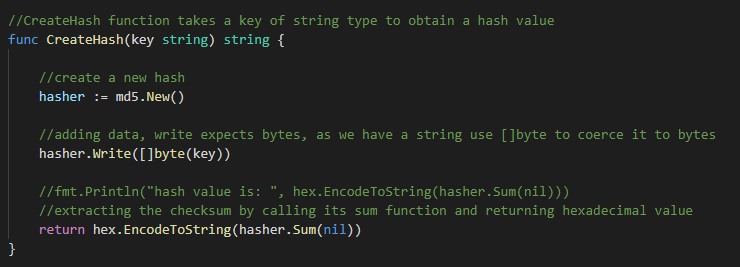
**ASSIGNMENT: STRING WORLD**

Encrypt the 8char string into 16 char string, decryption of the string, finding the length of the decrypted string, compare the input string and the decrypted string.

Hashing passwords:

For the encryption or decryption of the data, it is very important to make use of a 32 character or a 32 byte key. Being realistic, we might want to make use of a passphrase, and this passphrase will never be 32 characters in length.

To get around this, we can actually hash the passphrase by making use of any of the hashing algorithms that are available and that produces 32 byte hashes. Many algorithms like MD5, SHA1 etc are available. Out of all these, MD5 is chosen. This is a simple hash. It is insecure, but we can make use of this as we are not storing the output.



The CreateHash function will take a passphrase as an input and it evaluates the hash value and returns a hexadecimal value of it.

GO implements several hash functions in various crypto/\* packages.

For MD5, the package that needs to be imported is “crypto/md5”

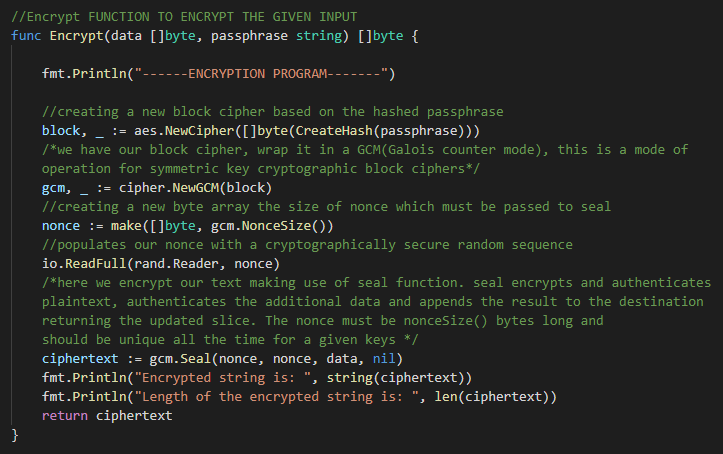
The pattern for generating a hash is md5.New( ), md5.Write(bytes) and then md5.Sum([]byte). First step is to start with a new hash.

Write expects bytes, so if we have a string “key”, use Write([]byte(key)) to coerce it to bytes.

This will get the finalized hash result as a byte slice. The argument to sum is used to append data to the existing slice, but generally it is not that needed and we put it as nil. It returns a hexadecimal value.

Data Encryption with an AES Cipher

Now, we have got a key of an appropriate size. So we can start with the encryption process.



The above shown is the code for encrypting the data. A function named “encrypt” is written for which the arguments passed are: data that we want to encrypt and also the passphrase which is the key.

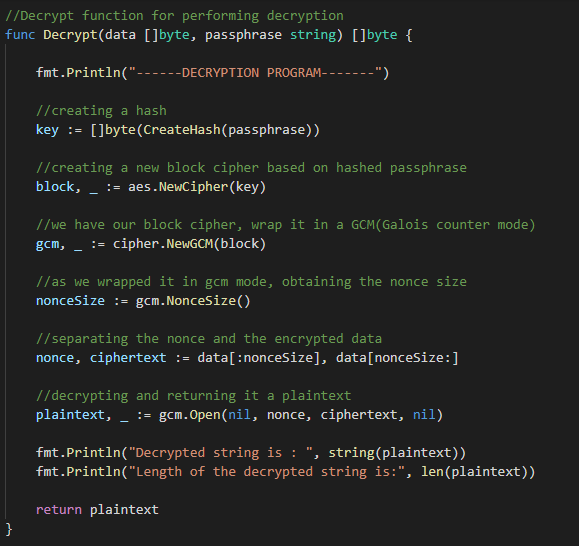
Firstly, we create a new block cipher by making use of hashed passphrase.

Once, we have our block, wrap it in GCM (Galois Counter Mode) mode with a standard nonce length. This is a mode used for the operation of symmetric key cryptographic block ciphers.

Before the creation of ciphertext, we have to create a nonce. This nonce that we create should be of the length which is specified by GCM. One important factor here is that the nonce that we use for encryption should be the same for decryption.

In order to match there are few strategies. First one is, if the encrypted data is going into the database, then store that nonce. The other one is prepend or append the nonce to the encrypted data. Here, in the above code we followed the second approach.

Encryption is achieved by making use of seal function. The first parameter given is the prefix value and we are appending the data to the nonce which are the second and third parameters. With this ciphertext will be obtained and is printed on the console using println.



The above shown is the decrypt function.

This is similar to that of the encryption function like we create a new block and wrap it in GCM mode and will create a nonce of specified length.

As we appended the data to the nonce in the encryption process, so while decrypting we need to separate them. Here the data will come into the nonce from starting till the noncesize and the ciphertext will be extracted from noncesize to the end.

Once the separation is done, by making use of the open function the decryption is done and the plaintext will be obtained.

The length of the decrypted string is obtained by function “len” which is available in strings package.

Comparison of the input string and the decrypted string is done by “strings.compare” function which is available in strings package. If the two strings that are passed to the function are equal then it returns a “0” otherwise “1” or “-1” where the strings are not equal.

**Testing:**

In the testing which is “\_test.go” file, a struct is being created which carries out table test where different kinds of inputs are given and the output is observed accordingly.

In this program we took the passphrase as “password” and the hash value is obtained for that. The hash value is kept as reference and checking whether the passphrase is same or not.

Case 1: If the passphrase is “password” then whatever the input we give should be same as the output in that “testcases” struct, the encryption and decryption are done appropriately and the output is displayed saying the strings are equal.

Case 2: If the passphrase is something other than “password” then the encryption will fail as the passphrase doesn’t match.

Case 3: If the passphrase matches i.e., if it is given as “password” and a word is given as an input which is in lower case whereas the expected output is the same word but is in the upper case then also we get an error, the test will fail.

For the cases that are mentioned above, the encryption and decryption process has been verified.

Code coverage is 100%