

Если вы видите что-то необычное, просто сообщите мне.

Реализация RSA шифрования и подписи на Golang

Эта статья описывает как работает RSA алгоритм, и как его можно реализовать на Go.

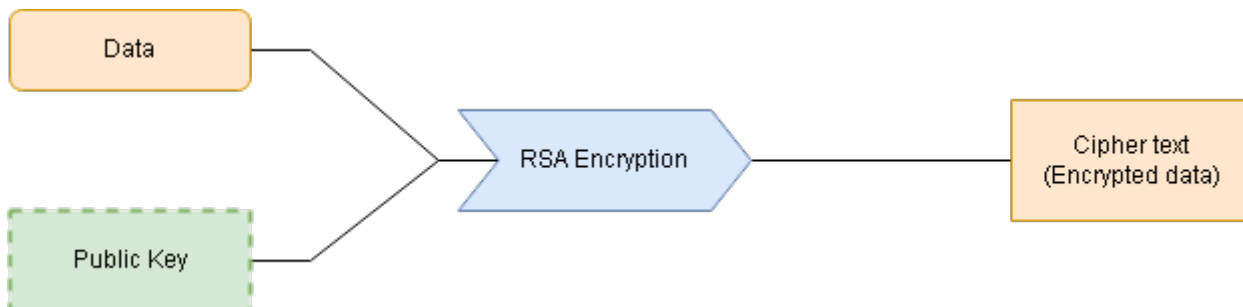
RSA (Rivest-Shamir-Adleman) шифрование наиболее широко распространено для безопасного шифрования данных.

Это асимметрический алгоритм шифрования, который по простому можно назвать "в одну сторону". В данном случае, легко для кого угодно зашифровать кусочек данных, но расшифровать его может только тот у кого есть настоящий ключ.

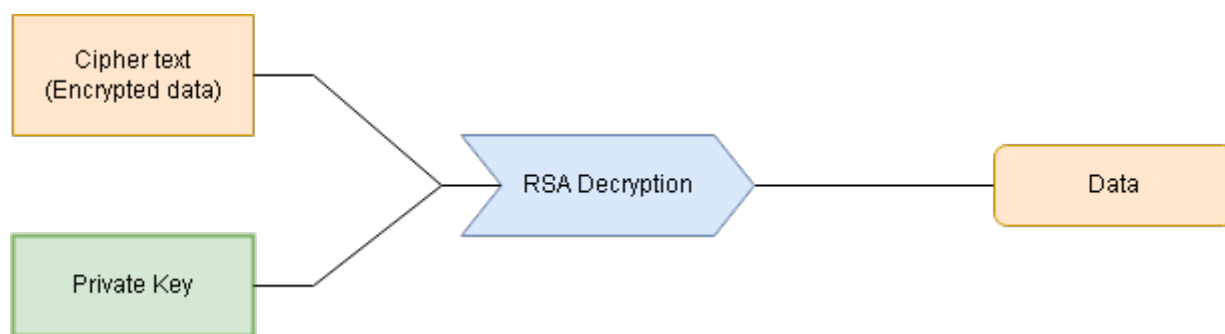
RSA Шифрование изнутри

RSA работает генерируя публичный и приватный ключ. `public` и `private` ключи создаются одновременно и формируют пару ключей.

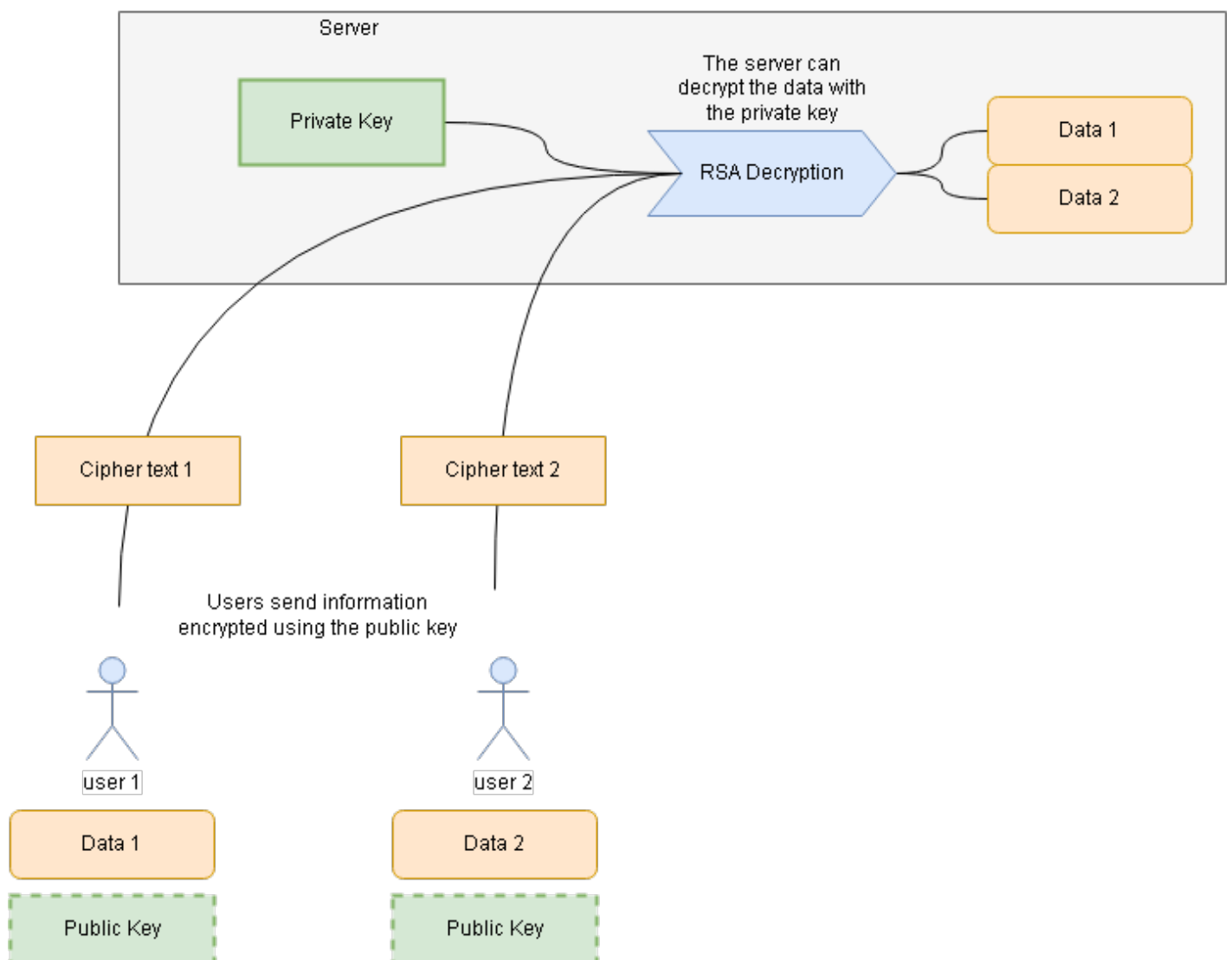
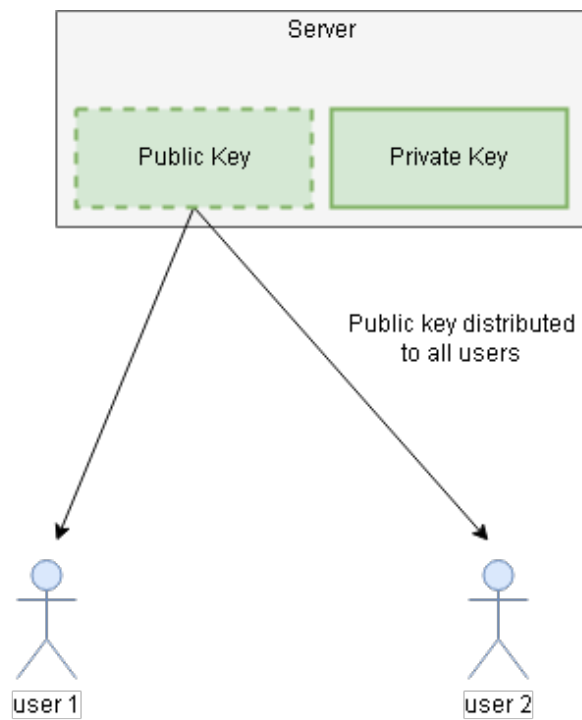
Публичный ключ может быть использовать для шифрования произвольной части кода, но не может его расшифровать.



Приватный ключ может быть использовать для расшифровки любой части данных которая была зашифрована соответствующим ключем.



Это значит, что мы даем наш публичный ключ, кому угодно. Они могут затем зашифровать любую информацию, которую они хотят нам послать, и единственный способ получить к ней доступ можно только с помощью приватного ключа.



Генерация ключа

Первое, что мы хотим сделать, это создать пару ключей: `public` и `private`. Эти ключи случайным образом генерируются, и будут использоваться в дальнейшем.

Мы используем `crypto/rsa` стандартную библиотеку для генерации ключа и `crypto/rand` библиотеку для генерации случайных чисел.

```
// Метод GenerateKey принимает reader, который возвращает случайные биты, и количество бит.
privateKey, err := rsa.GenerateKey(rand.Reader, 2048)
if err != nil {
    panic(err)
}

// Публичный ключ это часть *rsa.PrivateKey структуры
publicKey := privateKey.PublicKey

// Используем приватный и публичные ключи
// ...
```

Переменные `publicKey` и `privateKey` будут использоваться шифрования и расшифровки соответственно.

Шифрование

Мы будем использовать метод `EncryptOAEP` для шифрования случайного сообщения. Мы должны предоставить этому методу:

1. Хэширующую функцию, выбранную таким образом, что даже если ввод изменится едва ли, вывод должен меняться совершенно. Алгоритм SHA256 подходит для этого.
2. Случайный reader используемый для создания случайных битов, таким образом, чтобы каждый одинаковый ввод будет отдавать тот же вывод.
3. Публичный ключ ранее сгенерированный.
4. Сообщение которое мы шифруем.

5. Дополнительные параметры, которые мы опустим в этот раз.

```
encryptedBytes, err := rsa.EncryptOAEP(  
    []sha256.New(),  
    []rand.Reader,  
    []&publicKey,  
    []byte("super secret message"),  
    []nil)  
if err != nil {  
    []panic(err)  
}  
  
fmt.Println("encrypted bytes: ", encryptedBytes)
```

Функция выведет зашифрованные байты, которые выглядят как шум.

Расшифровка

Чтобы получить доступ к содержанию информации в зашифрованных битах, её нужно расшифровать.

Единственный способ расшифровать их, использовать приватный ключ соответствующий публичному ключу, с помощью которого мы зашифровали.

Структура `*rsa.PrivateKey` имеет метод `Decrypt` который мы будем использовать для получения оригинальной информации обратно из зашифрованной.

Данные необходимые для расшифровки:

1. Зашифрованные данные(называемые cipher text)
2. Хэш который мы использовать для шифрования данных

```
// Первый аргумент это генератор случайных данных( rand.Reader который мы использовали ранее)  
// Мы можем указать это значение как nil  
// OAEPOptions в конце означают, что мы шифруем данные используя OAEP, и то мы используем SHA256 в  
// качестве входного хэша.  
decryptedBytes, err := privateKey.Decrypt(nil, encryptedBytes, &rsa.OAEPOptions{Hash: crypto.SHA256})
```

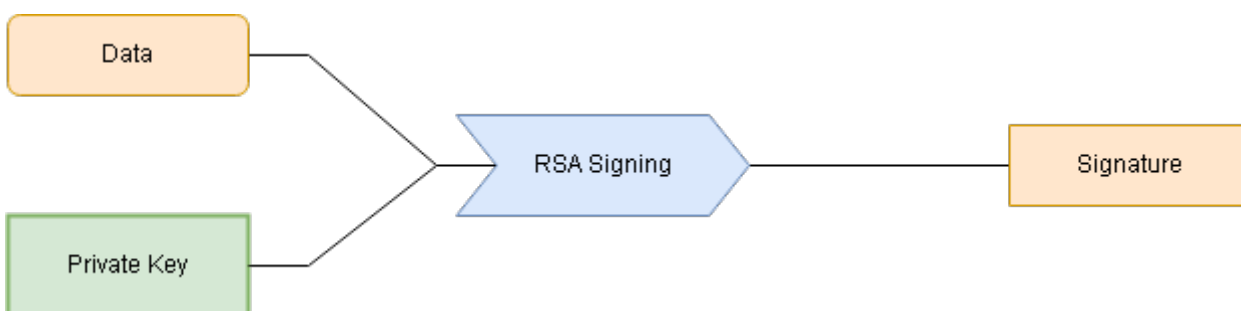
```
if err != nil {  
    panic(err)  
}
```

```
// Мы возвращаем информацию в оригинальной форме байт, которые мы приводим к строке и выводим.  
fmt.Println("decrypted message: ", string(decryptedBytes))
```

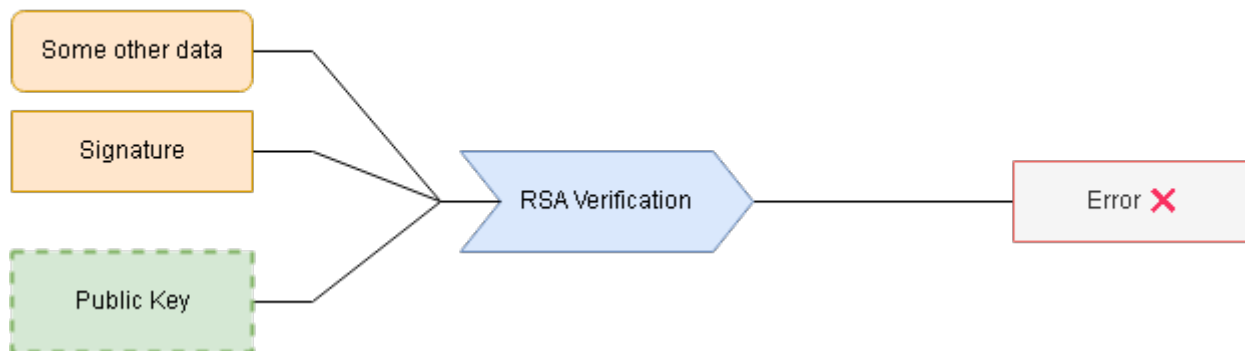
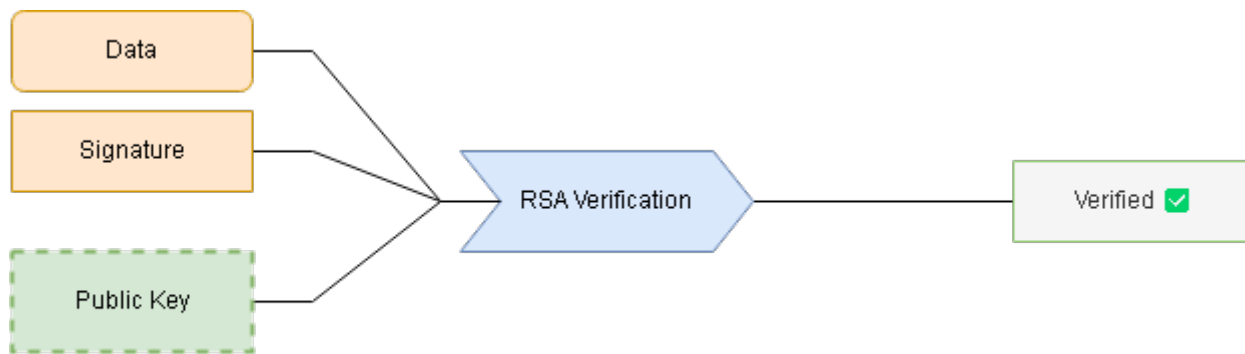
Подпись и проверка

RSA ключ используется так же для подписи и проверки. Подпись отличается от шифрования, что позволяет нам осуществить аутентификацию и не нарушить конфиденциальность.

Что это значит? Что вместо маскирования содержания оригинального сообщения(как это было сделано ранее), часть данных созданная из сообщения называется `signature` сигнатурой.



У кого есть сигнатура, сообщение и публичный ключ, могут использовать RSA для проверки, того, что сообщение действительно пришло от того, чей публичный ключ используется. Если данные или подпись не совпадают, проверка завершается неудачей.



Только часть с приватным ключем может подписать сообщение, но кто угодно с публичным ключем могут проверить его.

```
msg := []byte("verifiable message")

// Перед подписью, нужен хэш сообщения
// Хэш - то что мы подписали.
msgHash := sha256.New()
_, err = msgHash.Write(msg)
if err != nil {
    panic(err)
}
msgHashSum := msgHash.Sum(nil)

// Чтобы сгенерировать подпись, нам нужен генератор случайных чисел.
// наш приватный ключ, алгоритм хеширования, который мы использовали и хэш сумму нашего
// сообщения

signature, err := rsa.SignPSS(rand.Reader, privateKey, crypto.SHA256, msgHashSum, nil)
if err != nil {
    panic(err)
}
```

```
// Для проверки подписи, мы предоставляем публичный ключ, алгоритм хэширования,  
// хэш сумму нашего сообщения и подпись которую мы ранее сгенерировали.  
// А так же "options" параметры которые мы опустим  
err = rsa.VerifyPSS(&publicKey, crypto.SHA256, msgHashSum, signature, nil)  
if err != nil {  
    fmt.Println("could not verify signature: ", err)  
    return  
}  
// Если не получили ошибки от метода `VerifyPSS`, значит наша подпись верна.  
fmt.Println("signature verified")
```

Выводы

В этой статье мы разобрали, как сгенерировать RSA публичный и приватные ключи и как использовать их для шифрования, подписи и проверки данных.

Есть ограничение, о которых стоит знать прежде чем использовать алгоритм.

1. Данные, которые вы пытаетесь зашифровать должны быть гораздо короче чем сила вашего ключа. Для примера: EncryptOAEP документация говорит, что сообщение не должно быть больше, чем длинна публичного модуля минус двойная длинна хэша, и дальше минус 2.
2. Используемый алгоритм хэширования должен быть подобран к случаю.
SHA256(который мы используем) является достаточным для большинства случаев, но вы можете захотеть использовать что-то по серьезнее типа SHA512 для критически важных данных приложения.

Revision #1

Created 15 August 2022 20:14:01 by gasick

Updated 16 April 2023 19:30:04 by gasick