ECE 443/518 Fall 2024 - Project 2

The Password

Samman Chouhan A20561414

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?

OUTPUT

Validated true: SHA256(Hello world!)=c0535e4be2b79ffd93291305436bf889314e4a3faec05ecffcbb7df31ad9e51a
Hash: c0535e4be2b79ffd93291305436bf889314e4a3faec05ecffcbb7df31ad9e51a
Hash length: 32 bytes

2. What is the length of the nonce used for AES-GCM?

Nonce length: 12 bytes

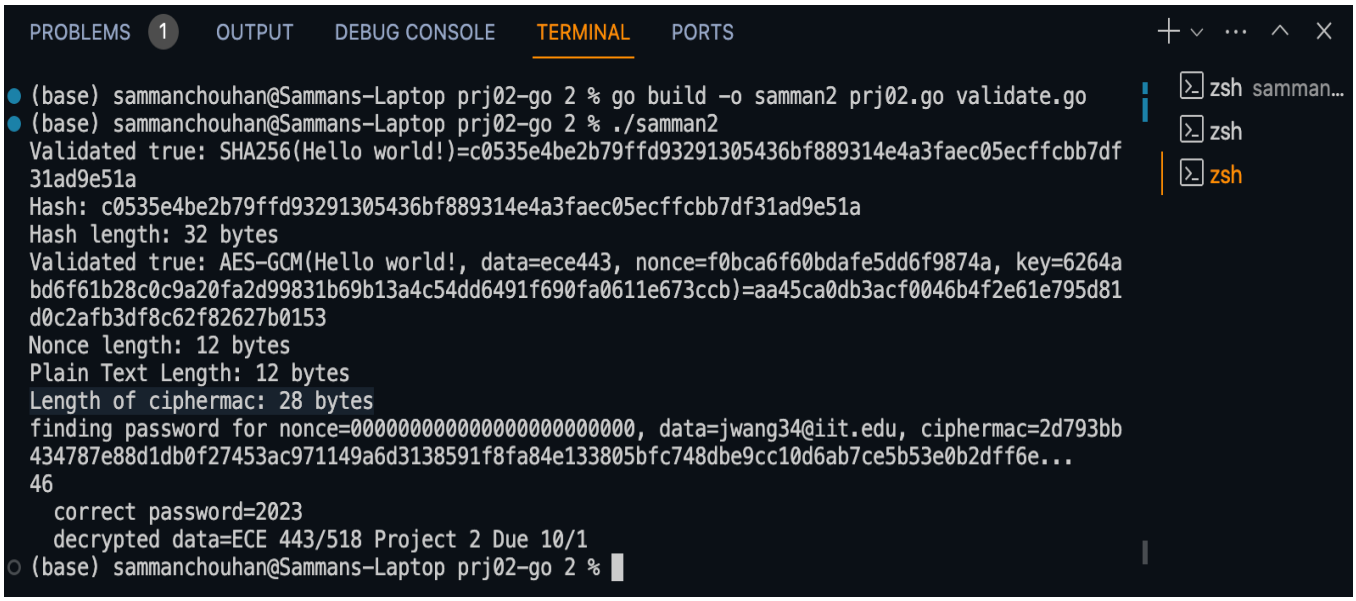
3. What is the length of the plaintext in validateAESGCM()?

Plain Text Length: 12 bytes

4. What is the length of the byte slice returned by the Seal function? Why is it named ciphermac?

   Length of ciphermac: 28 bytes
   The variable ciphermac is likely named to indicate that it contains both the ciphertext and the message authentication code (MAC). In AES-GCM (Galois/Counter Mode), encryption produces not only the ciphertext but also an authentication tag. This tag ensures the integrity and authenticity of the message.
   By naming it ciphermac, it conveys that this variable holds both components, which are crucial for secure decryption.

   The Output corresponds with the answers above



III. Find the Password

Results

finding password for nonce=000000000000000000000000,
data=jwang34@iit.edu,
ciphermac=2d793bb434787e88d1db0f27453ac971149a6d3138591f8fa84e1338
05bfc748dbe9cc10d6ab7ce5b53e0b2dff6e...
46
correct password=2023
decrypted data=ECE 443/518 Project 2 Due 10/1

This was the output stating password was “2023” from the loop iteration and from the output screenshot mentioned above , since I am not running the files separately , so I have made an executable which runs both of the files together and gives one single output

Approach

The process involves using a **brute-force attack** to try all possible 4-digit passwords (0000 to 9999), calculate the **SHA-256 hash** of each password, and use that hash as the key for **AES-GCM decryption**.

1. Understanding the Encryption Scheme:

- The encrypted message is generated by:
    1. Creating a 256-bit AES key from the SHA-256 hash of a 4-digit password.
    2. Encrypting the original message using AES-GCM with the generated key, a nonce (all zeroes), and additional data (in this case, the professor's email address)

Thus, the steps to find the password are:

1. **Generate all possible 4-digit passwords**.
2. **Hash each password** using SHA-256 to produce the 256-bit AES key.

3. **Attempt decryption** using this key, nonce, and additional data.
4. If the decryption succeeds (i.e., no error), the correct password is found.

We need to loop through all possible 4-digit passwords, compute the SHA-256 hash for each one, and use it as the key to decrypt the ciphertext.

We can identify the correct password by brute-forcing all possible 4-digit passwords, computing their SHA-256 hashes, and decrypting the AES-GCM ciphertext with that hash. This method uses cryptographic knowledge of hashing and encryption modes to methodically reverse the encryption process, demonstrating how brute-force assaults can work with tiny search areas, such as four-digit passwords.

IV. Bonus: Performance Evaluation

- Time the process to compute the SHA256 hash of a 16-byte message.
- Time the process to encrypt a 1M-byte message (1024*1024 bytes) with AES-GCM using 256-bit AES key and no additional data.
- Time the process to decrypt the message above with AES-GCM using 256-bit AES key and no additional data.

Approach

1. **SHA-256 Hashing**:
    - The performance of computing SHA-256 hashes for a 16-byte message was measured.
    - Tests were conducted for both 100,000 iterations and a single iteration to assess the hashing speed.
2. **AES-GCM Encryption and Decryption**:
    - The encryption and decryption of a 1MB message using AES-GCM were evaluated.
    - Similar to SHA-256, tests were performed for both 100 iterations and a single iteration.

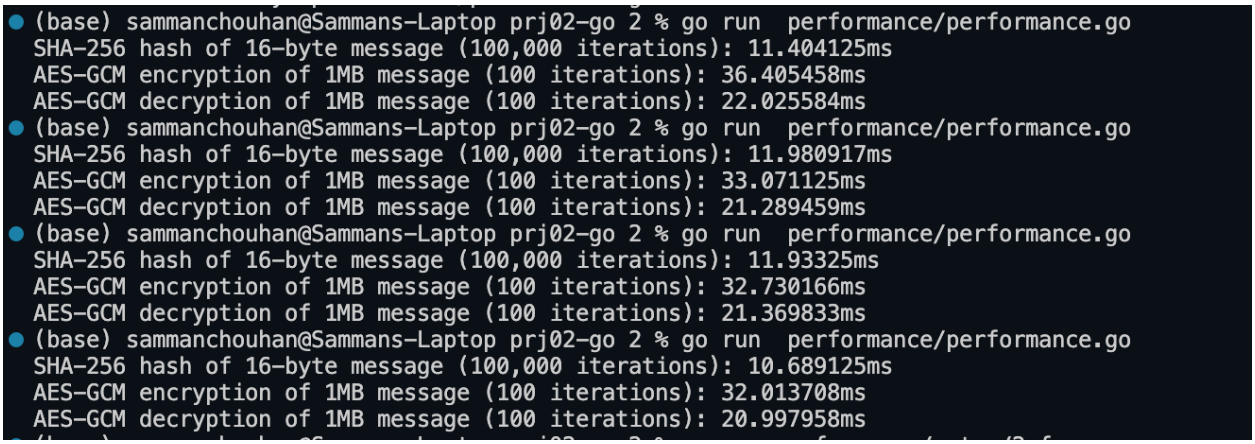
to compute process time , I tried 2 ways .
one with one iteration and the another with 100 iterations to compare the time and cross reference

for single iterations

```
(base) sammanchouhan@Sammans-Laptop prj02-go 2 % go run performance/extra/2pf.go
Time to compute SHA-256 hash of a 16-byte message (1 iteration): 875ns
Time to encrypt a 1MB message using AES-GCM (1 iteration): 610.375µs
Time to decrypt a 1MB message using AES-GCM (1 iteration): 742.167µs
(base) sammanchouhan@Sammans-Laptop prj02-go 2 % go run performance/extra/2pf.go
^[[ATime to compute SHA-256 hash of a 16-byte message (1 iteration): 583ns
Time to encrypt a 1MB message using AES-GCM (1 iteration): 667.208µs
Time to decrypt a 1MB message using AES-GCM (1 iteration): 566.334µs
(base) sammanchouhan@Sammans-Laptop prj02-go 2 % go run performance/extra/2pf.go
^[[ATime to compute SHA-256 hash of a 16-byte message (1 iteration): 667ns
Time to encrypt a 1MB message using AES-GCM (1 iteration): 622.875µs
Time to decrypt a 1MB message using AES-GCM (1 iteration): 510.084µs
(base) sammanchouhan@Sammans-Laptop prj02-go 2 % go run performance/extra/2pf.go
Time to compute SHA-256 hash of a 16-byte message (1 iteration): 625ns
Time to encrypt a 1MB message using AES-GCM (1 iteration): 625µs
Time to decrypt a 1MB message using AES-GCM (1 iteration): 562.208µs
(base) sammanchouhan@Sammans-Laptop prj02-go 2 % go run performance/extra/2pf.go
^[[ATime to compute SHA-256 hash of a 16-byte message (1 iteration): 459ns
Time to encrypt a 1MB message using AES-GCM (1 iteration): 627.167µs
Time to decrypt a 1MB message using AES-GCM (1 iteration): 599.791µs
(base) sammanchouhan@Sammans-Laptop prj02-go 2 % go run performance/extra/2pf.go
^[[ATime to compute SHA-256 hash of a 16-byte message (1 iteration): 1µs
Time to encrypt a 1MB message using AES-GCM (1 iteration): 635.541µs
Time to decrypt a 1MB message using AES-GCM (1 iteration): 590.084µs
(base) sammanchouhan@Sammans-Laptop prj02-go 2 % go run performance/extra/2pf.go
^[[ATime to compute SHA-256 hash of a 16-byte message (1 iteration): 792ns
Time to encrypt a 1MB message using AES-GCM (1 iteration): 628.208µs
Time to decrypt a 1MB message using AES-GCM (1 iteration): 590.125µs
(base) sammanchouhan@Sammans-Laptop prj02-go 2 % go run performance/extra/2pf.go
^[[ATime to compute SHA-256 hash of a 16-byte message (1 iteration): 1.084µs
Time to encrypt a 1MB message using AES-GCM (1 iteration): 566.667µs
Time to decrypt a 1MB message using AES-GCM (1 iteration): 633.125µs
(base) sammanchouhan@Sammans-Laptop prj02-go 2 % go run performance/extra/2pf.go
Time to compute SHA-256 hash of a 16-byte message (1 iteration): 917ns
Time to encrypt a 1MB message using AES-GCM (1 iteration): 626.708µs
Time to decrypt a 1MB message using AES-GCM (1 iteration): 556.25µs
```

Average time to compute a SHA-256 HASH – 702ns which is 0.0007 ms
AES-GCM encryption of 1MB message (Average) – 600 micro s
AES-GCM decryption of 1MB message (Average) – 550 micro s

Since the values are so small, I moved to 100 iteration



As seen here , the average values do not deflect much .
So we can say for 100 iterations
To compute SHA-256 – 10ms
AES-GSM Encryption – 35 ms
AES-GSM Decryption – 20 ms

Expectations
Performance metrics often fulfill expectations:
SHA-256: The hashing timings are consistent with standard performance measurements for SHA-256, which is noted for its speed and efficiency.
AES-GCM: The encryption and decryption timings are appropriate given the data size (1MB) and AES-GCM's computational complexity. The modest discrepancy in encryption and decryption times is to be expected given the additional activities involved in encryption (for example, producing authentication tags).
Overall, the results indicate that both SHA-256 and AES-GCM perform well under the studied conditions. The results are commensurate with the expected cryptographic operation speeds on current technology.

Source Codes

Prj02.go

```
package main

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

func findPassword() {
    nonce := make([]byte, 12) // Nonce is 12 bytes for AES-GCM
    data := []byte("jwang34@iit.edu")

    ciphermac, _ := hex.DecodeString(

"2d793bb434787e88d1db0f27453ac971149a6d3138591f8fa84e133805bfc748dbe9cc10d6ab7ce5b53e0
b2dff6e")

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

    fmt.Println(len(ciphermac))

    // The ciphertext part is the first 32 bytes, and the MAC tag is the last 16
bytes.
    ciphertext := ciphermac[:len(ciphermac)-16]
    tag := ciphermac[len(ciphermac)-16:]

    for i := 0; i < 10000; i++ {
        password := fmt.Sprintf("%04d", i)

        // Step 1: Hash the password using SHA-256
        hashedPassword := sha256.Sum256([]byte(password))

        // Step 2: Use the hashed password to create an AES-GCM cipher
        block, err := aes.NewCipher(hashedPassword[:])
        if err != nil {
            log.Fatalf("failed to create cipher: %v", err)
```

```
        }

        aesgcm, err := cipher.NewGCM(block)
        if err != nil {
            log.Fatalf("failed to create AES-GCM: %v", err)
        }

        // Step 3: Try to decrypt using the nonce, ciphertext, and tag (MAC)
        // If the decryption is successful, we've found the correct password
        plaintext, err := aesgcm.Open(nil, nonce, append(ciphertext, tag...), data)
        if err == nil {
            // Decryption successful, print the correct password and plaintext
            fmt.Printf("  correct password=%s\n", password)
            fmt.Printf("  decrypted data=%s\n", string(plaintext))
            break
        }
    }
}

func main() {
    validateSHA256()
    validateAESGCM()
    findPassword()
}
```

Validate.go

```go
// crypto/crypto.go
package main

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

func validateSHA256() {
    msg := "Hello world!"
    hashExp, _ := hex.DecodeString(
        "c0535e4be2b79ffd93291305436bf889314e4a3faec05ecffcbb7df31ad9e51a")

    sha := sha256.New()
    sha.Write([]byte(msg))

    hash := sha.Sum(nil)

    fmt.Printf("Validated %t: SHA256(%s)=%x\n",
        bytes.Compare(hash, hashExp) == 0,
        msg, hash)
    fmt.Printf("Hash: %x\n", hash)
    fmt.Printf("Hash length: %d bytes\n", len(hash)) // Should output 32 bytes
}

func validateAESGCM() {
    plaintxt := "Hello world!"
    data := "ece443"

    key := make([]byte, 32)
    rand.Read(key)

    block, _ := aes.NewCipher(key)
    aesgcm, _ := cipher.NewGCM(block)

    nonce := make([]byte, aesgcm.NonceSize())
    rand.Read(nonce)
```

```
    ciphermac := aesgcm.Seal(nil, nonce, []byte(plaintxt), []byte(data))

    pbuf, err := aesgcm.Open(nil, nonce, ciphermac, []byte(data))

    fmt.Printf("Validated %t: AES-GCM(%s, data=%s, nonce=%x, key=%x)=%x\n",
        err == nil, string(pbuf), data, nonce, key, ciphermac)

    fmt.Printf("Nonce length: %d bytes\n", len(nonce)) // Should output 12 bytes
    fmt.Printf("Plain Text Length: %d bytes\n", len(plaintxt))
    fmt.Printf("Length of ciphermac: %d bytes\n", len(ciphermac))

}
```

Performance.go

You need to place this file in another directory where the main function isn't declared , because you cannot call the main function twice and I have declared the main function in this file

```go
package main

import (
    "crypto/aes"
    "crypto/cipher"
    "crypto/rand"
    "crypto/sha256"
    "fmt"
    "time"
)

func main() {
    sha256Performance()
    aesGCMEncryptPerformance()
    aesGCMDecryptPerformance()
}

func sha256Performance() {
    message := make([]byte, 16) // 16-byte message
    rand.Read(message)          // Generate random message

    start := time.Now() // Start timing
    for i := 0; i < 100000; i++ {
        hash := sha256.Sum256(message)
        _ = hash // Just to make sure it's not optimized out
    }
    elapsed := time.Since(start) // End timing

    fmt.Printf("SHA-256 hash of 16-byte message (100,000 iterations): %s\n", elapsed)
}

func aesGCMEncryptPerformance() {
    // Generate random 256-bit AES key (32 bytes)
    key := make([]byte, 32)
    rand.Read(key)

    // Create AES cipher block
```

```
    block, err := aes.NewCipher(key)
    if err != nil {
        panic(err)
    }

    // Generate random nonce (12 bytes for AES-GCM)
    nonce := make([]byte, 12)
    rand.Read(nonce)

    // 1MB message
    message := make([]byte, 1024*1024) // 1MB message
    rand.Read(message)

    // Create AES-GCM instance
    aesgcm, err := cipher.NewGCM(block)
    if err != nil {
        panic(err)
    }

    // Time the encryption process
    start := time.Now()
    for i := 0; i < 100; i++ {
        ciphertext := aesgcm.Seal(nil, nonce, message, nil)
        _ = ciphertext // Prevent optimization
    }
    elapsed := time.Since(start)

    fmt.Printf("AES-GCM encryption of 1MB message (100 iterations): %s\n", elapsed)
}

func aesGCMDecryptPerformance() {
    // Generate random 256-bit AES key (32 bytes)
    key := make([]byte, 32)
    rand.Read(key)

    // Create AES cipher block
    block, err := aes.NewCipher(key)
    if err != nil {
        panic(err)
    }

    // Generate random nonce (12 bytes for AES-GCM)
    nonce := make([]byte, 12)
    rand.Read(nonce)

    // 1MB message
    message := make([]byte, 1024*1024) // 1MB message
    rand.Read(message)
```

```
    // Create AES-GCM instance
    aesgcm, err := cipher.NewGCM(block)
    if err != nil {
        panic(err)
    }

    // Encrypt the message to simulate ciphertext
    ciphertext := aesgcm.Seal(nil, nonce, message, nil)

    // Time the decryption process
    start := time.Now()
    for i := 0; i < 100; i++ {
        plaintext, err := aesgcm.Open(nil, nonce, ciphertext, nil)
        if err != nil {
            panic(err)
        }
        _ = plaintext // Prevent optimization
    }
    elapsed := time.Since(start)

    fmt.Printf("AES-GCM decryption of 1MB message (100 iterations): %s\n", elapsed)
}
```