Criptografia Assimétrica (RSA)

Rafael Campos Nunes[1], Rafael Henrique Nogalha de Lima[2]
19/0098295[1] 19/0036966[2]

Contents

1 Introdução

Foi inicialmente introduzido em 1977 por Ron Rivest, Adi Shamir e Leonard Adleman, por isso o acrônimo RSA. Ele é o método de criptografia assimétrica mais usado no mundo, no meio da internet. É assimétrico porque usa chave pública e privada, isso permite uma maior segurança na troca de dados e na integridade de informações. Além disso, o RSA, para a maior segurança, pode trocar mensagens em que são marcadas com uma assinatura, permitindo assim que os originadores criem mensagens inteligíveis apenas para os destinatários pretendidos.

2 Arquitetura

O projeto foi escrito na linguagem Python utilizando-se de módulos para organização do código. A pasta RSA, neste projeto, é um módulo contendo vários objetos e funções que separam a responsabilidade do programa de maneira adequada afim de aumentar a compreensão do código.

Nessa esteira, o código contém um módulo principal denominado *crypto* que armazena vários outros arquivos, cada um contendo responsabilidades bem determinadas. Os arquivos do módulo são denotados na tabela abaixo, assim como a descrição de sua responsabilidade

Arquivo	Funcionalidade
cipher.py	Interface para cifração e decifração de dados e outras funções criptográficas
certificate.py	Interface para assinatura e validação de assinaturas
hash.py	Contém abstração de uma função hash utilizada no módulo
primes.py	Interface para geração de números primos, teste de primalidade etc
key.py	Interface para geração de chaves públicas e privadas
utils.py	Contém função matemática para cálculo do MDC extendido

O segundo módulo do sistema é denominado *ioi*, responsável por disponibilizar funções para serialização e deserialização de objetos utilizados no contexto criptográfico.

2.1 Cifração e Decifração

2.2 Serialização

A serialização de dados nesse projeto foi fundamental pois, de outra forma, não conseguiríamos cifrar ou decifrar arquivos dado que, a título de exemplo, precisamos do tamanho da chave que foi gerada e também dados sobre a paginação (*padding*) do arquivo que foi cifrado. Portanto, o módulo *ioi* nos permite que coloquemos metadados nos arquivos que são gravados em disco.

3 Problemas

O algoritmo do RSA é, por si só, um algoritmo lento por possuir etapas na implementação com manipulação de números primos. Mas ele torna-se ainda mais lento com a sua implementação em determinadas linguagens, no nosso caso, foi implementado em Python e sendo assim, a implementação torna-se mais lenta que o normal.

A segunda dificuldade na implementação do projeto foi manipular inteiros e bytes, visto que o algoritmo trabalha com a criptografia de string para inteiros e durante o processo, são convertidos para bytes. Além de que, tanto a verificação e assinatura também trabalham com inteiros e bytes. Sendo assim, essa manipulação tornou-se uma dificuldade, visto que em algumas funções como em "oaep_encode" os inteiros tinham que ser tratados como bytes e por isso foi necessário uma maior atenção, além de serem realizados constantes debugs em toda a implementação, tornando-a cansativa em alguns momentos.

4 Ambiente

O ambiente utilizado para construção e teste do trabalho é o GNU/Linux, com o python na versão 3.6.9. No Windows o python3 é instalado com o nome python. certifique-se de que está utilizando a versão correta com $ `python --version`$.

No projeto também há a escrita de testes unitários para garantir o funcionamento de partes do sistema. Nesse sentido, para executar os testes certifique-se de que está no diretório correto e execute, a depender do diretório, a seguinte linha de comando para cada teste:

```
$ PYTHONPATH=. python3 '<nome_do_teste>_test.py'
```

Ou, para realizar a execução de todos os testes, utilize a linha de comando abaixo dentro da pasta `src`.

```
$ python3 -m unittest discover -s . -p '*_test.py'
```

4.1 Como utilizar a aplicação?

A aplicação contém diversos argumentos de entrada afim de facilitar seu uso. Para visualizá-las é necessário utilizar a *flag* `--help`. As flags do programa podem ser visualizadas na tabela abaixo.

Flag	Descrição
-g	Gera um par de chaves (pública e privada)
-k	Utilizado em conjunto com outros comandos para descrever uma chave
-s	Gera a assinatura de um arquivo, precisa ser utilizado com outros três parâmetros
-v	Valida uma assinatura utilizando uma chave pública, a assinatura e o dado em si
-f	Especifica o nome de um arquivo para cifração, assinatura ou decifração de dados

Alguns comandos são combinados com outras *flags* para especificar parâmetros específicos. Tais comandos, como os de assinatura e validação, se utilizam das chaves para funcionarem.

O primeiro passo para utilização da aplicação é a geração do par de chaves que pode ser realizado com o parâmetro *-g*.

```
$ python3 main.py -g
```

Após a execução desse comando é possível ver que o programa gerou duas chaves, uma com a extensão `.pub` e a outra `.prv`. Essas duas chaves podem ser utilizadas para cifrar e decifrar como nos exemplos abaixo.

```
$ python3 main.py -e -f arquivo.txt -k key.pub
```

Observe que após a execução do comando de cifração é possível ver que o programa escreveu o criptograma resultante em um arquivo com a extensão `.enc`.

```
$ python3 main.py -d -f arquivo.txt.enc -k key.prv
```

A mesma coisa acontece ao decifrar um arquivo .enc, onde o arquivo resultante terá a extensão `.dec`. Para assinaturas de arquivos de texto basta simplesmente utilizar a *flag -s* de acordo com o exemplo abaixo.

```
$ python3 main.py -s key.pub key.prv data.txt
```

O resultado da execução do comando anterior é um arquivo com a extensão *.sig* que contém a assinatura do arquivo *data.txt* em base64. A validação da assinatura é análoga, só que utiliza a assinatura, a chave pública e o dado em claro.