

🕒 < Soham Kamani />

Implementing RSA Encryption and Signing in Golang (With Examples)

April 8, 2020 · Soham Kamani

This post will describe what the RSA algorithm does, and how we can implement it in Go.



RSA (*Rivest–Shamir–Adleman*) encryption is one of the most widely used algorithms for secure data encryption.

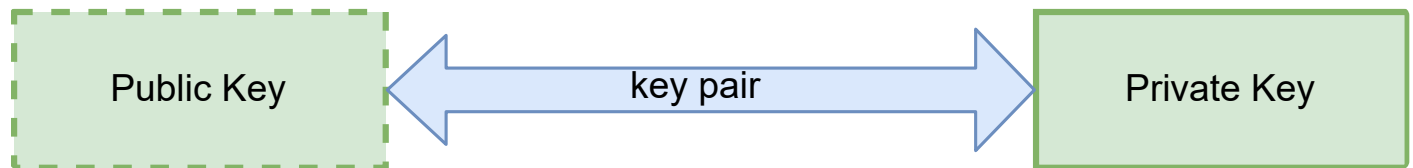
It is an *asymmetric* encryption algorithm, which is just another way to say “one-way”. In this case, it’s easy for anyone to encrypt a piece of data, but only possible for someone with the correct “key” to decrypt it.

If you want to skip the explanation and just see the working source code, you can

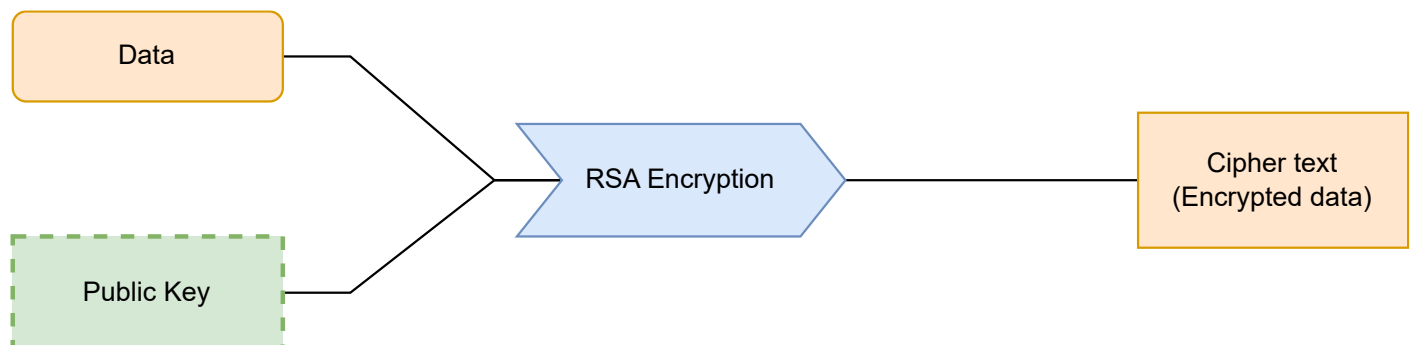
X

RSA Encryption In A Nutshell

RSA works by generating a public and a private key. The public and private keys are generated together and form a key pair.

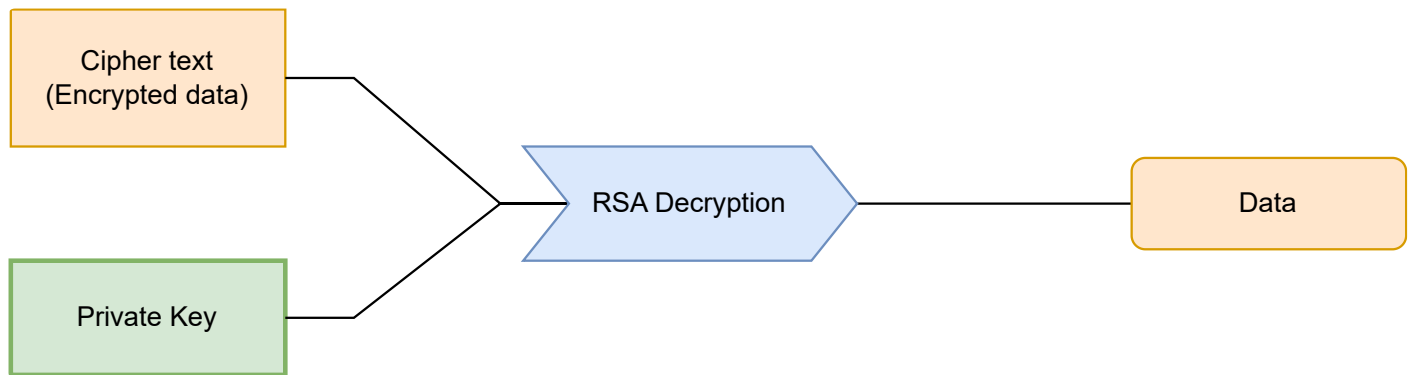


The public key can be used to encrypt any arbitrary piece of data, but cannot decrypt it.



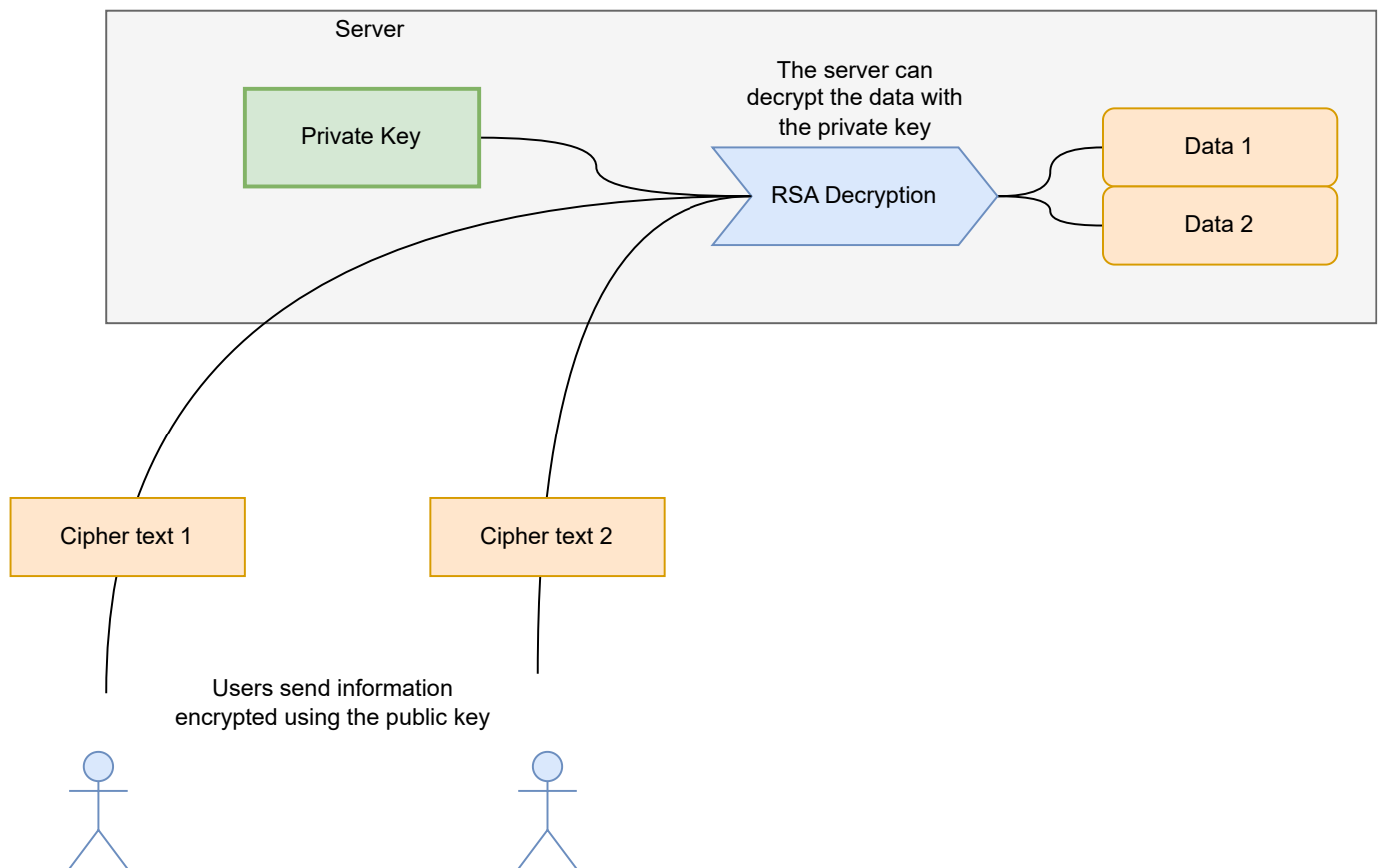
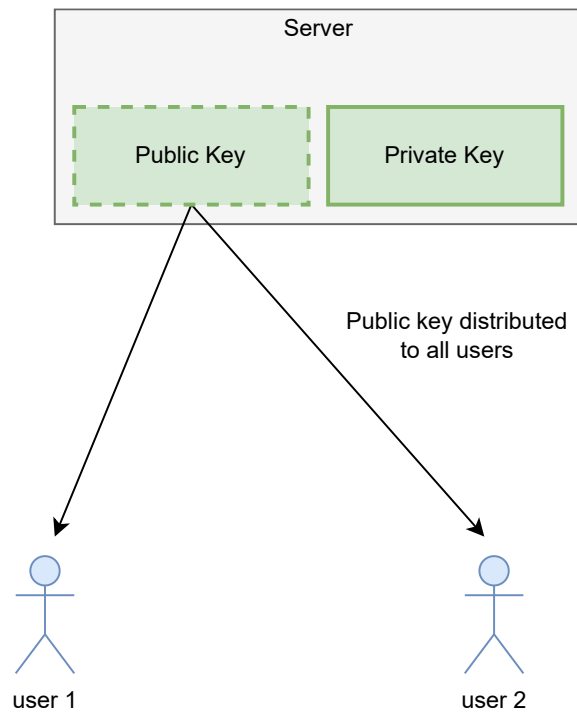
The private key can be used to decrypt any piece of data that was encrypted by its corresponding public key.

Advertisements



This means we can give our public key to whoever we want. They can then encrypt any information they want to send us, and the only way to access this information is by using our private key to decrypt it.





The details of how the keys are generated, and how information is encrypted and decrypted is beyond the scope of this post, but if you want to delve into the details, there is a [great video](#) on the topic

Advertisements

Key Generation

The first thing we want to do is generate the public and private key pairs. These keys are randomly generated, and will be used for all following operations.

We use the [crypto/rsa](#) standard library for generating the keys, and the [crypto/rand](#) library for generating random numbers.

```
// The GenerateKey method takes in a reader that returns random bits, and
// the number of bits
privateKey, err := rsa.GenerateKey(rand.Reader, 2048)
if err != nil {
```



```
// The public key is a part of the *rsa.PrivateKey struct
publicKey := privateKey.PublicKey

// use the public and private keys
// ...
```

The `publicKey` and `privateKey` variables will be used for encryption and decryption respectively.

Encryption

We will use the `EncryptOAEP` method for encrypting an arbitrary message. We must provide a few inputs to this method:

1. A hashing function, chosen so that even if the input is changed slightly, the output hash changes completely. The SHA256 algorithm is suitable for this
2. A random reader used for generating random bits so that the same input doesn't give the same output twice
3. The public key generated previously
4. The message we want to encrypt
5. An optional label (which we will omit in this case)

Advertisements

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

This will print out the encrypted bytes, which look more or less like garbage.

Decryption

To access the information contained in the encrypted bytes, they need to be decrypted.

The only way we can decrypt them is by using the private key corresponding to the public key we encrypted them with.

The `*rsa.PrivateKey` struct comes with a `Decrypt` method which we will use to get the original information back from the encrypted data.

The data we have to provide for decryption is:

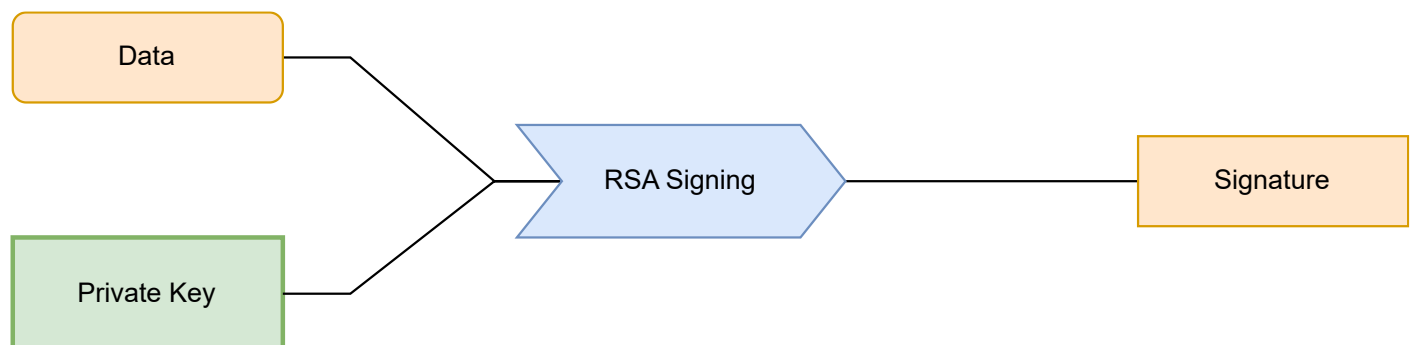
1. The encrypted data (called the *cipher text*)
2. The hash that we used to encrypt the data

```
// The first argument is an optional random data generator (the rand.Reader we use  
// we can set this value as nil  
// The OAEPOptions in the end signify that we encrypted the data using OAEP, and t  
// SHA256 to hash the input.  
decryptedBytes, err := privateKey.Decrypt(nil, encryptedBytes, &rsa.OAEPOptions{Ha  
if err != nil {  
    panic(err)  
}
```

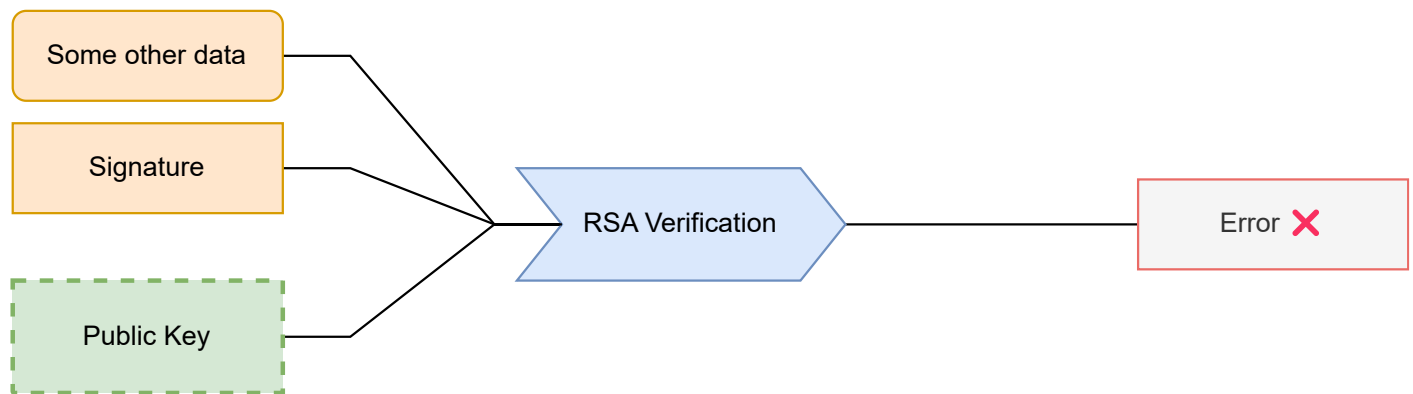
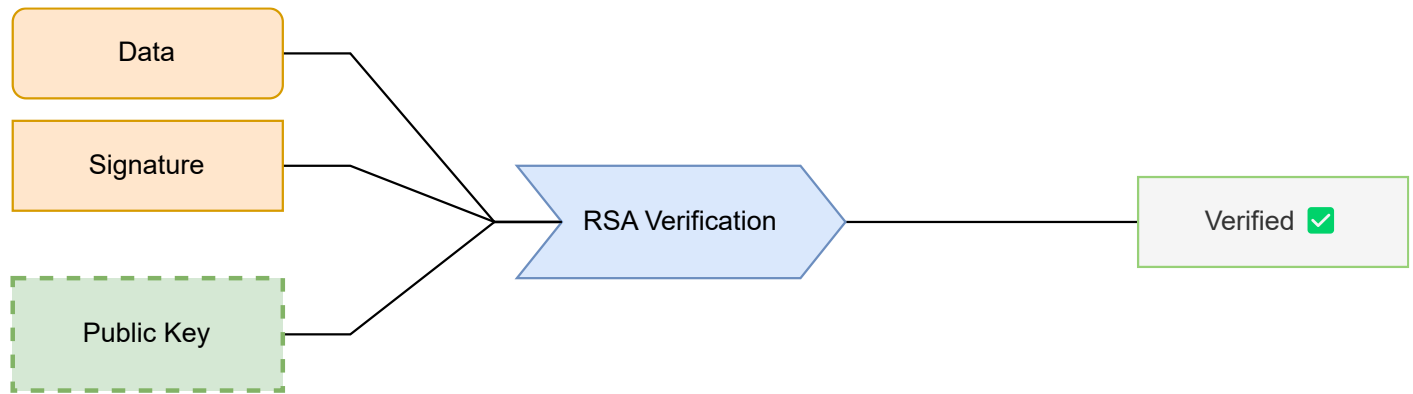

Signing And Verification

RSA keys are also used for signing and verification. Signing is different from encryption, in that it enables you to assert authenticity, rather than confidentiality.

What this means is that instead of masking the contents of the original message (like what was done in [encryption](#)), a piece of data is generated from the message, called the "signature".



public key is issued. If the data or signature don't match, the verification process fails.



Note that only the party with the private key can *sign* a message, but anyone with the public key can *verify* it.

```

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

// Before signing, we need to hash our message
// The hash is what we actually sign
msgHash := sha256.New()
_, err = msgHash.Write(msg)
if err != nil {
    panic(err)
}
msgHashSum := msgHash.Sum(nil)
  
```

// In order to generate the signature, we provide a random number generator



```
if err != nil {  
    panic(err)  
}  
  
// To verify the signature, we provide the public key, the hashing algorithm  
// the hash sum of our message and the signature we generated previously  
// there is an optional "options" parameter which can omit for now  
err = rsa.VerifyPSS(&publicKey, crypto.SHA256, msgHashSum, signature, nil)  
if err != nil {  
    fmt.Println("could not verify signature: ", err)  
    return  
}  
// If we don't get any error from the `VerifyPSS` method, that means our  
// signature is valid  
fmt.Println("signature verified")
```

Advertisements

Conclusion

In this post we have seen how to generate RSA public and private keys and how to use them to encrypt, decrypt, sign and verify arbitrary data.

There are some limitations that you should know before using these on your data. First, the data you are trying to encrypt should be much shorter than the bit strength of your



The hashing algorithm used should also be appropriate for your use case. SHA256 (which is used in the examples here) is considered sufficient for most use cases, but you may want to consider something like SHA512 for more data-critical applications.

You can find the complete working source code for all examples [here](#)

Advertisements



report this ad

Share:      

Like what I write? Let me know your email, and I'll send you more posts like this. No spam, I promise!

Your Email 

Subscribe 



G

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS

?

Name

♡

3

Share

Best

NewestOldest

- K

kargirwar

a year ago


Very useful. Keep it up!

0

0

Reply • Share ›

—

🚩
- 

Francesco Sirocco

2 years ago

Thanks for this post! All the info I need to start are here and the example are very clear!


0

0

Reply • Share ›

—

🚩



Soham Kamani

2 years ago

Thanks! I'm glad you found it useful :)

0

0

Reply • Share ›

—

🚩

➔

Francesco Sirocco
- Subscribe

Privacy

Do Not Sell My Data

© 2023 < Soham Kamani />
-
- https://www.sohamkamani.com/golang/rsa-encryption/

13/13