Learn Golang encryption and decryption

October 21, 2021 6 min read

Given that security is not always 100% guaranteed, there is always a need to protect your information, especially online data. By using encryption, we can convert information into computer code, thereby preventing unauthorized access.

For developers, encryption is essential to protect data in our applications. Imagine if we leave users' passwords in plain text in a database and the database becomes compromised; this can be catastrophic, leading the wrong people to your information.

With encryption, this can be prevented.

In this tutorial, we will look at how to encrypt and decrypt data in Go, keeping our data safe by making it difficult to use if it falls into the wrong hands.

Golang encryption tutorial prerequisites

To follow this tutorial, you must have the following:

- Golang installed on your machine
- Basic understanding of Go
- A command terminal
- A text editor

Setting up the Golang project

To get started, let's set up our Go project quickly.

If you installed Golang globally on your machine, you can create a folder where your Go project will reside. If you did not install Golang globally, create a folder in the root folder where your Go installation is.

This all depends on the operating system you use and your Go installation method.

To ensure you have Go working properly in the folder you are at, run the following command in your terminal:

```
go version
```

You will see the version of Go that you are running in the terminal:

Next, create a folder and cd into it:

```
mkdir Encrypt
cd Encrypt
```

You can then enable dependency tracking by running the following:

```
go mod init code/encrypt
```

This creates a `go.mod` file. Think of this as `package.json` in JavaScript or `composer.json` in PHP. This `go.mod` file is where all external modules used in any Go project are listed.

For this tutorial, we do not necessarily need to install external dependencies because Go comes with many modules that can generate, encrypt, and decrypt data.

Generating random numbers in Golang

Generating random numbers or strings is important in programming and is the base of encryption. Without generating random numbers, encryption would be useless and the encrypted data predictable.

Over 200k developers use LogRocket to create better digital experiences

Learn more →

To generate random numbers in Go, let’s create a new Go file in the project directory:

```
touch numbers.go
```

Next, copy and paste the following code into the newly created file:

```
package main
import (
    "fmt"
    "math/rand"
)
func main() {
    fmt.Println(rand.Intn(100))
}
```

Here, we imported the `fmt` package to format data and the `math/rand` package to generate random numbers. While these two packages are built into Go, be mindful that Go will not run successfully if there is an imported package that is unused in your program.

The additional `main()` function, which is an entry point of every executable file, prints a random integer that ranges from zero to 99 using the `rand.Intn()` function.

To do this, let's run the following:

```
run go numbers.go
```

In my case, I got 81. The problem now, however, is that when I rerun the program, I always get 81. While this isn't technically a problem, it does defeat the aim of generating a random number whenever running the code.

Nothing a computer does is simply random; it follows algorithms. To fix this, we must use the `Seed()` method with `rand`. This runs under the hood, but it takes `1` as the default parameter.

Add the following code at the beginning of the `main()` function:

```
rand.Seed(time.Now().UnixNano())
```

Since we are using time, we must import the time package `time.Now().UnixNano()`, which gives us the current time down to the second, thereby changing the `Seed()` parameter.

So, when we now run the `numbers.go` file, we always get a different random number.

Our code should now look like so:

```
package main
import (
    "fmt"
    "math/rand"
     "time"
)
func main() {
    rand.Seed(time.Now().UnixNano())
    fmt.Println(rand.Intn(100))
}
```

Then, we can run the code again and finally get a different random number between zero and 99 without it repeating:

```
run go numbers.go
```

Generating random strings in Golang

To generate random strings in Go, we'll use Base64 encoding and an external package because it's a more practical and secure way of generating random numbers.

To begin, create a file called `strings.go` in the root directory of the project. Then, after stating `package main`, tell Go that this is an executable file, followed by importing the `encoding/base64` and `fmt` modules:

```go
package main

import (
    "encoding/base64"
    "fmt"
)

func main() {

    StringToEncode := "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"

    Encoding := base64.StdEncoding.EncodeToString([]byte(StringToEncode))
    fmt.Println(Encoding)
}
```

By using the Base64 encoding, we can now encode and decode strings.

We then follow with the `main()` function that has the `StringToEncode` variable, which is the string we are encrypting. Afterward, we call the methods that come with the Base64 package and pass the variable created that needs encoding.

More great articles from LogRocket:

- Don't miss a moment with The Replay, a curated newsletter from LogRocket
- Learn how LogRocket's Galileo cuts through the noise to proactively resolve issues in your app
- Use React's useEffect to optimize your application's performance
- Switch between multiple versions of Node
- Discover how to animate your React app with AnimXYZ
- Explore Tauri, a new framework for building binaries
- Compare NestJS vs. Express.js

Running this program produces the following:

To ensure this always returns different strings all the time, we can use a third-party package called `randstr`.

`randstr` solves the problem quicker and better than using the `Seed()` method. To use the package, download the following:

```
go get -u github.com/thanhpk/randstr
```

This adds a `go.sum` file, which means we do not need to reinstall packages previously installed because it caches the packages within it and provides the path to the downloaded package to the `go.mod` file.

To generate a random number so the string's length will always be 20 characters, for example, create a new file and paste the following code:

```
 package main
import(
  "github.com/thanhpk/randstr"
  "fmt"
)
func main() {
    MyString := randstr.String(20)
    fmt.Println(MyString)

}
```

Every time we run this, the code reproduces different random strings that are 20 characters long. Easy? The package already handles a lot of the seeding when we generated random numbers, providing cleaner code.

Encrypting and decrypting data in Golang

We learned how to generate random numbers and strings, so we can now learn how to encrypt and decrypt data.

In almost all cases, security is the main reason why we need to understand this. So, we'll use the following modules: `crypto/aes` , `crypto/cipher` , `encoding/base64` , and `fmt` . However, the `crypto` modules specifically lend their security functionality to help us in our endeavors.

Encrypting

Encryption is simply a method of hiding data so that it is useless if it falls into the wrong hands. To encrypt in Go, we'll use the Advanced Encryption Standard, which `crypto/aes` provides.

To begin, create the file `encrypt.go` and paste the following code into it:

```
package main
import (
 "crypto/aes"
 "crypto/cipher"
 "encoding/base64"
 "fmt"
)

var bytes = []byte{35, 46, 57, 24, 85, 35, 24, 74, 87, 35, 88, 98, 66, 32, 14, 05}
// This should be in an env file in production
const MySecret string = "abc&1*~#^2^#s0^=)^^7%b34"
func Encode(b []byte) string {
 return base64.StdEncoding.EncodeToString(b)
}


// Encrypt method is to encrypt or hide any classified text
func Encrypt(text, MySecret string) (string, error) {
 block, err := aes.NewCipher([]byte(MySecret))
```

By adding random bytes, we can use them as an argument in the `crypto/cipher` module method, `NewCFBEncrypter()` . Then, before the `Encode` function, which encodes and returns the string to Base64, there is the `MySecret` constant that contains the secret for the encryption.

The `Encrypt` function, which takes two arguments, provides the text to encode and the secret to encode it. This then returns the `Encode()` function and passes the `cipherText` variable defined with the scope of `Encrypt` .

By running the file, the `main` function executes with the `StringToEncrypt` variable that contains the string to encrypt. The `Encrypt()` function also executes when the main function executes and now has two parameters: `StringToEncrypt` and `MySecret` .

Running this code produces the following:

Decrypting

After encrypting our string successfully, we can take it and decrypt it to its original state. But why should we even do this in the first place?

One of the common use cases of this is users' passwords, which should be encrypted before saving to the database. However, we must always decrypt it before we can give access to the user in our application.

To do this, we must take the encrypted string we received from the previous code block, `Li5E8RFcV/EPZY/neyCXQYjrfa/atA==` , and decrypt it by adding the following functions to the `encrypt.go` file:

```
func Decode(s string) []byte {
 data, err := base64.StdEncoding.DecodeString(s)
 if err != nil {
  panic(err)
 }
 return data
}
```

With the `Decode` function taking a single parameter, we can call it within the `Decrypt` function below:

```
// Decrypt method is to extract back the encrypted text
func Decrypt(text, MySecret string) (string, error) {
 block, err := aes.NewCipher([]byte(MySecret))
 if err != nil {
  return "", err
 }
 cipherText := Decode(text)
 cfb := cipher.NewCFBDecrypter(block, bytes)
 plainText := make([]byte, len(cipherText))
 cfb.XORKeyStream(plainText, cipherText)
 return string(plainText), nil
}
```

The `Decrypt` function takes two parameters that are strings: the `text`, which is the text from the encrypted data, and `MySecret`, which is a variable we already defined and gave a value to.

Inside the `main()` function, add the following code below `fmt.Println(encText)`, which prints on the next line of the encrypted text:

```
decText, err := Decrypt("Li5E8RFcV/EPZY/neyCXQYjrfa/atA==", MySecret)
 if err != nil {
  fmt.Println("error decrypting your encrypted text: ", err)
 }
 fmt.Println(decText)
```

At the end, we should have the full code in `encrypt.go`:

```
package main
import (
 "crypto/aes"
 "crypto/cipher"
 "encoding/base64"
 "fmt"
)

var bytes = []byte{35, 46, 57, 24, 85, 35, 24, 74, 87, 35, 88, 98, 66, 32, 14, 05}
// This should be in an env file in production
const MySecret string = "abc&1*~#^2^#s0^=)^^7%b34"
func Encode(b []byte) string {
 return base64.StdEncoding.EncodeToString(b)
}
func Decode(s string) []byte {
 data, err := base64.StdEncoding.DecodeString(s)
 if err != nil {
  panic(err)
 }
 return data
```

Running this encrypts and decrypts the data and will print the following:

Conclusion

You have successfully seen this through. We covered things like generating random data like strings and numbers, looked at how to encrypt using the Advanced Encryption Standard with Go modules like `crypto/aes` , `crypto/cipher` , `encoding/base64` .

And, we not only encrypt data, but we also decrypted the encrypted data.

You can find the entire source code here.