Pontificia Universidad Católica de Chile
Departamento de Ciencia de la Computación
IIC3253 – Criptografía y Seguridad Computacional
Primer Semestre del 2025

Ayudantía 7

HMAC

Ayudante: Cristóbal Rojas – cristobalrojas@uc.cl

Resumen

Construcción Davies-Meyer

Partimos de un esquema criptográfico

$$(Gen, Enc, Dec) \quad \text{sobre} \quad M = K = C = \{0,1\}^*$$

Para un parámetro de seguridad n, definimos

$$Gen'(1^n) = n$$

y la función de compresión de bloque fijo

$$h' : \{0,1\}^{2n} \to \{0,1\}^n \quad \text{por} \quad h'(u\|v) = \text{Enc}_v(u) \oplus u$$

En esta construcción:

- El primer bloque $u \in \{0,1\}^n$ es el texto claro a cifrar.
- El segundo bloque $v \in \{0,1\}^n$ se usa como clave de cifrado.
- Se cifra u con la clave v y luego se aplica XOR con u mismo.

La construcción Davies-Meyer tiene propiedades importantes para la seguridad:

- Proporciona preimagenes fijas: Dado un valor de salida, es computacionalmente difícil encontrar las entradas que lo produjeron, incluso si el cifrado por bloques subyacente se comporta como una permutación aleatoria.
- La operación XOR final introduce una asimetría que previene ataques meet-in-the-middle.
- Si el cifrado por bloques subyacente no tiene vulnerabilidades conocidas, encontrar colisiones requiere aproximadamente $2^{n/2}$ operaciones, donde n es el tamaño de salida en bits.

Construcción Merkle-Damgård

Merkle-Damgård extiende una función de compresión segura a entradas de longitud variable. La construcción toma una función de compresión h' y la utiliza para crear una función hash h que puede procesar mensajes de cualquier longitud.

Dado un mensaje M, se procede así:

1. **Padding**: Al mensaje M se le añade un bit '1', ceros y la longitud de M en bits, para que su tamaño sea múltiplo del bloque:

$$M' = \text{Pad}(M) = M \parallel \text{`1'} \parallel \text{`0'}^k \parallel |M|$$

   Donde:

   - '1' es un bit '1' individual
   - '0'^{k} representa k bits '0' (tantos como sean necesarios)
   - $|M|$ es la representación binaria de la longitud original del mensaje

2. **Particionamiento**: Dividir M' en bloques de tamaño adecuado: $M' = m_1 \| m_2 \| \cdots \| m_l$

3. **Iteración**: Se inicializa con un valor fijo H_0 y se procesa cada bloque i usando la función de compresión:

$$H_i = h'(m_i \| H_{i-1}) \quad \text{para } i = 1, 2, \ldots, l$$

4. **Salida**: El digest final es $h^s(M) := H_l$

Importancia del padding: La inclusión de $\text{len}(m)$ en el padding es crucial para la seguridad de la función hash. Sin este elemento, la construcción sería vulnerable a ataques de extensión, donde para un mensaje m con hash $h(m)$, un atacante podría calcular fácilmente el hash de $m \| p_m \| x$ (para cualquier extensión x) sin conocer m completo.

HMAC

Sea h una función de compresión que puede ser usada para construir una función hash mediante la transformación *Merkle-Damgård*. Definimos el *MAC* basado en hash (*HMAC*) de la siguiente forma:

- ***Gen:*** con entrada 1^n, genera una clave k de longitud apropiada.
- ***Mac:*** con entrada la clave k y un mensaje $m \in \{0,1\}^*$, calcula:

$$k' = \begin{cases} h(k) & \textit{si } k \text{ usa más de un bloque} \\ k & \textit{en otro caso} \end{cases}$$
$$k_1 = k' \oplus 5\text{c} \ \cdots \ 5\text{c} \quad (\text{donde } 5\text{c} = 01011100 \text{ en binario})$$
$$k_2 = k' \oplus 36 \ \cdots \ 36 \quad (\text{donde } 36 = 00110110 \text{ en binario})$$

Y finalmente:

$$HMAC(k, m) = h\left(k_1 \| h\left(k_2 \| m\right)\right)$$

- ***Verify:*** con entrada una clave k, un mensaje $m \in \{0,1\}^*$ y una etiqueta t, devuelve 1 *si y solo si* $t \stackrel{?}{=} HMAC(k, m)$.

En esta construcción, k_1 y k_2:

- Ocupan exactamente un bloque cada uno.
- Se derivan de forma determinista a partir de k.
- Son distintos entre sí.
- No se pueden obtener sin conocer k.

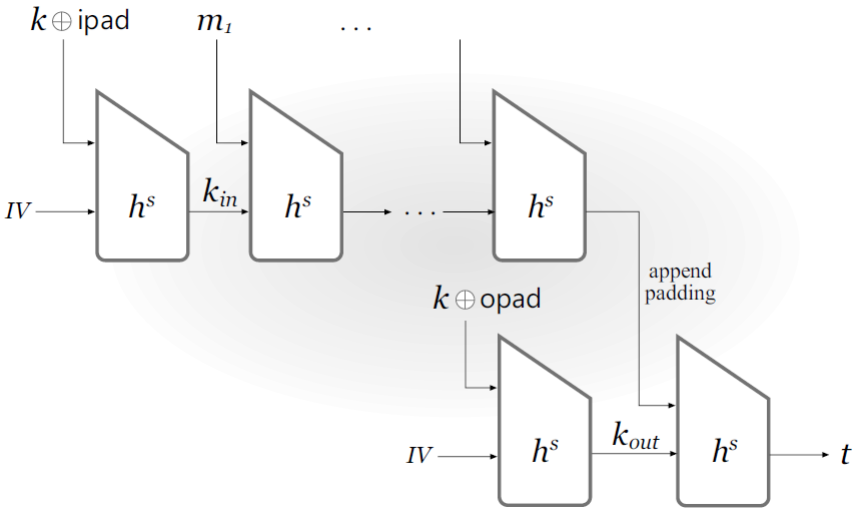

Figura 1: Estructura de HMAC

Algunos textos usan la siguiente notación:

- *IV* significa *Valor de Inicialización* (*Initial Value*). Es un valor constante predefinido utilizado al iniciar el proceso de hashing en la estructura Merkle-Damgård.
- *ipad* es la constante *inner padding* representada como $36 \cdots 36$ (en hexadecimal), o $00110110 \cdots 00110110$ en binario. En la notación del curso corresponde a k_2.
- *opad* es la constante *outer padding* representada como $5c \cdots 5c$ (en hexadecimal), o $01011100 \cdots 01011100$ en binario. En la notación del curso corresponde a k_1.

Estas constantes *ipad* y *opad* tienen valores diferentes para garantizar que k_1 y k_2 sean distintos entre sí. Ambas constantes tienen la longitud exacta de un bloque, y al combinarlas con k' mediante la operación XOR ($\oplus$), se generan las claves k_1 y k_2 que ocupan exactamente un bloque cada una.

Problema 1

¿Por qué en la construcción de Merkle-Damgård $\text{Pad}(m)$ se incluye el largo del mensaje? ¿Qué propiedad necesaria se rompería en caso contrario? De un ejemplo concreto para ilustrar qué pasa en el caso de que no se incluya el largo del mensaje.

Solución

Si $|m|$ no es el múltiplo del largo del bloque, entonces:

$$\text{Pad}(m) = m \| p_m$$

Ahora, si hasheamos directamente un m' tal que

$$m' = m \| p_m = \text{Pad}(m)$$

Entonces encontramos una colisión, ya que m' y m tienen el mismo Hash.

Ejemplo concreto:

Consideremos un escenario simple con un tamaño de bloque de 8 bits y un mensaje original: m_1 = "A" (que en ASCII es 01000001)

Supongamos que usamos un padding simple sin incluir la longitud:

$$\text{Pad}(m_1) = 01000001 \| 10000000 = 0100000110000000$$

Donde añadimos el bit '1' seguido de ceros para completar un múltiplo del tamaño de bloque.

Ahora, un atacante podría crear un mensaje malicioso $m_3 = m_1 \| 10000000$, es decir, "A" seguido del byte de padding que usamos para m_1.

El problema es que sin incluir la longitud original en el padding:

$$h(m_1) = h(\text{Pad}(m_1)) = h(01000001 \| 10000000)$$

Y para m_3:

$$h(m_3) = h(01000001 \| 10000000) = h(\text{Pad}(m_1))$$

Por lo tanto, $h(m_3) = h(m_1)$, ¡lo que es una colisión!

En cambio, con el padding adecuado que incluye la longitud:

- Para m_1: $\text{Pad}(m_1) = 01000001 \| 10000000 \| 00001000$ (donde 00001000 es la representación binaria de 8, la longitud original en bits)
- Para m_3: $\text{Pad}(m_3) = 01000001 \| 10000000 \| 10000000 \| 00010000$ (donde 00010000 es la representación binaria de 16, la longitud original en bits)

Ahora $h(m_1) \neq h(m_3)$ porque sus representaciones con padding son diferentes, evitando la colisión y preservando la seguridad de la función hash.

Problema 2

JSON Web Token (JWT) es un estándar abierto (RFC 7519) que define una forma compacta y autónoma para transmitir información entre partes de forma segura. En la práctica, JWT se utiliza frecuentemente como mecanismo de autenticación y autorización en aplicaciones web y APIs. Un JWT se compone de tres partes separadas por puntos:

Header.Payload.Signature

Estructura

- **Header**: Contiene el tipo de token y el algoritmo de firma utilizado.

```
Header = {"alg": "HS256", "typ": "JWT"}
```

- **Payload**: Contiene los "claims" o afirmaciones sobre una entidad y datos adicionales.

```
Payload = {
    "sub": "1234567890",
    "name": "Usuario Ejemplo",
    "admin": true
}
```

- **Signature**: Se calcula utilizando HMAC sobre el header y payload codificados en base64.

La firma se calcula de la siguiente manera:

```
Signature = HMAC-SHA256(base64UrlEncode(Header)
            + '.'
            + base64UrlEncode(Payload), secret)
```

Un ejemplo de JWT completo sería:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IlVzdWFyaW8gRWplbXBsbyIsImFkbWluIjp0cnVlfQ.
rDZ6zA5T823vxlFBwXXDhxPE-GSYxNYnMl5jP1PL1JQ
```

El receptor del token puede verificar su autenticidad recalculando la firma con la clave secreta compartida.

En base a lo anterior, responde si las siguientes preguntas son verdaderas o falsas, justificando tu decisión:

1. **Afirmación:** Si la clave secreta utilizada en HMAC para JWT es revelada, la seguridad del sistema permanece intacta.

2. **Afirmación:** Un atacante que intercepta un JWT válido puede modificar el payload para cambiar la identidad del usuario o sus privilegios sin invalidar la firma.

3. **Afirmación:** HMAC proporciona las mismas garantías de seguridad que un simple hash del mensaje cuando se usa en JWT.

4. **Afirmación:** Utilizar $\text{MAC}_k(M) = h(k\|M)$ en lugar de HMAC para generar la firma del JWT es seguro contra ataques de extensión.

Solución

1. **Falso.** La clave secreta es fundamental para la seguridad de JWT porque permite calcular y verificar la firma. Si la clave secreta se revela, un atacante podría crear tokens fraudulentos con cualquier contenido y firmarlos correctamente, suplantando así cualquier identidad o privilegio.

2. **Falso.** Un atacante no puede modificar el payload sin invalidar la firma. Cualquier cambio en el header o payload resultaría en una firma diferente al recalcular el HMAC. Como el atacante no posee la clave secreta, no puede generar una firma válida para un payload modificado.

3. **Falso.** HMAC proporciona tanto integridad como autenticación, mientras que un simple hash solo garantiza integridad. Además, HMAC es resistente a los ataques de extensión de longitud y otros ataques que afectan a esquemas como $h(k\|M)$ o $h(M\|k)$.

4. **Falso.** Utilizar $\text{MAC}_k(M) = h(k\|M)$ es vulnerable a ataques de extensión. Un atacante que tiene un token válido podría extender el payload añadiendo información adicional sin conocer la clave, aprovechando la estructura iterativa de las funciones hash basadas en Merkle-Damgård.

Problema 3

Escribe un programa (en Python, o el lenguaje que desees) para calcular HMAC-SHA-384 de un mensaje de texto utilizando una clave dada. Puedes escribir tu código en el lenguaje de programación de tu elección.

1. Primer caso:

   - **Entrada:**
     - **Mensaje:** hello
     - **Clave:** cryptography
   - **Salida:** 83d1c3d3774d8a32b8ea0460330c16d1b2e3e5c0ea86 ccc2d70e603aa8c8151d675dfe339d83f3f495fab226795789d4

2. Segundo caso:

   - **Entrada:**
     - **Mensaje:** hello
     - **Clave:** again
   - **Salida:** 4c549a549aa037e0fb651569bf271faa23cfa20e8a9d2 1438a6ff5bf6be916bebdbaa48001e0cd6941ec74cd02be70e5

Solución

A continuación se presenta una solución en Python para calcular HMAC-SHA-384:

```
import hashlib

def hmac_sha384(clave, mensaje):
    # Convertir mensaje y clave a bytes si son strings
    if isinstance(mensaje, str):
        mensaje = mensaje.encode('utf-8')
    if isinstance(clave, str):
        clave = clave.encode('utf-8')

    # Tamaño de bloque para SHA-384 (en bytes)
    block_size = 128

    # Si la clave es más larga que el tamaño del bloque, aplicar hash
    if len(clave) > block_size:
        clave = hashlib.sha384(clave).digest()

    # Si la clave es más corta que el tamaño del bloque, rellenar con ceros
    if len(clave) < block_size:
        clave = clave + b'\x00' * (block_size - len(clave))

    # Constantes para el padding
    ipad = bytes([0x36] * block_size)
    opad = bytes([0x5c] * block_size)

    # Combinar la clave con ipad y opad mediante XOR
    key_ipad = bytes(x ^ y for x, y in zip(clave, ipad))
    key_opad = bytes(x ^ y for x, y in zip(clave, opad))

    # Paso 1: hash(key_ipad + mensaje)
    inner_hash = hashlib.sha384(key_ipad + mensaje).digest()

    # Paso 2: hash(key_opad + inner_hash)
    outer_hash = hashlib.sha384(key_opad + inner_hash).hexdigest()

    return outer_hash
```

El algoritmo HMAC funciona de la siguiente manera:

1. Se preparan las claves: Si la clave es más larga que el tamaño del bloque de la función hash, se aplica hash a la clave. Si es más corta, se rellena con ceros.
2. Se combina la clave con dos constantes (ipad y opad) usando XOR.
3. Se concatena el resultado del paso anterior con el mensaje y se le aplica la función hash.
4. Se concatena la clave combinada con opad y el resultado del paso 3, y se le aplica hash nuevamente.

Este proceso garantiza que el HMAC sea seguro incluso si la función hash subyacente tiene ciertas vulnerabilidades.

Problema 4 (Propuesto)

a) Demuestre que definir $MAC_k(M) = h(k\|M)$ resulta en un MAC inseguro. Es decir, demuestre que dado un par texto/MAC válido (M, H), se puede construir eficientemente otro par texto/MAC válido (M', H') sin conocer la clave k. Asuma por simplicidad que la longitud de la clave es igual a la longitud del bloque del mensaje $|k| = |M_i|$.

Pista: La función de compresión h' es, por supuesto, conocida, es decir, dadas las dos entradas cualquiera puede calcular su salida.

b) Demuestre que añadir la clave secreta k al final, es decir, definir $MAC_k(M) = h(M\|k)$ resulta en un MAC inseguro. Recuerde que, por definición, la propiedad de ser resistente a colisiones para un MAC significa que encontrar dos mensajes $x \neq x'$ tales que:

$$MAC(x, k) = MAC(x', k),$$

implica el esfuerzo computacional de 2^n operaciones, donde n es el tamaño del MAC (y del hash). Describa un ataque (basado en la estructura Merkle-Damgård anterior) que utilice menos de 2^n operaciones para crear una falsificación MAC, un valor MAC legítimo para algún mensaje x' sin revelar la clave k.

Solución

a) Un adversario crea el mensaje $M' = x\|y$ por lo que necesita calcular $MAC_k(k\|x\|y)$. Pero la estructura iterativa del esquema Merkle-Damgård implica que:

$$MAC_k(k\|x\|y) = h'\left(MAC_k(k\|x), y\right)$$

Por lo tanto, dado el valor $MAC_k(k\|x)$, el adversario simplemente evalúa la función de compresión públicamente conocida.

b) Aquí el procesamiento del mensaje no depende de la clave hasta el último bloque, por lo que podemos utilizar un *birthday attack*. Es decir, creamos $2^{n/2}$ mensajes diferentes:

$$M^1, \ldots, M^{2^{n/2}}$$

de la misma longitud (digamos que todos los mensajes tienen t bloques) y calculamos los valores hash $h(M^i)$. Entonces, la paradoja del cumpleaños implica que hay $M^i \neq M^j$ tales que $h(M^i) = h(M^j)$ para algún $1 \leq i < j \leq 2^{n/2}$. Entonces:

$$MAC_k(M^i\|k) = h'(h(M^i), k) = h'(h(M^j), k) = MAC_k(M^j\|k)$$

El adversario puede simplemente obtener el valor MAC en el mensaje M^i, y luego producir un par texto-MAC correcto $(M^j, MAC_k(M^j\|k))$.