ECE 518 Project 1

Alan Palayil Due Date: 10/02/2022

Section I Overview

This project is used to understand how encryption process takes place. We are provided with the information of a 256-bit key to find a 4-digit password using SHA256 and AESGCM cryptographic hash functions, we use the Go programming language to figure out the 4-digit password using brute-force computation.

Section II Cryptographic Hash Functions and Ciphers

1. From the output of validateSHA256(), decide the length of the hash generated by SHA256. Is it as expected?
   - The length in the output of validateSHA256(): **c0535e4be2b79ffd93291305436bf889314e4a3faec05ecffcbb7df31ad9e51a** which is 32bytes and SHA256() should have 32bytes or 256bits.
2. What is the length of the nonce used for AES-GCM?
   - The length of the nonce used for AES-GCM: **3fac2e807972e1c33abfe433** is 12bytes or 96bits.
3. What is the length of the plaintext in validateAESGCM()?
   - The length of the plaintext in validateAESGCM() is **12bytes** as each character is in the string is 8bits long.
4. What is the length of the byte slice returned by the Seal function? Why is it named ciphermac?
   - The length of the byte slice returned by the Seal function is **28 bytes**. The reason why we use ciphermac as the name is because we use MAC in the cryptographic checksum on the data along with the session key to detect accidental and intentional modification in the data.

Section III Find the Password

In the findPassword() function with the for loop provided by the professor, we can find the correct password by generating all possible 4-digit password and compare the hashes of each of the generated password.

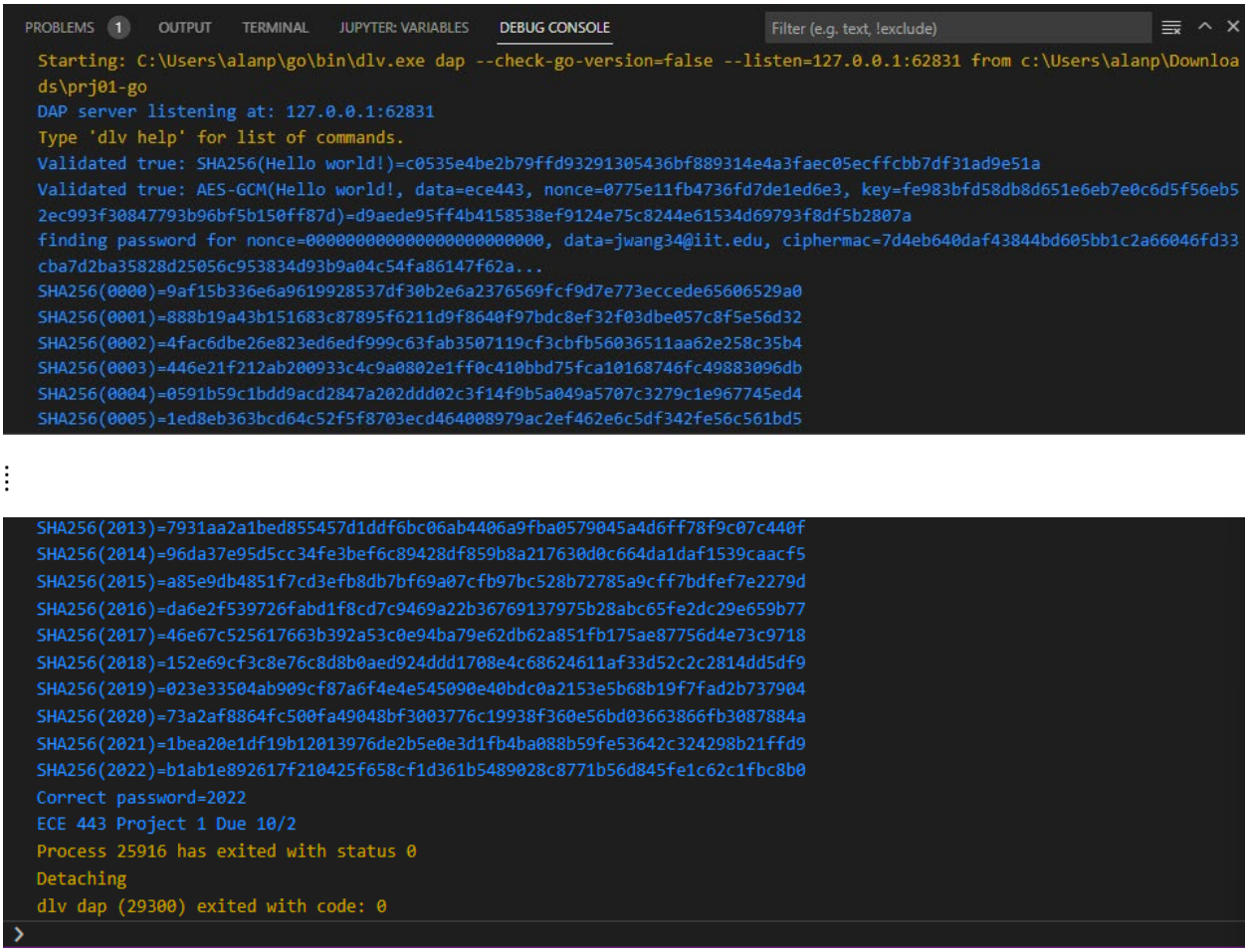
This implementation uses brute-force computation. We create a new SHA hash with each iteration of the 4-digit password during the for loop. We open the hash and use the functions

provided in validateAESGCM() and compare it with the original ciphermac. Using the provided nonce, we can use the aewsgcm.Open() function which is the decryption for the seal function. The loop breaks once we reach the correct 4-digit password.

The following is the code used within the findPassword() function with the comments to explain the lines:

```
func findPassword() {
    nonce := make([]byte, 12)
    data := "jwang34@iit.edu"

    ciphermac, _ := hex.DecodeString(
        "7d4eb640daf43844bd605bb1c2a66046fd33cba7d2ba35828d25056c953834d93b9a04c5
4fa86147f62a")

    fmt.Printf("finding password for nonce=%x, data=%s, ciphermac=%x...\n",
        nonce, data, ciphermac)

    for i := 0; i < 10000; i++ {
        password := fmt.Sprintf("%04d", i)
        /*
            Modify code below to check that if sha256(password) is a correct
            key to decrypt the ciphertext with MAC in ciphermac, which is
            obtained via AES-GCM with a 0 nouce and the data being
jwang34@iit.edu
            Once found, show the correct password as well as the plaintext.
        */
        sha := sha256.New() //Creating a new SHA256
        sha.Write([]byte(password)) //SHA of the new password
        hash := sha.Sum(nil) //SHA hash
        fmt.Printf("SHA256(%s)=%x\n",

            password, hash) //Printing each 4-digit password and hash
        block, _ := aes.NewCipher(hash) //From AES function in validate.go
        aesgcm, _ := cipher.NewGCM(block) //From AES function in validate.go
        plaintext, error := aesgcm.Open(nil, nonce, []byte(ciphermac),
[]byte(data)) //From AES function in validate.go
        if error == nil {
            fmt.Printf("Correct password=%s\n", password)
            fmt.Printf("%s\n", string(plaintext))
            break
        }
    }
}
```

Output:



⋮

Section IV Bonus: Performance Evaluation

- Time the process to compute the SHA256 hash of a 16-byte message.
  - The following is the code used for computing the time to process the SHA256 hash of a 16-byte message.

    We create a SHA256time() function, in which a 16-byte message and a variable t for the duration of time. In the for loop, we start the time and create the SHA256 hash of message. Variable t is updated once the SHA is created, and we print it the time it takes to process using the Printf function and format the print.

Code:

```
func SHA256time() {
    message := make([]byte, 16) //Creating a 16-bit message
    var t time.Duration = 0     //Variable t for the duration of computation
```

```
    for i := 0; i < 100000; i++ {
        start := time.Now()        //Start time
        sha := sha256.New()        //Creating a new SHA256
        sha.Write([]byte(message)) //Taking the SHA of the message
        t += time.Since(start)
    }
    fmt.Printf("SHA256 hash computation Time=%s\n", (t / 100000)) // Formatted
string as "2.5ns"
}
```

Output:

From looking over the data about SHA256 hashing function, its implementation goes around 45ns for a single unpadded single block (16-byte message) and thus the performance of SHA256 computation is as expected since we got 49ns.

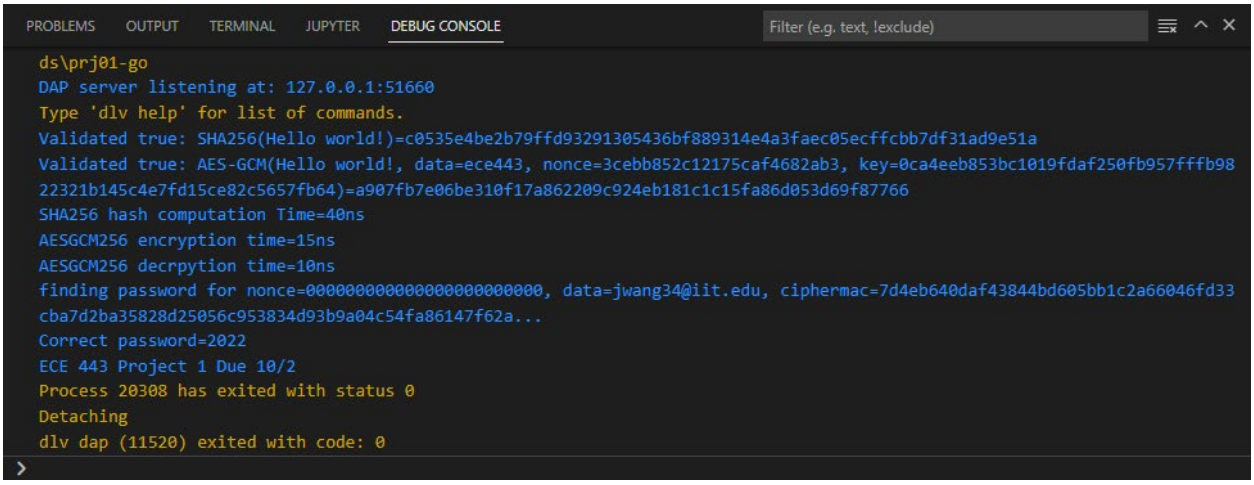
- Time the process to encrypt a 1M-byte message (1024*1024 bytes) with AES-GCM using 256-bit AES key and no additional data.
    - ➢ We create the AESGCM256time() function and declare 1M-byte message along with a SHA256 hash. We take functions from validateAESGCM() to encrypt the SHA hash. Variable e and d are created to calculate the time taken for encryption and decryption using a for loop. We print it the time it takes to encrypt and decrypt using the Printf function and format the print.

Code:

```
func AESGCM256time() {
    password := fmt.Sprintf("%04d", 1234)
    plaintext := make([]byte, 1024*1024) //Creating 1M-byte message
    rand.Read(plaintext)                 //From AES function in validate.go
    sha := sha256.New()                  //Creating a new SHA256
    sha.Write([]byte(password))          //Taking the SHA of the password
```

```
    hash := sha.Sum(nil)                    //Creating a hash for the SHA
    c, _ := aes.NewCipher(hash)             // create a cipher
    out := make([]byte, 1024*1024)          // allocate space for ciphered data
    var e time.Duration = 0                 //Variable e for the duration of
encryption
    var d time.Duration = 0                 //Variable t for the duration of
decryption
    for i := 0; i < 10000; i++ {
        en_start := time.Now() //Start time
        c.Encrypt(out, []byte(plaintext))
        e += time.Since(en_start) //Add encryption time for the process
        pt := make([]byte, len(out))
        de_start := time.Now()
        c.Decrypt(pt, out)
        d += time.Since(de_start) //Add decryption time for the process
    }
    fmt.Printf("AESGCM256 encryption time=%s\n", (e / 100000)) // Formatted
string as "2.5ns"
    fmt.Printf("AESGCM256 decrpytion time=%s\n", (d / 100000)) // Formatted
string as "2.5ns"
}
```

Output:



From searching the data online, the data states that a 1-M-byte message takes around 20ns and depending on the CPU the value of the encryption changes. The processing time for the code is 15ns which does come around the expected performance.

- Time the process to decrypt the message above with AES-GCM using 256-bit AES key and no additional data.

➢ Using the code and output from the above AES-GCM encryption, the decryption time is usually around half the time of encryption. Our decrypting time is calculated as 10ns.