Encrypt data with a password in Go

Jan Pieter Bruins Slot Follow
Mar 18 · 12 min read ★

Photo by Shane Avery on Unsplash

Introduction

When we're encrypting data, typically we will create a random key that is able to decrypt that data. In some specific cases one wants to use a user specified key to decrypt that data like a password. However, the key that is used for cryptographic algorithms typically needs to be at least 32 bytes. But, it is likely that our password won't make that criteria, so we need to have a solution for that. Recently, I needed such a method, and in

this post I'll lay out what I've done in order to solve it. But before we get into the nitty-gritty.

DISCLAIMER: I'm not an expert at encryption, I've mentioned the sources that I've used to come to the solutions provided in this post. I implore you read/watch those sources to better understand it. And, as such if there are any errors in the post/code please let me know or leave a comment so I can update it, so that there is no perpetuation of wrong methods/techniques.

OK, since we've got that out of the way, let's begin! (Note: you can also view this post here)

Encrypt

Let's first start with encrypting our data. We'll start with creating the `Encrypt` function that will accept a `key` and a `data` argument. Based on that we will encrypt the data that can be decrypted using the `key` . First, we will generate that key by using 32 random bytes, later on we'll replace that with our password. Below, shows the code that is able to encrypt our data, provided by a generated key.

```
import (
    "crypto/aes"
    "crypto/cipher"
    "crypto/rand"
)

func Encrypt(key, data []byte) ([]byte, error) {
    blockCipher, err := aes.NewCipher(key)
    if err != nil {
        return nil, err
    }

    gcm, err := cipher.NewGCM(blockCipher)
    if err != nil {
        return nil, err
    }

    nonce := make([]byte, gcm.NonceSize())
    if _, err = rand.Read(nonce); err != nil {
        return nil, err
    }

    ciphertext := gcm.Seal(nonce, nonce, data, nil)

    return ciphertext, nil
}
```

So, let's go over the code, and inspect what we're doing.

```
func Encrypt(key, data []byte) ([]byte, error)
```

First, we start by creating our `Encrypt` function, and it will accept a `key` and a `data` argument. We'll be using a `byte` slice instead of an `io.Reader` as the `data` argument. While using `io.Reader` would allow us to use the `Encrypt` function with every other type that implements the `io.Reader` interface. (Ryer 2015) It is however because of the nature of `io.Reader`, being a stream of data, that when we want to decrypt the `ciphertext`, we need to see it in its entirety. A solution would be to break the stream into discrete chunks, however this would add significant complexity to the problem.[1] (Isom 2015)

```
blockCipher, err := aes.NewCipher(key)
```

We're initializing the block cipher based on the `key` that we provided. Here we're using the `crypto/aes` [2] package that implements the AES[3] [4] (Advanced Encryption Standard) encryption algorithm. AES is a symmetric-key encryption algorithm, that will be secure enough for modern use cases. Additionally, AES uses hardware acceleration on most platforms, so it'll be pretty fast to use. (Tankersley 2016)

```
gcm, err := cipher.NewGCM(blockCipher)
```

Here we're wrapping the block cipher, with a specific *mode*. We do this because we shouldn't use a `cipher.Block` interface directly. This is because the block cipher only encrypts 16 bytes of data, nothing more. So if you would call `blockCiper.Encrypt()` it would only encrypt the first 16 bytes. Thus we need something on top of that, and wrap the block cipher, and those are called *modes*. Again we have several *modes* to choose from, and here we're going to use the *Galois Counter Mode* (GCM)[5], with a standard nonce length.

Only GCM provides authenticated encryption, and it implements the `cipher.AEAD` interface (Authenticated Encryption with Associated Data)[6]. Authenticated encryption means that not only is your data going to be confidential, secret, and encrypted, it's also now going to be tamper proof. If someone alters the `ciphertext` you will not then be able to validly decrypt it. When you're using authenticated encryption and someone messes with your data it just fails to decrypt. (Tankersley 2016; Isom 2015)

```
nonce := make([]byte, gcm.NonceSize())
if _, err = rand.Read(nonce); err != nil {
    return nil, err
}
```

Before we can encrypt our bytes we need to generate a randomized `nonce`, and its length is specified by the GCM. The `nonce` stands for: *number once used*, and it's a piece of data that should not be repeated and only used once in combination with any particular key. Meaning: don't repeat the combination of a `key` and a `nonce` more than once. But, how do you keep track of that? If we use sufficiently large numbers for a `nonce` we should probably be fine for this use-case. (Isom 2015; Viega and Messier 2003, 134-35) We do that by using Go's `crypto/rand` package to read randomized bytes into the `nonce` byte slice.[7]

```
encryptedData := gcm.Seal(nonce, nonce, bData, nil)
```

The `nonce` that we're going to use for encrypting our data, is also needed to decrypt it. So we need to be able to refer to it while decrypting, and one of the strategies is to add it to the encrypted data. In this example we will prepend the `nonce` to the encrypted data. We do that by passing in the `nonce` as the first argument `dst` of the `Seal` function, and as such the encrypted data will be appended to it.[8] We can do this because the `nonce` doesn't have to be secret, it just has to be unique. (Tankersley 2016)

Decrypt

Now, we're able to encrypt our data, and let's implement the `Decrypt` function.

```
import (
    "crypto/aes"
    "crypto/cipher"
)

func Decrypt(key, data []byte) ([]byte, error) {
    blockCipher, err := aes.NewCipher(key)
    if err != nil {
        return nil, err
    }

    gcm, err := cipher.NewGCM(blockCipher)
    if err != nil {
        return nil, err
    }
```

```
    nonce, ciphertext := data[:gcm.NonceSize()],
data[gcm.NonceSize():]

    plaintext, err := gcm.Open(nil, nonce, ciphertext, nil)
    if err != nil {
        return nil, err
    }

    return plaintext, nil
}
```

Again let's go over the code and check what it does. It is largely the same code as the `Encrypt` function, so let's inspect the parts that differ.

```
nonce, ciphertext := data[:gcm.NonceSize()], data[gcm.NonceSize():]
```

Remember from the last section, that we prepended the `nonce` to the `data` using `gcm.Seal` to create the `ciphertext`? Now we need to split those parts so we can use them independently. And we're creating those part by slicing the `data` based on the size of the `nonce` that `gcm` provides.

```
plaintext, err := gcm.Open(nil, nonce, ciphertext, nil)
```

Now, we're using `gcm.Open` to decrypt the `ciphertext` into `plaintext`.[9]

Key

We've been passing in a `key` to both the `Encrypt` and `Decrypt` functions, but we have yet to make it, so let's do that.

```
import (
    "crypto/rand"
)

func GenerateKey() ([]byte, error) {
    key := make([]byte, 32)

    _, err := rand.Read(key)
    if err != nil {
        return nil, err
    }

    return key, nil
}
```

Here we're generating a random `key` using Go's `crypto/rand` package. For AES we need a `key` that has the length of 32 bytes, so we make a byte slice of size 32. Then we let `rand.Read()` fill the slice with random bytes.[10]

Now we have enough to encrypt and decrypt some data, so let's put it all together and test it out:

```
// crypto.go
package main

import (
    "crypto/aes"
    "crypto/cipher"
    "crypto/rand"
    "encoding/hex"
    "fmt"
    "log"
)

func Encrypt(key, data []byte) ([]byte, error) {
    blockCipher, err := aes.NewCipher(key)
    if err != nil {
        return nil, err
    }

    gcm, err := cipher.NewGCM(blockCipher)
    if err != nil {
        return nil, err
    }

    nonce := make([]byte, gcm.NonceSize())
    if _, err = rand.Read(nonce); err != nil {
        return nil, err
    }

    ciphertext := gcm.Seal(nonce, nonce, data, nil)

    return ciphertext, nil
}

func Decrypt(key, data []byte) ([]byte, error) {
    blockCipher, err := aes.NewCipher(key)
    if err != nil {
        return nil, err
    }

    gcm, err := cipher.NewGCM(blockCipher)
    if err != nil {
        return nil, err
    }

    nonce, ciphertext := data[:gcm.NonceSize()],
data[gcm.NonceSize():]
```

```
    plaintext, err := gcm.Open(nil, nonce, ciphertext, nil)
    if err != nil {
        return nil, err
    }

    return plaintext, nil
}

func GenerateKey() ([]byte, error) {
    key := make([]byte, 32)

    _, err := rand.Read(key)
    if err != nil {
        return nil, err
    }

    return key, nil
}

func main() {
    data := []byte("our super secret text")

    key, err := GenerateKey()
    if err != nil {
        log.Fatal(err)
    }

    ciphertext, err := Encrypt(key, data)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("ciphertext: %s\n", hex.EncodeToString(ciphertext))

    plaintext, err := Decrypt(key, ciphertext)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("plaintext: %s\n", plaintext)
}
```

And, we can run this example with:

```
$ go run crypto.go
```

Now, we have enough to encrypt and decrypt our `data` with a randomized key. This is cool and now we have a `key` that allows us to encrypt and decrypt our `data`. But that means that the `key` now becomes our password and weren't able to choose it ourselves, additionally it has a length of 32 bytes.

But, as mentioned in the start of the post, we want to be able to encrypt and decrypt the data by providing our own `key` namely a password that we've chosen to use. We will be doing that in the following section.

Password

Now, the `aes.NewCipher()` needs a 16, 24, or a 32 byte `key`, and in this example we are using a 32 byte `key`. However, our password likely isn't going to be 32 bytes. So we need to transform our password to a suitable `key`. And we do that by using a *key derivation function* (KDF)[11] to 'stretch' the password to make it a suitable cryptographic key. This key-stretching[12] characterizes itself by being slow. This is done in order to make it that, an attacker needs to spend a lot of resources to attempt to brute force an attack the on the password. We have several options for KDF's: Argon2[13], scrypt[14], bcrypt[15], and pbkdf2[16]. Choosing one depends on several factors, but mainly how safe it is.[17] [18] [19] [20] [21] [22]

Typically in a KDF we have a `password`, a `salt`, and an `iterations` argument. The `salt`[23] is used to prevent an attacker from just storing password/key pairs, and prevents an attacker from precomputing a dictionary of derived keys, as a different salt yields a different output. Each password has to be checked with the `salt` used to derive the key. (Isom 2015; Wikipedia 2020) The `salt` is related to the `nonce` in that it also needs to be randomly generated. And as with the `nonce`, the `salt` doesn't need to be secret, it needs to be unique. The `iterations` argument or the *difficulty parameter*, signifies how many times to repeat the process. This is because, even with `salt`, a *dictionary attack* is still possible, but with the `iterations` count, it will slow down the time it takes to compute a `key` from a password. (Viega and Messier 2003, 141-42)

In this example we'll be using *scrypt*, so let's see how we can implement that into our program.

```
import (
    "crypto/rand"

    "golang.org/x/crypto/scrypt"
)

func DeriveKey(password, salt []byte) ([]byte, []byte, error) {
    if salt == nil {
        salt = make([]byte, 32)
        if _, err := rand.Read(salt); err != nil {
            return nil, nil, err
        }
    }
```

```
    key, err := scrypt.Key(password, salt, 1048576, 8, 1, 32)
    if err != nil {
        return nil, nil, err
    }

    return key, salt, nil
}
```

Again let's go over the code and see what it does.

```
func DeriveKey(password, salt []byte) ([]byte, []byte, error)
```

Here we accept the password as a slice of bytes as the argument, and we return the resulting `key`, and `salt`.

```
salt := make([]byte, 32)
if _, err := rand.Read(salt); err != nil {
    return err
}
```

Just like our `Encrypt` function, we'll be creating the `salt` with 32 random bytes.

```
key, err := scrypt.Key(password, salt, 1048576, 8, 1, 32)
```

Here we're using the `scrypt` package from `golang.org/x/` library.[24] From the documentation we can read that the `Key` function accepts the following arguments:

```
func Key(password, salt []byte, N, r, p, keyLen int) ([]byte, error)
```

The arguments `password` and `salt` speak for themselves. `N` is the number of iterations. In a presentation given by C. Percival it is recommended that for interactive logins 16384 (2^14) iterations, and for file encryption 1048576 (2^20) iterations are used. (Percival 2005a, 2005b; Isom 2015) The arguments `r` and `p` must satisfy $r * p < 2^{30}$, if it doesn't satisfy the limits, the function returns a `nil` byte slice and an error. (Golang Documentation 2020). The `r` argument defines the relative memory cost parameter it controls the blocksize in the underlying hash, the recommended value is 8. The `p` argument is the relative CPU cost parameter and the recommended value for this is 1.

(Isom 2015; Percival 2005a) The `keyLen` argument defines the length of the bytes that are returned as key, as discussed this will be 32 bytes.

Result

Now that we've created our `DeriveKey` function we need to update our code to support it. So let's do that, it should resemble the code below:

```
// scrypt.go
package main

import (
    "crypto/aes"
    "crypto/cipher"
    "crypto/rand"
    "crypto/sha256"
    "encoding/hex"
    "fmt"
    "log"

    "golang.org/x/crypto/scrypt"
)

func Encrypt(key, data []byte) ([]byte, error) {
    key, salt, err := DeriveKey(key, nil)
    if err != nil {
        return nil, err
    }

    blockCipher, err := aes.NewCipher(key)
    if err != nil {
        return nil, err
    }

    gcm, err := cipher.NewGCM(blockCipher)
    if err != nil {
        return nil, err
    }

    nonce := make([]byte, gcm.NonceSize())
    if _, err = rand.Read(nonce); err != nil {
        return nil, err
    }

    ciphertext := gcm.Seal(nonce, nonce, data, nil)

    ciphertext = append(ciphertext, salt...)

    return ciphertext, nil
}

func Decrypt(key, data []byte) ([]byte, error) {
    salt, data := data[len(data)-32:], data[:len(data)-32]
```

```
    key, _, err := DeriveKey(key, salt)
    if err != nil {
        return nil, err
    }

    blockCipher, err := aes.NewCipher(key)
    if err != nil {
        return nil, err
    }

    gcm, err := cipher.NewGCM(blockCipher)
    if err != nil {
        return nil, err
    }

    nonce, ciphertext := data[:gcm.NonceSize()],
data[gcm.NonceSize():]

    plaintext, err := gcm.Open(nil, nonce, ciphertext, nil)
    if err != nil {
        return nil, err
    }

    return plaintext, nil
}

func DeriveKey(password, salt []byte) ([]byte, []byte, error) {
    if salt == nil {
        salt = make([]byte, 32)
        if _, err := rand.Read(salt); err != nil {
            return nil, nil, err
        }
    }

    key, err := scrypt.Key(password, salt, 1048576, 8, 1, 32)
    if err != nil {
        return nil, nil, err
    }

    return key, salt, nil
}

func main() {
    var (
        password = []byte("mysecretpassword")
        data     = []byte("our super secret text")
    )

    ciphertext, err := Encrypt(password, data)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("ciphertext: %s\n", hex.EncodeToString(ciphertext))

    plaintext, err := Decrypt(password, ciphertext)
    if err != nil {
        log.Fatal(err)
    }
```

```
    fmt.Printf("plaintext: %s\n", plaintext)
}
```

And, we're able to run and test it:

```
# First we need to get the scrypt package
$ go get -u golang.org/x/crypto/scrypt

$ go run scrypt.go
```

We've updated some parts, so let's go over it.

```
key, salt, err := DeriveKey(key, nil)
```

In the `Encrypt` function we create our key by passing in our password, which is contained in the `key` argument. We pass in `nil` as the salt argument, that is because we want to create the `salt` since it is the first time we encrypt our `data`.

```
ciphertext = append(ciphertext, salt...)
```

Additionally, in the `Encrypt` function, we append the `salt` to our `ciphertext`.

```
salt, data := data[len(data)-32:], data[:len(data)-32]
```

And, because we append the `salt` to the `ciphertext`, we need to split and slice it in the `Decrypt` function, because we're going to use it in the `DeriveKey` function.

```
key, _, err := DeriveKey(key, salt)
```

As you can see here we pass in the salt to the `DeriveKey` function and we'll be able to retrieve the `key` that we used in order to encrypt our data.

Conclusion

With that, we've created two ways in order to encrypt and decrypt our data in Go. First we've encrypted our data by using the AES encryption algorithm, for which we've

created a randomized key to be used for decrypting our data. Subsequently, we've updated our code to support using a password as our key. We've done that by *key-stretching* our password using a *key derivation function*, and we've used *scrypt* to achieve that. Hopefully, you found this post useful, and again I advice you to read and watch the sources that I've listed, and check out other sources to get a good overview on how to correctly and securely encrypt your data, and if you have any suggestions let me know.

. . .

References

Golang Documentation. 2020. "Package Scrypt." 2020. https://godoc.org/golang.org/x/crypto/scrypt.

Isom, Kyle. 2015. *Practical Cryptography with Go*. Leanpub. https://leanpub.com/gocrypto/read.

Percival, C. 2005a. "scrypt: A new key derivation function." https://www.bsdcan.org/2009/schedule/attachments/86_scrypt_slides.pdf.

— -. 2005b. "Strong Key Derivation via Sequential Memory-Hard Functions." https://www.tarsnap.com/scrypt/scrypt.pdf.

Ryer, M. 2015. "io.Reader in Depth." 2015. https://medium.com/@matryer/golang-advent-calendar-day-seventeen-io-reader-in-depth-6f744bb4320b.

Tankersley, G. 2016. "Go for Crypto Developers." https://youtu.be/2r_KMzXB74w.

Viega, J., and M. Messier. 2003. *Secure Programming Cookbook for C and C++*. O'Reilly Media.

Wikipedia. 2020. "Key Derivation Function — Wikipedia, the Free Encyclopedia." 2020. https://en.wikipedia.org/wiki/Key_derivation_function.

. . .

Footnotes

[1] Some references on how to implement encryption with streams: Reddit Comment; Michael Turner — Encrypting Streams in Go; Minio — Github Issue

[2] Go Documentation: AES

[3] Wikipedia: AES

[4] Computerphile — AES Explained

[5] Wikipedia: GCM

[6] Go Documentation: AEAD

[7] Go Documentation: rand.Read

[8] Go Source Code: GCM

[9] Go Source Code: GCM

[10] Go Documentation: rand.Read

[11] Wikipedia: KDF

[12] Wikipedia: Key Stretching

[13] Wikipedia: Argon2

[14] Wikipedia: scrypt

[15] Wikipedia: bcrypt

[16] Wikipedia: pbkdf2

[17] Michele Prezioso — Password Hashing: PBKDF2, Scrypt, Bcrypt

[18] Michele Prezioso — Password Hashing: Scrypt, Bcrypt and ARGON2

[19] StackExchange — What's the difference between PBKDF and SHA and why use them together?

[20] Github — Bitwarden Issue 52

[21] Github — Bitwarden Issue 589

[22] Wikipedia: pbkdf2 alternatives

[23] Wikipedia: Salt (cryptography)

[24] Go Documentation: scrypt

. . .

Originally published at https://bruinsslot.jp/post/golang-crypto/.