UVG
Cifrados
Adrian Rodríguez 21691

Ejercicio Block Cipher

Nota: Los fragmentos de código proporcionados están escritos en Go (Golang), el lenguaje utilizado para la implementación de este laboratorio. En particular, la función `make()` en Go se utiliza para la asignación dinámica de memoria.

Tamaños de Clave:
Las implementaciones cubren tres algoritmos de cifrado por bloques comunes: DES, 3DES y AES. Cada uno utiliza un tamaño de clave diferente:

DES (Data Encryption Standard): Implementado con una clave de 56 bits (representada como 8 bytes en el código, con un bit de paridad por byte).

```
key := make([]byte, 8)
```

3DES (Triple DES): Utiliza una clave efectiva de 168 bits (24 bytes en la implementación).

```
key := make([]byte, 24)
```

AES (Advanced Encryption Standard): Implementado con una clave de 256 bits (32 bytes).

```
key := make([]byte, 32)
```

Modos de Operación:

Se implementaron los siguientes modos de operación:

- DES: Modo ECB (Electronic Codebook).
- 3DES: Modo CBC (Cipher Block Chaining).
- AES: Modos ECB y CBC.

Importancia de la Selección del Modo de Operación:

El modo de operación es crucial para la seguridad. El modo ECB es vulnerable y no debe usarse para datos sensibles debido a que:

No oculta patrones en los datos: bloques de texto plano idénticos se cifran en bloques de texto cifrado idénticos.

Es susceptible a ataques de análisis estadístico.

No hay difusión entre bloques, lo que significa que la modificación de un bloque no afecta a los demás.

La diferencia entre ECB y CBC es visualmente evidente al cifrar imágenes. Con ECB, los patrones de la imagen original permanecen visibles en la versión cifrada. En cambio, CBC produce una imagen cifrada que se asemeja a ruido aleatorio, ocultando completamente los patrones originales. Esta demostración ilustra claramente la inseguridad de ECB para datos con patrones predecibles.

Vector de Inicialización (IV):

El IV es un valor aleatorio utilizado en conjunto con la clave para cifrar el primer bloque de datos. Es esencial para modos como CBC, donde se combina con el texto plano inicial mediante una operación XOR. El IV no necesita ser secreto, pero debe ser único para cada mensaje cifrado con la misma clave para prevenir la repetición de patrones en el texto cifrado. En esta implementación, se genera un IV aleatorio para los modos CBC:

```
import (
    "crypto/rand"
)

iv := make([]byte, 16)
_, err := rand.Read(iv)
if err != nil {
    panic(err)
}
```

Padding (Relleno):

Se utiliza el padding para completar el último bloque. En esta implementación, se utiliza el esquema *PKCS#7*, donde se añaden bytes al final del mensaje, y el valor de cada byte añadido corresponde al número de bytes añadidos.

Un ejemplo de implementación de PKCS#7 padding en Go:

```
func pkcs7Pad(data []byte, blockSize int) []byte {
    paddingSize := blockSize - (len(data) % blockSize)
    padding := make([]byte, paddingSize)
    for i := range padding {
        padding[i] = byte(paddingSize)
    }
    return append(data, padding...)
}
```

Recomendaciones de Uso:

ECB: Se desaconseja su uso en la mayoría de los escenarios. Puede ser útil para pruebas, demostraciones educativas o para cifrar datos muy pequeños y aleatorios.

CBC: Es adecuado para el cifrado general de datos donde la seguridad es importante y se necesita ocultar patrones.

Otros Modos de Operación (No Implementados):

- CTR (Counter): Ideal para cifrado de flujo y grandes volúmenes de datos debido a su paralelización.
- GCM (Galois/Counter Mode): Proporciona cifrado autenticado, garantizando tanto la confidencialidad como la integridad de los datos.
- XTS: Específico para el cifrado de dispositivos de almacenamiento.

Selección de Modos Seguros por Lenguaje de Programación:

La elección del modo seguro varía según el lenguaje de programación. En general, se recomienda utilizar modos autenticados como GCM o ChaCha20-Poly1305 para aplicaciones modernas, ya que ofrecen tanto confidencialidad como integridad de los datos. A continuación, se presentan ejemplos en diferentes lenguajes:

Python:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
import os

def encrypt_aes_gcm(key, plaintext):
    cipher = AES.new(key, AES.MODE_GCM)
        ciphertext,  tag  =  cipher.encrypt_and_digest(pad(plaintext,
AES.block_size))
    return ciphertext, tag
```

Java

```
import javax.crypto.Cipher;
import javax.crypto.spec.GCMParameterSpec;
import java.util.Arrays;


public  byte[]  encryptAESGCM(byte[]  key,  byte[]  plaintext)  throws
Exception {
    Cipher cipher = Cipher.getInstance("AES/GCM");
```

```
        GCMParameterSpec gcmSpec = new GCMParameterSpec(128, new
byte[12]); // 128-bit tag
    cipher.init(Cipher.ENCRYPT_MODE, key, gcmSpec);
    return cipher.doFinal(plaintext);
}
```

JavaScript (Node.js):

```
const crypto = require('crypto');

function encryptAESGCM(key, plaintext) {
  const iv = crypto.randomBytes(12);
  const cipher = crypto.createCipheriv('aes-256-gcm', key, iv);
  let encrypted = cipher.update(plaintext);
  encrypted = Buffer.concat([encrypted, cipher.final()]);
  return {
    ciphertext: encrypted,
    iv: iv,
    authTag: cipher.getAuthTag()
  };
}
```

Recomendación general: Para aplicaciones modernas, se recomienda usar modos autenticados como GCM o ChaCha20-Poly1305, que proporcionan tanto confidencialidad como integridad de los datos.

Reflexión sobre las Limitaciones de los Generadores Pseudoaleatorios Simples
Los generadores pseudoaleatorios simples representan una vulnerabilidad en los sistemas criptográficos modernos debido a tres factores: su predictibilidad (dado que producen secuencias deterministas), su periodicidad limitada (eventualmente repiten valores) y sus posibles sesgos estadísticos (distribuciones no uniformes).

Estas debilidades han provocado fallos de seguridad históricos como el caso de Netscape SSL en 1996, donde la semilla del generador se basaba en datos fácilmente predecibles, o el incidente del monedero Bitcoin de Android en 2013, donde una implementación deficiente del PRNG permitió la generación de claves privadas vulnerables.

Para garantizar la seguridad en implementaciones criptográficas reales, es esencial utilizar generadores de números aleatorios criptográficamente seguros (CSPRNG) que aprovechen fuentes de entropía del sistema operativo y superen pruebas estadísticas rigurosas. La generación de claves, vectores de inicialización y nonces debe realizarse exclusivamente con estos generadores robustos, como crypto/rand en Go, secrets en Python o SecureRandom en Java, en lugar de sus contrapartes simples.