAlgo == 9:
                                bs = Util.blockSize(self.symAlgo)
                                lack = bs - len(encrData)%bs
                                data = AES.new(self.symKey, AES.MODE_CFB, chr(0)*bs,
segment_size = 128).decrypt(encrData + ' '*lack)[:-lack]
                        else:
                                raise OpenPGPException('9.3. Compression Algorithms: ' +
self.symAlgo)
                        if data[14:16] != data[16:18]:
                                raise OpenPGPIncorrectException('session key', 'repetition
bits', data[16:18], data[14:16])
                        print 'Reading myOpenPGP Protected Data Packet'

        myOpenPGP().setAsymmetricKeys(self.asymmetricKeys).readFile(data[18:], data[:18])
                else:
                        raise OpenPGPVersionException('Sym. Encrypted Integrity Protected
Data Packet', version, [1])
                return p

        def write_SymEncryptedIntegrityProtectedDataPacket(self, tags):
                #5.13.  Sym. Encrypted Integrity Protected Data Packet (Tag 18)
                version = chr(1)

                IV = ''.join(chr(Util.myRandInt(0,255)) for i in range(16))
                IV += IV[-2:]

                data = IV + myOpenPGP().writeFile(tags, IV).encodedFile

                bs = Util.blockSize(self.symAlgo)
                lack = bs - len(data)%bs
                encrData = AES.new(self.symKey, AES.MODE_CFB, chr(0)*bs, segment_size =
128).encrypt(data + ' '*lack)[:-lack]
                return version + encrData

        def read_LiteralDataPacket(self, p, pEnd):
                #5.9.  Literal Data Packet (Tag 11)
                formatted = self.encodedFile[p]
                p += 1
                if formatted == 'b':
                        fileNameLen = ord(self.encodedFile[p])
                        p += 1
                        fileName = self.encodedFile[p:p+fileNameLen]
                        p += fileNameLen
                        if fileName == "_CONSOLE":
                                print '''5.9. Literal Data Packet (Tag 11)//for your eyes
only'''
                        if self.encodedFile[p: p+4] != chr(0)*4:
                                dateCreated =
datetime.fromtimestamp(Util.toint(self.encodedFile[p: p+4])).strftime('%H:%M:%S %d/%m/%Y')
                        else:
                                dateCreated = 'whithout date'
                        p += 4
                        self.paketStart = p
```

```
                        self.paketEnd = pEnd
                        literalData = self.encodedFile[p:pEnd]
                        p = pEnd
                        print 'file:', (fileName, dateCreated, literalData)
                        if os.path.isfile(fileName):
                                print 'Already exists the file', fileName, ', the file will
not be verified.'
                        elif fileName != "_CONSOLE":
                                open(fileName, "w").write(literalData)
                                print 'The data was saved in file', fileName
                elif formatted == 't':
                        raise OpenPGPException('Not Implemented format "t" for Literal Data
Packet')
                elif formatted == 'u':
                        raise OpenPGPException('Not Implemented format "u" for Literal Data
Packet')
                else:
                        raise OpenPGPNotValidException('Literal Data Packet format',
formatted, ['b', 't', 'u'])
                return p

        def write_LiteralDataPacket(self, fileName):
                #5.9.  Literal Data Packet (Tag 11)
                formatted = 'b'
                fileNameLen = chr(len(fileName))
                date = chr(0)*4
                self.literalData = open(fileName, "r").read()

                return (formatted
                        + fileNameLen
                        + fileName
                        + date
                        + self.literalData)

        def read_ModificationDetectionCodePacket(self, p):
                #5.14.  Modification Detection Code Packet (Tag 19)
                sha = hashlib.sha1(self.extraParam + self.encodedFile[:p]).digest()
                if sha != self.encodedFile[p:]:
                        raise OpenPGPException('Detected Modification on Packet')
                return p+20

        def write_ModificationDetectionCodePacket(self):
                #5.14.  Modification Detection Code Packet (Tag 19)
                return hashlib.sha1(self.extraParam + self.encodedFile + chr(20)).digest()

        def read_UserIDPacket(self, p, pEnd):
                # 5.11.  User ID Packet (Tag 13)
                self.asymKey.userId = self.encodedFile[p:pEnd]
                print 'userId:', self.asymKey.userId
                return pEnd

        def write_UserIDPacket(self):
                # 5.11.  User ID Packet (Tag 13)
                return self.asymKey.userId

        def leSubPacket(self, p, pEnd):
                dataSet = {}
                while p != pEnd:
                        stOctet = ord(self.encodedFile[p])
                        p += 1
                        if stOctet < 192:
                                length = stOctet
                        elif stOctet < 255:
                                ndOctet = ord(self.encodedFile[p])
                                p += 1
                                length = (stOctet - 192 << 8) + ndOctet + 192
```

```
                else:
                        length = Util.toint(self.encodedFile[p: p + 4])
                        p += 4
                subpacketType = ord(self.encodedFile[p])
                subpacketData = self.encodedFile[p+1: p+length]
                p += length
                dataSet[subpacketType] = subpacketData
        return dataSet

    def makeSubPacket(self, dataSet):
        out = ''
        for subpacketType in dataSet:
                length = self.len2NewFormat(1 + len(dataSet[subpacketType]))
                out += length + chr(subpacketType) + dataSet[subpacketType]
        return out

    def read_SignaturePacket(self, p, pEnd):
        # 5.2.  Signature Packet (Tag 2)
        pv = p
        version = ord(self.encodedFile[p])
        p += 1
        if version == 3:
                raise OpenPGPException('5.2.2. Version 3 Signature Packet Format')
        elif version == 4:
                signatureType = ord(self.encodedFile[p])
                p += 1

                publicKeyAlgo = ord(self.encodedFile[p])
                p += 1

                hashAlgoId = ord(self.encodedFile[p])
                p += 1
                hashAlgo = Util.hashAlgo(hashAlgoId)

                hashedSubpacketLen = Util.toint(self.encodedFile[p: p+2])
                p += 2

                hashedSubpacket = self.leSubPacket(p, p+hashedSubpacketLen)
                p += hashedSubpacketLen

                ph = p

                unhashedSubpacketLen = Util.toint(self.encodedFile[p: p+2])
                p += 2

                unhashedSubpacket = self.leSubPacket(p, p+unhashedSubpacketLen)
                p += unhashedSubpacketLen

                signedHashValue = self.encodedFile[p: p+2]
                p += 2

                while p != pEnd:
                        p, mm = Util.leMPI(self.encodedFile, p)

                        #9.1.  Public-Key Algorithms
                        if publicKeyAlgo == 1 or publicKeyAlgo == 2 or publicKeyAlgo
== 3:
                                #rsa
                                if signatureType == 0x18:
                                        sig = self.asymKey.packet
                                        sig += self.asymSubKey.packet

                                        asymKeys = [self.asymKey]
                                elif signatureType == 0x13:
                                        sig = self.asymKey.packet
```

```
                                        sig += binascii.unhexlify('b4' +
'{0:0{1}x}'.format(len(self.asymKey.userId), 8))
                                        sig += self.asymKey.userId

                                        asymKeys = [self.asymKey]
                                elif signatureType == 0x00:
                                        sig =
self.encodedFile[self.paketStart:self.paketEnd]

                                        asymKeys = []
                                        for asymKey in self.allKeys():
                                                if asymKey.keyId == self.keyId:
                                                        asymKeys.append(asymKey)
                                elif signatureType == 0x10:
                                        sig = self.asymKey.packet
                                        sig += binascii.unhexlify('b4' +
'{0:0{1}x}'.format(len(self.asymKey.userId), 8))
                                        sig += self.asymKey.userId

                                        asymKeys = [self.asymKey]

                                        if 16 in unhashedSubpacket:
                                                keyId = unhashedSubpacket[16]
                                                asymKeys = []
                                                for asymKey in self.allKeys():
                                                        if asymKey.keyId == keyId:
                                                                asymKeys.append(asymKey)
                                else:
                                        raise OpenPGPException('Not Implemented yet
signatureType: ' + hex(signatureType))

                                sig += self.encodedFile[pv:ph]
                                sig += binascii.unhexlify("04ff")
                                sig += binascii.unhexlify('{0:0{1}x}'.format(ph-pv, 8))


                                hld = hashAlgo(sig).digest()
                                if hld[:2] != signedHashValue:
                                        raise OpenPGPIncorrectException('left 16 bits
of signed hash', hashAlgo.__name__, hld[:2], signedHashValue)
                                if len(asymKeys) == 0:
                                        keyId = self.keyId if 16 not in
unhashedSubpacket else unhashedSubpacket[16]
                                        raise OpenPGPKeyIdException(keyId,
len(self.asymmetricKeys))
                                elif len(asymKeys) > 1:
                                        raise OpenPGPException('>>> has multiples
possibilities of key for this signature <<<' + len(self.asymmetricKeys))
                                else:
                                        mm2 = asymKeys[0].unsignRSA(mm)[-len(hld):]
                                        if hld != mm2:
                                                raise OpenPGPIncorrectException('signed
hash', hashAlgo.__name__, hld, mm2)
                                        else:
                                                print 'Signature', hex(signatureType),
'validated with success:', binascii.hexlify(mm2)
                        elif publicKeyAlgo == 16:
                                #Elgamal
                                raise OpenPGPException('5.5.2. Public-Key Packet
Formats Elgamal public key')
                        elif publicKeyAlgo == 17:
                                #DSA
                                raise OpenPGPException('5.5.2. Public-Key Packet
Formats DSA public key')
                        else:
```

```
                                raise OpenPGPNotValidException('Public Key Algo',
publicKeyAlgo, [1, 2, 3, 16, 17])
                else:
                        raise OpenPGPVersionException('Signature Packet', version, [3, 4])
                return p

        def write_SignaturePacket(self, signatureType, hashAlgoId, passphrase):
                # 5.2.  Signature Packet (Tag 2)
                version = chr(4)
                hashedSubpacket = self.makeSubPacket({33: chr(4) + self.asymKey.fingerPrint,
2: Util.TIMENOW()})#'Z\xb9\x9c?'#Util.int2str256(int(time.time()), 4)
                hashedSubpacketLen = Util.int2str256(len(hashedSubpacket), 2)

                fistPart = (version
                        + chr(signatureType)
                        + chr(self.asymKey.publicKeyAlgo)
                        + chr(hashAlgoId)
                        + hashedSubpacketLen
                        + hashedSubpacket)

                unhashedSubpacket = self.makeSubPacket({16: self.asymKey.keyId})
                unhashedSubpacketLen = Util.int2str256(len(unhashedSubpacket), 2)

                if signatureType == 0x00:
                        sig = self.literalData
                elif signatureType == 0x13:
                        sig = self.asymKey.packet
                        sig += binascii.unhexlify('b4' +
'{0:0{1}x}'.format(len(self.asymKey.userId), 8))
                        sig += self.asymKey.userId
                elif signatureType == 0x18:
                        sig = self.asymKey.packet
                        sig += self.asymSubKey.packet
                else:
                        raise OpenPGPException('Not Implemented(write) yet signatureType ' +
hex(signatureType))

                sig += fistPart
                sig += binascii.unhexlify("04ff")
                sig += binascii.unhexlify('{0:0{1}x}'.format(len(fistPart), 8))
                hld = Util.hashAlgo(hashAlgoId)(sig).digest()

                signedHashValue = hld[:2]

                mm2 = Util.EMSA_PKCS1_v1_5(hld, hashAlgoId, self.asymKey.messegeLen)
                mm = self.asymKey.signRSA(mm2, passphrase)

                return (fistPart
                        + unhashedSubpacketLen
                        + unhashedSubpacket
                        + signedHashValue
                        + Util.toMPI(mm))

        def read_CompressedDataPacket(self, p, pEnd):
                # 5.6.  Compressed Data Packet (Tag 8)
                compressAlgo = ord(self.encodedFile[p])
                p += 1
                # 9.3.  Compression Algorithms
                if compressAlgo == 1:
                        #zip
                        decompressedFile = zlib.decompress(self.encodedFile[p: pEnd], -15)
                        print 'Reading myOpenPGP Compressed Data Packet'

        myOpenPGP().setAsymmetricKeys(self.asymmetricKeys).readFile(decompressedFile,
self.extraParam)
                elif compressAlgo == 2:
```

```
                        #zlib
                        raise OpenPGPException('9.3. Compression Algorithms: zlib')
                elif compressAlgo == 3:
                        #bzip2
                        raise OpenPGPException('9.3. Compression Algorithms: bzip2')
                else:
                        raise OpenPGPNotValidException('Compress Algo', compressAlgo, [1, 2,
3])
                return pEnd

        def write_CompressedDataPacket(self, compressAlgo, tags):
                # 5.6.  Compressed Data Packet (Tag 8)
                # 9.3.  Compression Algorithms
                if compressAlgo == 1:
                        #zip
                        data = myOpenPGP().setAsymmetricKeys(self.asymmetricKeys,
self.asymKey, self.asymSubKey).writeFile(tags, self.extraParam).encodedFile
                        compressZip = zlib.compressobj(zlib.Z_DEFAULT_COMPRESSION,
zlib.DEFLATED, -15)
                        return chr(compressAlgo) + compressZip.compress(data) +
compressZip.flush()
                elif compressAlgo == 2:
                        #zlib
                        raise OpenPGPException('9.3. Compression Algorithms: zlib')
                elif compressAlgo == 3:
                        #bzip2
                        raise OpenPGPException('9.3. Compression Algorithms: bzip2')
                else:
                        raise OpenPGPNotValidException('Compress Algo', compressAlgo, [1, 2,
3])
                return pEnd

        def read_One_Pass_Signature_Packets(self, p):
                # 5.4.  One-Pass Signature Packets (Tag 4)
                version = ord(self.encodedFile[p])
                p += 1
                if version == 3:
                        signatureType = ord(self.encodedFile[p])
                        #5.2.1.  Signature Types
                        p += 1

                        hashAlgoId = ord(self.encodedFile[p])
                        p += 1

                        publicKeyAlgo = ord(self.encodedFile[p])
                        p += 1

                        self.keyId = self.encodedFile[p: p + 8]
                        print 'keyId', binascii.hexlify(self.keyId)
                        p += 8

                        flagLastOnePass = ord(self.encodedFile[p])
                        p += 1

                        if flagLastOnePass == 0:
                                raise OpenPGPException('5.4. One-Pass Signature Packets (Tag
4)')
                else:
                        raise OpenPGPVersionException('One-Pass Signature Packet', version,
[3])
                return p

        def write_One_Pass_Signature_Packets(self, signatureType, hashAlgoId):
                # 5.4.  One-Pass Signature Packets (Tag 4)
                version = chr(3)
                flagLastOnePass = chr(1)
```

```
                return (version
                        + chr(signatureType)
                        + chr(hashAlgoId)
                        + chr(self.asymKey.publicKeyAlgo)
                        + self.asymKey.keyId
                        + flagLastOnePass)

        def readTag(self, tag, p, length):
                print('tag:', tag, 'length of packet:', length)
                if tag == 5 or tag == 7 or tag == 6 or tag == 14:
                        keyRSA = RSAOpenPGP().read(self.encodedFile[p:p+length])
                        Util.display('fingerPrint', keyRSA.fingerPrint)
                        if tag == 5 or tag == 6:
                                for asymKey in self.asymmetricKeys:
                                        if asymKey.fingerPrint == keyRSA.fingerPrint and
asymKey.nRSA == keyRSA.nRSA:
                                                if not asymKey.hasSecretData and
keyRSA.hasSecretData:
                                                        asymKey.insertSecretData(keyRSA)
                                                self.asymKey = asymKey
                                                break
                                else:
                                        self.asymmetricKeys.append(keyRSA)
                                        self.asymKey = keyRSA
                        else:
                                for asymSubKey in self.asymKey.subKeys:
                                        if asymSubKey.fingerPrint == keyRSA.fingerPrint and
asymSubKey.nRSA == keyRSA.nRSA:
                                                if not asymSubKey.hasSecretData and
keyRSA.hasSecretData:
                                                        asymSubKey.insertSecretData(keyRSA)
                                                self.asymSubKey = asymSubKey
                                                break
                                else:
                                        self.asymKey.subKeys.append(keyRSA)
                                        self.asymSubKey = keyRSA
                        return p + length
                elif tag == 1:
                        return self.read_Public_Key_Encrypted_Session_Key_Packets(p)
                elif tag == 18:
                        return self.read_SymEncryptedIntegrityProtectedDataPacket(p,
p+length)
                elif tag == 11:
                        return self.read_LiteralDataPacket(p, p+length)
                elif tag == 19:
                        return self.read_ModificationDetectionCodePacket(p)
                elif tag == 13:
                        return self.read_UserIDPacket(p, p+length)
                elif tag == 2:
                        return self.read_SignaturePacket(p, p+length)
                elif tag == 8:
                        return self.read_CompressedDataPacket(p, p+length)
                elif tag == 4:
                        return self.read_One_Pass_Signature_Packets(p)
                else:
                        raise OpenPGPNotValidException('Read Tag', tag, [1, 2, 4, 5, 6, 7, 8,
11, 13, 14, 18, 19])

        def writeTag(self, tagInfo):
                tag = tagInfo[0]
                if tag == 1:
                        return self.write_Public_Key_Encrypted_Session_Key_Packets()
                elif tag == 18:
```

```
                        return
self.write_SymEncryptedIntegrityProtectedDataPacket(tagInfo[1])
                elif tag == 11:
                        return self.write_LiteralDataPacket(tagInfo[1])
                elif tag == 19:
                        return self.write_ModificationDetectionCodePacket()
                elif tag == 4:
                        return self.write_One_Pass_Signature_Packets(tagInfo[1], tagInfo[2])
                elif tag == 2:
                        return self.write_SignaturePacket(tagInfo[1], tagInfo[2], tagInfo[3])
                elif tag == 5:
                        return self.write_secretKeyPaket(tagInfo[1])
                elif tag == 7:
                        return self.write_secretKeyPaket(tagInfo[1], tagInfo[2])
                elif tag == 6:
                        return self.write_publicKeyPaket(tagInfo[1])
                elif tag == 14:
                        return self.write_publicKeyPaket(tagInfo[1], tagInfo[2])
                elif tag == 13:
                        return self.write_UserIDPacket()
                elif tag == 8:
                        return self.write_CompressedDataPacket(tagInfo[1], tagInfo[2])
                else:
                        raise OpenPGPNotValidException('Write Tag', tag, [1, 2, 4, 5, 6, 7,
8, 11, 13, 14, 18, 19])

        def crc24(self, octets):
                #6.1.  An Implementation of the CRC-24 in "C"
                crc = CRC24_INIT = 0xB704CEL
                CRC24_POLY = 0x1864CFBL
                for x in octets:
                        crc ^= ord(x) << 16
                        for i in xrange(8):
                                crc <<= 1
                                if (crc & 0x1000000):
                                        crc ^= CRC24_POLY
                return crc

        def encodeAsc(self, title = 'MESSAGE'):
                crcFile = self.crc24(self.encodedFile)
                base64File = base64.b64encode(self.encodedFile)
                self.encodedFile = "-----BEGIN PGP " + title + "-----\n\n"
                p = 0
                while p < len(base64File):
                        self.encodedFile += base64File[p: p+64] + '\n'
                        p += 64
                self.encodedFile += '=' + base64.b64encode(Util.int2str256(crcFile, 3)) +
'\n'
                self.encodedFile += '-----END PGP ' + title + '-----\n'
                return self

        def decodeAsc(self):
                stringFile = self.encodedFile.split('\n')
                p = 0
                while stringFile[p].strip() != '':
                        p += 1
                p += 1
                q = len(stringFile) - 1
                while len(stringFile[q]) == 0 or stringFile[q][0] != '=':
                        q -= 1

                self.encodedFile = base64.b64decode(''.join(stringFile[p:q]))
                crcFile = self.crc24(self.encodedFile)
                if Util.toint(base64.b64decode(stringFile[q][1:])) != crcFile:
                        raise OpenPGPIncorrectException('OpenPGP Armor', 'crc24',
Util.toint(base64.b64decode(stringFile[q][1:])), crcFile)
```

```
                return self

        def saveFile(self, fileName, armor = None, needValidadion = False):
                if armor:
                        self.encodeAsc(armor)
                        fileName += '.asc'
                else:
                        fileName += '.gpg'
                if fileName == '.asc' or fileName == '.gpg':
                        print self.encodedFile
                else:
                        if needValidadion and os.path.isfile(fileName):
                                confirm = raw_input("The " + fileName + " file already exists.
Do you want to overwrite it?")
                                if confirm[0].lower() != 'y':
                                        print 'File was not saved'
                                        return self
                        open(fileName, "wb").write(self.encodedFile)
                        print 'File', fileName, 'saved with success'
                return self

        def savePrivateKey(self, userId, fileName, passphrase, armor = False, compress = 0,
needValidadion = False):
                tags = []
                for i, asymKey in enumerate(self.asymmetricKeys):
                        if userId.lower() not in asymKey.userId.lower():
                                continue
                        if not asymKey.hasSecretData:
                                continue
                        tags.append([5, i])
                        tags.append([13])
                        tags.append([2, 0x13, 8, passphrase])
                        for j in range(len(asymKey.subKeys)):
                                tags.append([7, i, j])
                                tags.append([2, 0x18, 8, passphrase])

                if tags != []:
                        if compress != 0:
                                tags = [[8, compress, tags]]
                        self.writeFile(tags)
                else:
                        print 'There is no Secret Keys for user', userId
                        return self
                if armor:
                        self.saveFile(fileName, 'PRIVATE KEY BLOCK', needValidadion)
                else:
                        self.saveFile(fileName, None, needValidadion)
                return self

        def savePublicKey(self, userId, fileName, passphrase, armor = False, compress = 0,
needValidadion = False):
                tags = []
                for i, asymKey in enumerate(self.asymmetricKeys):
                        if userId.lower() not in asymKey.userId.lower():
                                continue
                        tags.append([6, i])
                        tags.append([13])
                        tags.append([2, 0x13, 8, passphrase])
                        for j in range(len(asymKey.subKeys)):
                                tags.append([14, i, j])
                                tags.append([2, 0x18, 8, passphrase])

                if tags != []:
                        if compress != 0:
                                tags = [[8, compress, tags]]
                        self.writeFile(tags)
```

```
                else:
                        print 'There is no Public Keys for user', userId
                        return self
                if armor:
                        self.saveFile(fileName, 'PUBLIC KEY BLOCK', needValidadion)
                else:
                        self.saveFile(fileName, None, needValidadion)
                return self

        def signFile(self, signFile, userId, fileName, passphrase, armor = False, compress =
0, needValidadion = False):
                tags = []
                for i, asymKey in enumerate(self.asymmetricKeys):
                        if userId.lower() not in asymKey.userId.lower():
                                continue
                        if not asymKey.hasSecretData:
                                continue
                        self.asymKey = asymKey
                        self.asymSubKey = asymKey.subKeys[0]
                        signatureType = 0x00
                        hashAlgoId = 8
                        tags = [[4, signatureType, hashAlgoId], [11, signFile], [2,
signatureType, hashAlgoId, passphrase]]

                if tags != []:
                        if compress != 0:
                                tags = [[8, compress, tags]]
                        self.writeFile(tags)
                else:
                        print 'There is no Secret Keys for user', userId
                        return self
                if armor:
                        self.saveFile(fileName, 'MESSAGE', needValidadion)
                else:
                        self.saveFile(fileName, None, needValidadion)
                return self

        def encrypt(self, encryptFile, userId, fileName, armor = False, compress = 0,
needValidadion = False):
                tags = []
                for i, asymKey in enumerate(self.asymmetricKeys):
                        if userId.lower() not in asymKey.userId.lower():
                                continue
                        self.asymKey = asymKey.subKeys[0]
                        if compress == 0:
                                tags = [[1], [18, [[11, encryptFile], [19]]]]
                        else:
                                tags = [[1], [18, [[8, compress, [[11, encryptFile], [19]]]]]]

                if tags != []:
                        self.writeFile(tags)
                else:
                        print 'There is no Public Keys for user', userId
                        return self
                if armor:
                        self.saveFile(fileName, 'MESSAGE', needValidadion)
                else:
                        self.saveFile(fileName, None, needValidadion)
                return self

        def len2NewFormat(self, length):
                if length < 192:
                        return chr(length)
                elif length < 8383:
                        length -= 192
```

```
                        return chr((length>>8) + 192) + chr(length & 255)#chr(length &
((1<<8)-1))
                elif length < (1<<32):
                        return chr(255) + int2str256(length, 4)
                else:
                        raise OpenPGPException('4.2.2.4. Partial Body Lengths: ' + length)

        def writeFile(self, tags, extraParam = None):
                self.encodedFile = ''
                self.extraParam = extraParam

                for tag in tags:
                        self.encodedFile += chr(192 + tag[0])#chr((1<<7) + (1<<6) + tag)
                        packet = self.writeTag(tag)
                        self.encodedFile += self.len2NewFormat(len(packet)) + packet
                return self

        def readFile(self, encodedFile, extraParam = None):
                print '== Start Read File'
                self.encodedFile = encodedFile
                self.extraParam = extraParam
                if self.encodedFile[0] == '-':
                        self.decodeAsc()

                p = 0
                while(p < len(self.encodedFile)):
                        pTag = ord(self.encodedFile[p])
                        p += 1
                        one = pTag & 128#1<<7
                        if not one:
                                raise OpenPGPException('the beggin of packet must be 1: ' +
pTag)
                        newFormat = pTag & 64#1<<6
                        if newFormat:
                                tag = pTag & 63#(1<<6)-1
                                stOctet = ord(self.encodedFile[p])
                                p += 1
                                if stOctet < 192:
                                        length = stOctet
                                elif stOctet < 224:
                                        ndOctet = ord(self.encodedFile[p])
                                        p += 1
                                        length = (stOctet - 192 << 8) + ndOctet + 192
                                elif stOctet == 255:
                                        length = Util.toint(self.encodedFile[p: p + 4])
                                        p += 4
                                else:
                                        raise OpenPGPException('4.2.2.4. Partial Body Lengths:
' + stOctet)
                        else:
                                tag = (pTag & 63) >> 2#(pTag & (1<<6)-1) >> 2
                                lenType = pTag & 3#(1<<2)-1
                                if lenType < 3:
                                        length = Util.toint(self.encodedFile[p: p +
(1<<lenType)])
                                        p += (1<<lenType)
                                else:
                                        length = len(self.encodedFile) - p
                        p = self.readTag(tag, p, length)

                if p == len(self.encodedFile):
                        print '== File read with success'
                else:
                        print '== Error reading file'
                        raise OpenPGPException('Error reading file')
                return self
```

File Name: OpenPGPExceptions.py

```
import binascii

class OpenPGPException(Exception):
	pass

class OpenPGPVersionException(OpenPGPException):
	def __init__(self, value, triedVersion, validVersions):
		msg = "version of " + value + " can't be " + str(triedVersion) + " must be "
+ ', '.join(map(str, validVersions))
		super(OpenPGPVersionException, self).__init__(msg)

class OpenPGPNotValidException(OpenPGPException):
	def __init__(self, value, tried, validValues):
		msg = value + " can't be " + str(tried) + " must be " + ', '.join(map(str,
validValues))
		super(OpenPGPVersionException, self).__init__(msg)

class OpenPGPIncorrectException(OpenPGPException):
	def __init__(self, incorrect, value, actual, expected):
		msg = incorrect + " is incorrect the " + value + " is " +
binascii.hexlify(actual) + " and must be " + binascii.hexlify(expected)
		super(OpenPGPIncorrectException, self).__init__(msg)

class OpenPGPKeyIdException(OpenPGPException):
	def __init__(self, keyId, numKeys):
		if numKeys > 0:
			msg = "Has " + str(numKeys) + " keys but none of them has the keyId
equals as " + binascii.hexlify(keyId) + ", signature cannot be validated."
		else:
			msg = "You do not have any keys, signature cannot be validated."
		super(OpenPGPKeyIdException, self).__init__(msg)
```

File Name: testMyOpenPGP.py

```
from myOpenPGP import myOpenPGP

praTestar = False
if not praTestar:
	myOpenPGP().start()
else:
	secretKeyFile = open("secretKey.asc", "rb").read()
	publicKeyQWFile = open("publicKeyQW.asc", "rb").read()#open("qwPk.gpg", "rb").read()
	ml2File = open("ml2.txt.gpg", "rb").read()
	File2File = open("file2.txt.asc", "rb").read()
	compressZipFile = open("compressZip.gpg", "rb").read()
	rnKeyFile = open("rnKey.asc", "rb").read()
	ExampleFile = open("Example.asc", "rb").read()
	mTxtFile = open("m.txt.asc", "rb").read()
	passphase = "this is a pass"

	myOpenPGP().readFile(ExampleFile)
	myOpenPGP().readFile(publicKeyQWFile).readFile(File2File)
	myOpenPGP().readFile(secretKeyFile).readFile(ml2File)
	myOpenPGP().readFile(publicKeyQWFile).encrypt("file.txt", "", "file.txt", True)
	FileFile = open("file.txt.asc", "rb").read()
	myOpenPGP().readFile(secretKeyFile).signFile("file.txt", "", "mySign", passphase,
True)
	mySignFile = open("mySign.asc", "rb").read()
	myOpenPGP().readFile(publicKeyQWFile).readFile(mySignFile)
	myOpenPGP().readFile(publicKeyQWFile).readFile(rnKeyFile).readFile(compressZipFile)
	myOpenPGP().readFile(secretKeyFile).savePrivateKey("", "genSecrKey", passphase)
	genSecrKeyFile = open("genSecrKey.gpg", "rb").read()
	myOpenPGP().readFile(genSecrKeyFile).readFile(mTxtFile)
	myOpenPGP().readFile(secretKeyFile).savePrivateKey("", "genSecrKey", passphase,
True)
	genSecrKeyAscFile = open("genSecrKey.asc", "rb").read()
	myOpenPGP().readFile(genSecrKeyAscFile).readFile(FileFile)
	myOpenPGP().generateKeyRSA("myUser <my@user.com>", 'this is a
pass').savePrivateKey("user", "mySecrKey", passphase, True).savePublicKey("user",
"myPublKey", passphase).signFile("file2.txt", "my@user.com", "euSign", passphase)
	myPublKeyAscFile = open("myPublKey.gpg", "rb").read()
	myOpenPGP().readFile(myPublKeyAscFile).readFile(open("euSign.gpg",
"rb").read()).encrypt("file.txt", "my@user.com", "euEncript", True)
	mySecrKeyAscFile = open("mySecrKey.asc", "rb").read()
	myOpenPGP().readFile(mySecrKeyAscFile).readFile(open("euEncript.asc", "rb").read())
	myOpenPGP().readFile(publicKeyQWFile).readFile(publicKeyQWFile).readFile(File2File)
	myOpenPGP().readFile(publicKeyQWFile).readFile(secretKeyFile).readFile(ml2File).read
File(File2File)
	myOpenPGP().generateKeyRSA("uu <u@mail.com>", 'pass').savePrivateKey("uu",
"changeSKey", "pass").readFile(open("changeSKey.gpg",
"rb").read()).readFile(open("changeSKey.gpg", "rb").read()).generateKeyRSA("nn
<n@mail.com>", 'pp').readFile(open("changeSKey.gpg", "rb").read()).savePrivateKey("nn",
"changeSKey", "pp").readFile(open("changeSKey.gpg", "rb").read())
	myOpenPGP().readFile(secretKeyFile).savePrivateKey("", "genSecrKeyZip", passphase,
True, 1).savePublicKey("", "genPublKeyZip", passphase, True, 1).signFile("file.txt", "",
"mySignZip", passphase, True, 1)
	genPublKeyZipFile = open("genPublKeyZip.asc", "rb").read()
	myOpenPGP().readFile(genPublKeyZipFile).readFile(open("mySignZip.asc",
"rb").read()).encrypt("file.txt", "", "fileZip.txt", True, 1)
	genSecrKeyZipFile = open("genSecrKeyZip.asc", "rb").read()
	myOpenPGP().readFile(genSecrKeyZipFile).readFile(open("fileZip.txt.asc",
"rb").read())
	print "== OK"
```