Cifrador seguro de fichero

En clase hemos aprendido sobre cifrado simetrico (DES, 3DES, AES), funciones de un solo sentido (hash), MACs y los conceptos de condiferencialidad, integridad y autentificación.

Estos algortimos son considerados primitivas criptográficas, que se pueden usar juntos para crear otros productos:

- Criptomonedas
- Blockchain
- Web segura (https)
- Encriptación de ficheros
- Almacenamiento de contraseñas
- Evitar la censura

Y tambien para actividades maliciosas:

- Randomware
- Terrorismo

Como ingenierios es importante conocer el impacto "del arma" que tenemos entre nuestras manos y usarla de forma responsable y segura.

Veremos con en la práctica se plantean muchos más interrogantes de los que a priori pueden parecer. Por ejemplo, para encriptar un fichero, podemos pensar que simplemente usando AES sería imposible que nos descifraran el mensaje, al fin y al cabo, todo internet considera AES seguro, bancos, google, facebook lo usan... ¿que podría salir mal? TODO.

Esta práctica guiada nos llevará por las decisiones que deberemos tomar para el diseño e implementación de un cifrador seguro. Una herramienta que encripte un fichero con una contraseña, pero mucho de lo aprendido aquí se podría aplicar a otros servicios y utilidades.

Algoritmo de bloques

En primero lugar deberemos elegir el algortimo de bloques a usar: - DES - 3DES - AES - El nuestro propio

> DES queda descartado por su pequeño tamaño de clave (56bits). Cualquier ordenador moderno es capaces de romper claves de este tamaño en un tiempo razonable por fuerza bruta.
>
> 3DES reusa el algoritimo usado por DES, pero concatena 3 rondas de DES con diferentes claves. Hasta el momento se considera seguro. Pero es bastante lento.

> AES es el sucedor de DES y admite varios tamaños de claves, todos ellos seguros (128, 192 y 258bits). Se considera el más seguro y es el estandar por defecto de encriptación en internet. Además, es más rapido que 3DES.

De las tres opciones, pero nos decantamos por AES ya que es el más moderno de los tres, más rapido y nos ofrece más flexibilidad para elegir el tamaño de clave.

Advanced Encryption Standard (AES)

AES es un algoritmo de cifrado simétrico de bloques, cuyo tamaño es siempre 128bits (16bytes) independientemente del tamaño de clave.

Cuando vemos AES-128, AES-192, AES-256, no se refiere al tamaño de bloque, sino al tamaño de clave. Para esta práctica vamos a seleccionar el tamaño máximo 256bits (32bytes), por tanto usaremos AES-256.

Esto no significa que AES-128 sea poco seguro, ni mucho menos, de hecho sigue siendo una de las opciones más usadas en las conexiones HTTPS. Esto se debe a que un tamaño de clave mayor, implica un coste en CPU y memoria superior.

En cambio, nuestro caso de uso es diferente, no necesitamos cifrar/descifrar en tiempo real, sino que vamos a cifrar un fichero una única vez y posteriormente lo descifraremos. Metáforicamente, nuestro programa es como un baúl que tiene que mantener seguro nuestro mensaje a lo largo del paso de tiempo, entender esto es muy importante.

La Ley de Moore expresa que aproximadamente cada 2 años se duplica la potencia de las CPUs, este crecimiento expoencial significa que aunque a día de hoy una clave de 128bits sea segura, quizás no lo sea dentro de 10años.

Por este motivo tiene sentido elegir AES-256, ya que nos aportará una seguridad extra a largo plazo y no repercutirá negativamente.

Modo de operación de bloques

Continuando con la definición de nuestro cifrador, es necesaria la eleccion de un modo de operación de bloques. Como bien hemos estudiado, conocemos varios:

- Modo ECB
- Modo CBC
- Modo CFB
- Modo OFB
- Modo CTR

Inmediatamente descartamos el más simple de todos, el modo ECB. Este modo aplica el bloque de AES de forma independiente. De modo que un mismo input, siempre arroja el mismo output.

ECB es por tanto vulnerable al criptoanálisis menos sofisticado. Patrones en una imagen o incluso frecuencias de caracteres se propagan y son percibibles en el texto cifrado.

CBC, es uno de los modos más utilizados y con propiedades de seguridad reconocidas. A diferencia de ECB, CBC propagaga la salida un bloque como entrada del siguiente, esta propiedad le permite que diferentes bloques con el mismo plaintext y misma clave, generen diferentes outputs.

Como la entrada de un bloque es funcion del texto plano anterior y output del bloque anterior, es imposible su paralelización y no se puede hacer cifrado de flujo con el.

CFB, OFB, y CTR funcionan de forma similar, en el sentido de que se pueden usar para generar un stream de bits que XOR-ean con el plaintext.

Dado que nuestro caso de uso consite en el cifrado de un único fichero, no necesitamos cifrádo en flujo ni la paralelización, por tanto usaremos el modo CBC.

Una vez hemos seleccionado AES-256 en modo CBC, nos disponemos a la implementación.

Vector de Inicialización (IV)

El modo CBC necesita de un vector de inicialización o IV como entrada del primer bloque. IV debe tener el mismo tamaño que el bloque usado, como sabemos para AES es siempre 16bytes (independientemente de la clave).

Uno de los errores más comunes es el de siempre usar el mismo IV. El IV cae dentro de los llamado números "nonce", es decir, que nunca deberiamos usar el mismo más de una vez.

Usar el mismo IV convierte nuestro AES en un One Time Pad, y perdería toda su seguridad cuando se usasé la misma clave para más de un mensaje.

```
P1 = texto plano 1
P2 = texto plano 2
K = clave
IV = mismo IV

E(P1, K, IV) xor E(P2, K, IV) == P1 xor P2
```

Como vemos, en caso de usarse el mismo IV y la misma clave para cifrar dos textos diferentes, permitiría un criptoanalisis muy sencillo.

El IV por tanto deberá ser un número aleatorio. Lenguajes como C incluyen rand() para esta tarea, PERO, NUNCA debemos obtener un valor aleatorio generado de esta forma para tareas criptográficas, NUNCA.

La aleatoriedad no se define como uniformidad, sino como la falta de patrón, es decir, dado una secuencia

albitrariamente larga de valores aleatorios, es imposible hacer predicciones sobre el siguiente dígito de la secuencia.

Las funcion rand() en C, es altamente predecible y por tanto es una fuente terrible de entropía. Para la creación de un sistema criptográficamente seguro deberemos usar un generador de números aleatorios criptográfico. Por suerte, todos los sistemas operativos nos ofrecen uno (en linux este es `/dev/random` y `/dev/urandom`), el sistema operativo usa técnicas mucho más sofisticadas para esta tarea. Recogiendo entropía de fuentes relativamente impredecibles:

- Movimiento de raton
- Ruido electrogmagnetico
- Tiempo de la CPU al nanosegundo

Finalmente usará funciones diseñadas por criptólogos para eliminar el sesgo y finalmente presentarla al usuario final a través de una interfaz (API).

Para el IV por tanto deberemos obtener 16bytes aleatorios:

```
function getRandom(size) {
  file = open("/dev/random")
  return file.read(size)
}

iv = getRandom(16)
```

Recordemos que el IV debe ser aleatorio, pero no tiene porque ser privado, es perfectamente seguro exponer el IV que hemos usado, de hecho, usar el mismo IV será necesario a la hora de descifrar. Esto supone que lo tendremos que incluir en el fichero cifrado (de salida) como metadatos.

```
|     iv      |   texto cifrado...  |
|<- 16bytes ->|<- len(cipherText) ->|
```

Clave AES

El siguente factor a tener en cuenta es la clave a usar en el algoritmo de bloques AES. Nuestra intuición nos diría que deberiamos usar la contraseña. Es decir:

```
K = contraseña
```

Y otra vez nuestra intución nos traiciona. Existen dos problemas con este enfoque:

- La clave usaba debe tener exactamente 256bits (32bytes) y es muy probable que nuestra contraseña

sea mucho más corta (o incluso más larga).

- Idealmente la clave no solo debe medir 256bits, sino tener una entropia máxima (de 1), es decir, 256bits de entropia. Hablando en cristiano, esto quiere decir, que la clave debe ser completamente aleatoria. Los humanos somos terribles generando contraseñas aleatorias.

Para solucionar este problema deberemos derivar la contraseña y obtener una contraseña.

Una funcion de derivación relativamente sencilla y segura es HMAC.

```
function HMAC(mensaje, clave) { ... }
function derivar(contrasena, nonce) {
  return HMAC(nonce, contrasena);
}
```

Nuestra funcion de derivación sera una HMAC-SHA256, que tome un nonce como mensaje y la contraseña como clave.

El diseño interno de HMAC nos asegura que la clave no se pueda recuperar, por tanto nuestra contraseña no puede ser deducida a partir de su derivada.

```
aesEntropy = getRandom(32)
AESKey = derivar(contraseña, aesEntropy)
```

Es evidentemente que una función por si sola no puede "generar" entropía, de modo muy similar a como obtuvimos el IV, necesitamos obtener 32bytes (256bits) de entropia, para poder generar una clave AES con la misma entropia. Dado que SHA256 siempre genera un salida de 32bytes, esta se podrá usar directamente como clave del cifrador.

Al igual que para el IV, `aesEntropy` debe ser aleatorio, pero no tiene porque ser privado.

Para poder desencriptar de hecho lo necesitaremos y es necesario incluirlo en el "header", de forma análoga al IV.

```
| aesEntropy  |     iv      |   texto cifrado...  |
|<- 32bytes ->|<- 16bytes ->|<- len(cipherText) ->|
```

Texto plano

Una vez tenemos el IV y la clave lista para ser usada por AES-CBC, todavía nos faltaría adaptar el texto plano adecuadamente.

Al usar un cifrado por bloques, el tamaño del texto plano deberá ser multiplo del tamaño de bloque, en nuestro caso 16bytes, recordemos que el tamaño de bloque es independiente del tamaño de clave.

Imaginemos que tenemos un texto plano M que mide 23bytes.

```
M = "hola valladolid"
Tamaño = len("hola y adios valladolid") // == 23bytes
```

Nuestro cifrador AES, tiene un bloque de 16bytes. Por tanto el texto plano "hola y adios valladolid" necesitará de al menos 2 bloques, 32bytes.

```
2*Bloque = 2*16 = 32
```

Pero nuestro mensaje es de solo 23 bytes, eso significa que tendremos que añadir `32-23` bytes de padding.

Pero esto nos plantea una nueva pregunta, ¿que contenido añado como padding? Podrían ser 0s, pero entonces al descifrar no sabriamos si esos 0s son parte del padding o del texto plano original.

Para solucionarlo podriamos añadir el tamaño del texto plano en los metadatos, pero entonces estariamos filtrando información del texto plano, NUNCA DEBEMOS HACER ESO.

La solución, es usar un algoritmo de padding recomendado para AES, uno de ellos es **PKCS7**.

PKCS7

Aunque el nombre pueda intimidar, el funcinamiento de este algoritmo de padding es muy sencillo.

Imaginatemos que nuestro tamaño de bloque es 16, y nuestro texto plano a encriptar es 23 bytes.

1. Calculamos el número de bytes del padding

```
TamañoMensaje = 23
Bloque = 16
PaddingLen = Bloque - (TamañoMensaje mod Bloque)
```

Para nuestro, ejemplo:

```
PaddingLen = 16 - (23 mod 16)
PaddingLen = 9
```

Nos damos cuenta de que si sumamos `PaddingLen + Tamaño` nos da un multiplo del tamaño de

bloque:

```
TamañoPlainText = 23 + 9 = 32
32 mod 16 = 0
```

Rellenamos con PaddingLen

Continuando con el caso anterior. Deberemos añadir 9 bytes, con valor "9".

Veamos un ejemplo:

```
M = "hola y adios valladolid"
P = PKCS7(M)
P == "hola y adios valladolid" + 9,9,9,9,9,9,9,9,9
```

Una implementación en Golang sería:

```go
func padPKCS7(msg []byte) []byte {
    padSize := 16 - len(msg)%16
    padding := make([]byte, padSize)
    for i := 0; i < padSize; i++ {
        padding[i] = byte(padSize)
    }
    return append(msg, padding...)
}
```

En ocasiones todas estas consideraciones las realiza la libreria elegida, por ejemplo, en Java:

```java
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS7Padding");
```

Nos retornaria nuestro cifrador: AES, en modo CBC y con padding PKCS7,¡no tendremos que implementarlo nosotros!

Recopilacion del cifrador

Pongamos todo junto lo visto hasta ahora:

```
// Obtenemos entropia para generar el IV
aesIV := getRandom128()

// Obtenemos entropia y derivamos la contraseña para generar una clave AES
aesEntropy := getRandom256()
aesClave := HMAC_SHA256(aesEntropy, contraseña)

// Añadimos un padding PKCS7 a nuestro texto plano original para que el tamaño
// sea multiplo del tamaño de bloque usado
textoPlano = padPKCS7(textoPlano)

// Ciframos el texto plano usado AES256-CBC con la clave y el IV
textoCifrado := encryptAES256_CBC(textoPlano, aesClave, aesIV)

// Generamos el mensaje, concatenando los 32bytes de la entropia AES
// el IV y el texto cifrado
cabecera := aesEntropy + aesIV
mensaje := cabecera + textoCifrado
```

Podriamos pensar que ya hemos terminado, y el mensaje ya es perfectamente seguro, pero no es así.

Integridad

AES en modo CBC no aporta integridad, esto significa que si alguien modifica el texto cifrado, al descifrar no podremos detectar que una modificación sucedio. El texto descifrado evidentemente no será igual al texto plano original, PERO el receptor NO PODRÁ saberlo/detectarlo.

Para añadir integridad y con ello completar nuestro programa debemos firmar el mensaje, para ello deberemos determinar primero que "mensaje" vamos a firmar, que técnica de firma usaremos y la clave empleada.

Algoritmo de firma

Para el algoritmo de firma usaremos HMAC, teniendo en cuenta que solo usaremos una contraseña, no tiene sentido usar un arquitectura de clave asimétrica.

Al igual que nunca debemos diseñar nuestros propios algoritmos de cifrado, tampoco deberemos diseñar o implementar nuestro propio sistema de firma.

El algortimo HMAC siempre va acompañado de una función hash, la cual usaremos SHA-256.

> SHA es una familia de algoritmos de hash recomendados por criptólogos y organismos. SHA-1 se considera roto a diria de hoy, por lo que usaremos SHA-2 con un tamaño de 256bits.

```
Firma(M) = HMAC(M, K)
```

Clave de firma

La clave será una vez más nuestra contraseña derivada, pero con 256bits de entropia diferente. No es recomendable usar la misma clave AES usada para en el cifrado, para la firma, hay que usar dos diferentes, la razón para esto, es reducir las oportunidades especulativas de una criptoanálisis.

Mensaje a firmar

El mensaje podría ser solo el texto cifrado, pero eso significaría que los metadatos podrian modificarse (todos bits correspondientes a la entropia y el IV).

El lógico por tanto que el mensaje M a firmar, debe ser la concatenación del texto cifrado más todos los metadatos.

El mensaje "M" a firmar por tanto será el siguiente:

```
| hmacEntropy | aesEntropy  |     iv      |   texto cifrado...  |
|<- 32bytes ->|<- 32bytes ->|<- 16bytes ->|<- len(cipherText) ->|
```

Poniendolo todo en ensamblado en contexto:

```
M = hmacEntropy + aesEntropy + iv + texto_cifrado
S = firmar(M)
Final = S + M
```

Los datos en "Final" son los que finalmente escribiremos en el fichero de sálida.

Salida "Final" será la firma más todo el mensaje de la siguiente manera

```
|    firma    | hmacEntropy | aesEntropy  |     iv      |   texto cifrado...  |
|<- 32bytes ->|<- 32bytes ->|<- 32bytes ->|<- 16bytes ->|<- len(cipherText) ->|
```

Código fuente

```
func Encrypt(plaintext []byte, password string) []byte {
    // Collect random data for the IV (initial vector=), and for the key deriva
    // of the AES key and the HMAC key
    aesIV := getRandom128()
```

```
    aesEntropy := getRandom256()
    hmacEntropy := getRandom256()

    // Derive the password using the entropy collected
    aesKey := deriveKey(password, aesEntropy)
    signatureKey := deriveKey(password, hmacEntropy)

    // Plaintext size must be a multiple of the block size of AES
    plaintext = padPKCS7(plaintext)

    // Generate the ciphertext using fron the plaintext, the generated aesKey a
    cipherText := encryptAESCBC(plaintext, aesKey, aesIV)

    // Build the mensage to be signed using the HMAC
    // The message is all the entropy + ciphertext
    message := buildMessage(aesEntropy, hmacEntropy, aesIV, cipherText)

    // Generate signature using HMAC
    signature := hmacSHA256(message, signatureKey)

    // Build final package
    output := buildFinal(signature, message)

    return output
}

func deriveKey(masterKey string, random []byte) []byte {
    return hmacSHA256(random, []byte(masterKey))
}

func buildMessage(aesEntropy, hmacEntropy, aesIV, cipherText []byte) []byte {
    ////////////////////////////////////////////////////////////////////|
    // |  randomAES  |  randomHMAC |    aesIV    |  cipherText ......... |
    // |  32 bytes   |  32 bytes   |  16 bytes   |  rest of bytes        |
    ////////////////////////////////////////////////////////////////////|
    var buffer bytes.Buffer
    buffer.Write(aesEntropy)
    buffer.Write(hmacEntropy)
    buffer.Write(aesIV)
    buffer.Write(cipherText)
    return buffer.Bytes()
}

func buildFinal(signature, message []byte) []byte {
    ////////////////////////////////////////////////////////////////////////////
    // |  signature  ||  randomAES  |  randomHMAC |    aesIV    |  cipherText .
    // |  32 bytes   ||  32 bytes   |  32 bytes   |  32 bytes   |  rest of byte
    ////////////////////////////////////////////////////////////////////////////
```

```
53      var buffer bytes.Buffer
54      buffer.Write(signature)
55      buffer.Write(message)
56      return buffer.Bytes()
57  }
```

El codigo fuente completo implementado en Golang (un lenguaje similar a C y facil de entender) se puede encontrar en la raiz de esta carpeta.