

Resumen Criptografía y Seguridad Computacional

INDICE

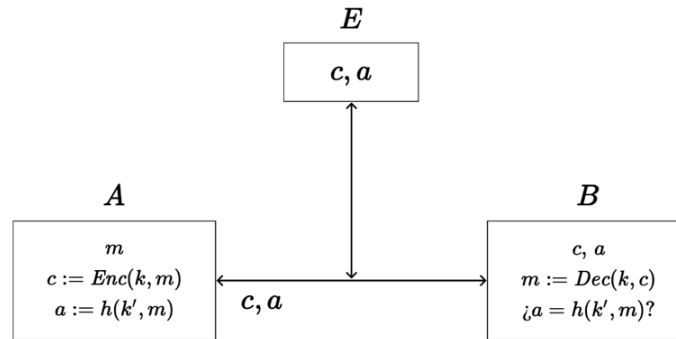
Cifrado simétrico y asimétrico.....	2
Firma digital con llave pública.....	3
Tipos de ataque.....	3
Cifrado del César – Shift → Permutación – OTP.....	4
Esquema criptográfico.....	5
<i>Perfect Secrecy</i>	6
PRP.....	7
Funciones de Hash.....	8
Función despreciable.....	8
Resistencia a colisiones.....	8
MACs.....	9
AES.....	9
ECB y CBC.....	11
Construcción de una función de hash.....	12
Función de compresión.....	12
<i>Davies-Meyer</i>	12
<i>Padding</i>	13
<i>Merkle Damgård</i>	14
HMAC.....	14
KDF (<i>Key Derivation Functions</i>).....	15
Aritmética modular.....	16
RSA.....	18
Test de primalidad.....	20
<i>Diffie-Hellman</i> y ElGamal.....	22
Firmas digitales.....	23
Firmas de <i>Schorr</i>	24
Seguridad Web.....	25

Cifrado simétrico

- A y B se tienen que poner de acuerdo en una clave k .
- Enc es la función de cifrado o encriptación.
- Dec es la función de descifrado o descryptación.
- Propiedad fundamental de estas funciones:

$$Dec(k, Enc(k, m)) = m$$

- A y B se tienen que poner de acuerdo en la clave k' para **autenticar**.
- $a := h(k', m)$ es llamado *Message Authentication Code (MAC)*, y usualmente es calculada usando una función de hash criptográfica.

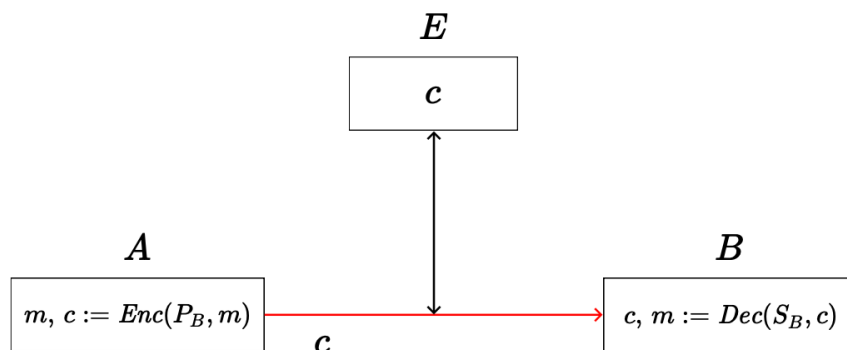


¿Problemas?

- El número de claves que un usuario debe almacenar es proporcional al número de sus contactos.
- Dos usuarios **deben reunirse** para compartir una clave.

Cifrado asimétrico

- Cada usuario A debe crear una llave pública P_A y una clave secreta S_A .
- P_A y S_A están relacionadas: P_A se usa para cifrar y S_A para descifrar.
- P_A es compartida con todos los otros usuarios.

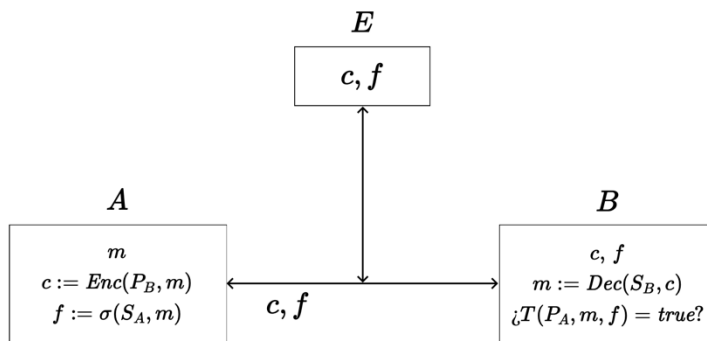


- Enc y Dec son las funciones de cifrado y descifrado.
- Propiedad fundamental:

$$Dec(S_B, Enc(P_B, m)) = m$$

Firma digital con una llave pública

- A está firmando un mensaje m , para cualquiera que lo necesite.
- $\sigma(S_A, m)$ utiliza la llave secreta de A para generar una firma f de m , de manera tal que solo A puede firmar.
- $T(P_A, m, f)$ verifica si f es una firma válida del mensaje m por el usuario A .
- $T(P_A, m, f)$ utiliza la clave pública de A , de manera que cualquiera puede verificar si f es una firma válida.



En este caso el usuario A está firmando el mensaje m para el usuario B

Tipos de ataques

- **Sólo texto cifrado:** El adversario conoce textos cifrados, sin saber que significan, simplemente escuchando por la red.
- **Texto plano conocido:** El adversario conoce textos planos junto a los mensajes no cifrados correspondientes (No tiene manera de elegir que textos específicos cifrar).
- **Texto plano elegido:** El adversario elige ciertos textos planos y obtiene los cifrados correspondientes. El adversario envía mensajes sabiendo que A los va a mandar cifrados a B , por ejemplo, en una guerra un bando envía mensajes que sabe van a ser interceptados y comunicados por el otro bando.
- **Texto cifrado elegido:** El adversario elige ciertos textos planos y textos cifrados, obteniendo el cifrado de los textos planos y los mensajes originales de los cifrados, es decir, puede tanto encriptar como desencriptar los mensajes interceptados.

Cifrado del César

Cada letra en realidad significa otra:

A B C D E F G H I J K L M N Ñ O P Q R S T U V W X Y Z		HOLA MUNDO
X Y Z A B C D E F G H I J K L M N Ñ O P Q R S T U V W		EMIX JRKAM

Problema: No hay llave. Fácil de descifrar mensajes.

Cifrado del César + llave

Llave = shift **7**. En vez de un shift predeterminado como se tenía arriba, este varía según la clave escogida.

Problema: La probabilidad de adivinar la llave correcta es $\frac{1}{|\Sigma|}$ con Σ el alfabeto elegido.
Puedo probar fácilmente $|\Sigma|$ combinaciones hasta obtener el mensaje original.

Shift → Permutación

Generar una permutación del alfabeto, mapeando una letra a otra en una función biyectiva.
Existen $|\Sigma|$ llaves posibles.

Problema: De tratarse de un lenguaje conocido, es posible conocer partes de la permutación al comparar la frecuencia de letras en el mensaje cifrado con la frecuencia de letras del idioma del mensaje.




Lo anterior nos deja con la importante noción que si bien un espacio de llaves grande es una condición necesaria para un buen cifrado, no es suficiente.

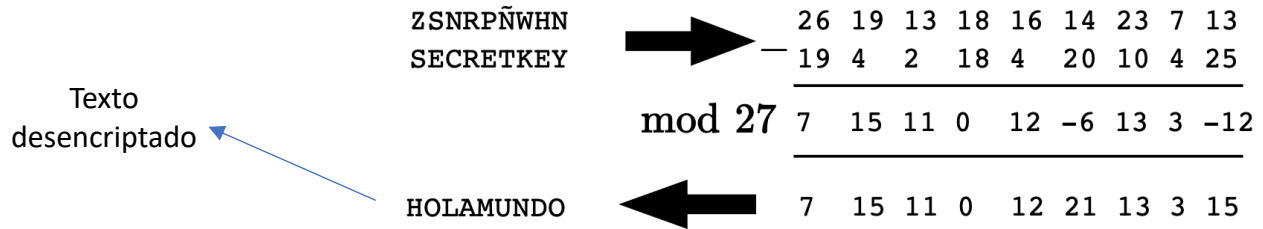
ONE-TIME PAD (OTP)

Partimos numerando el alfabeto a utilizar:

A B C D E F G H I J K L M N Ñ O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26

Para enviar un mensaje de largo l , necesitaremos una llave de largo l .

Texto encriptado		HOLAMUNDO		7 15 11 0 12 21 13 3 15
		SECRETKEY		+ 19 4 2 18 4 20 10 4 25
				mod 27 26 19 13 18 16 41 23 7 40
				
		ZSNRPÑWHN		4



Den general, tenemos que:

$$Enc(k, m) = (m + k) \bmod N = c$$

$$Dec(k, c) = (c - k) \bmod N = m$$

Esquema criptográfico

Definiremos un esquema criptográfico como la tripleta:

$$(Gen, Enc, Dec)$$

Definidos sobre los espacios:

$$\mathcal{K}, \mathcal{M}, \mathcal{C}$$

- \mathcal{K} corresponde al espacio de llaves, usualmente de forma Σ^* .
- \mathcal{M} corresponde al espacio de mensajes, usualmente de forma Σ^* .
- \mathcal{C} corresponde al espacio de texto cifrado, usualmente de forma Σ^* .
- Gen es una **distribución de probabilidades sobre \mathcal{K}** , es responsable de generar llaves criptográficas
- $Gen : \mathcal{K} \rightarrow [0, 1]$ tal que $\sum_{k \in \mathcal{K}} Gen(k) = 1$.
- $Enc : \mathcal{K} \times \mathcal{M} \rightarrow \mathcal{C}$ es el algoritmo de **encriptación**.
- $Dec : \mathcal{K} \times \mathcal{C} \rightarrow \mathcal{M}$ es el algoritmo de **desencriptación**.

Donde esperamos que se cumpla:

$$Dec(k, Enc(k, m)) = m \quad \forall k \in \mathcal{K}, \forall m \in \mathcal{M}$$

Ejemplo, en **OTP**:

$$\mathcal{K} = \mathcal{M} = \mathcal{C} = \{0, \dots, N - 1\}^l$$

Gen es la distribución uniforme sobre $\{0, \dots, N - 1\}^l$

$$Enc(k, m) = (m + k) \bmod N = c$$

$$Dec(k, c) = (c - k) \bmod N = m$$

Perfect Secrecy

Se refiere a que el atacante no gana ninguna información al ver un texto cifrado.

$$\Pr_{k \sim \text{Gen}} [\text{Enc}(k, m_0) = c_0] = \frac{1}{|\mathcal{M}|} \quad \forall m_0 \in \mathcal{M}$$

Lo anterior significa, que para cualquier mensaje, la probabilidad que este sea el que generó mi cifrado c_0 es igual para todos los posibles mensajes. Visto de otra forma, dado un mensaje cifrado, todos los mensajes posibles tienen la misma probabilidad de haberlo generado, por lo que un atacante no gana ninguna información al ver el mensaje. Donde la probabilidad es igual a la probabilidad de generar una llave que encripte m_0 como c_0 .

$$\sum_{k \in \mathcal{K} \mid \text{Enc}(k, m_0) = c_0} \text{Gen}(k)$$

(La suma de la probabilidad de todas las llaves tal que la llave encripte m_0 como c_0)
También podemos hacer definiciones equivalentes y alternativas, como

- Dado $n \leq 2$ mensajes y un texto cifrado, la probabilidad que el cifrado haya sido generado por cualquiera es la misma.
- La distribución de probabilidad de los mensajes es independiente a la distribución de los textos cifrados.
- La probabilidad de ver un mensaje cifrado es la misma si tengo o no conocimiento previo sobre el mensaje.

Si un atacante tiene información sobre el mensaje, digamos que tiene una probabilidad sobre el espacio de mensajes \mathbb{D} , un esquema cumple con *perfect secrecy* si:

$$\frac{\mathbb{D}(m_0) \sum_{k \in \mathcal{K} \mid \text{Enc}(k, m_0) = c_0} \text{Gen}(k)}{\sum_{m \in \mathcal{M}} \sum_{k \in \mathcal{K} \mid \text{Enc}(k, m) = c_0} \mathbb{D}(m) \cdot \text{Gen}(k)} = \mathbb{D}(m_0)$$

¿Es $OTP^{N,l}$ perfectamente secreto?

1. Gen es la distribución uniforme $1/N^l$.
2. Para cada c_0 y cada m_0 existe una única llave k tal que $\text{Enc}(k, m_0) = c_0$.

Sea $c_0 \in \mathcal{C}$ un texto cifrado y m_0 un mensaje

$$\frac{\mathbb{D}(m_0) \cancel{1/N^l}}{\sum_{m \in \mathcal{M}} \mathbb{D}(m) \cdot \cancel{1/N^l}} = \mathbb{D}(m_0) \quad \checkmark \checkmark \checkmark$$

PRP

Definiremos una noción de seguridad usando lo que se llama como una **Pseudo-Random Permutation (PRP)**. Empezamos considerando

$$\mathcal{K} = \mathcal{M} = \mathcal{C} = \{0,1\}^n$$

Y un esquema criptográfico (Gen, Enc, Dec)

Con distribución uniforme entonces elegimos un valor $b \in \{0,1\}$, donde:

- Si $b = 0$ entonces elegimos una llave $k \in \mathcal{K}$ usando Gen y definimos $f(x) = Enc(k, x)$.
- Si $b = 1$ elegimos una permutación π con distribución uniforme y definimos $f(x) = \pi(x)$.

El juego consiste entonces en que el **adversario** elige n palabras $y \in \{0,1\}^n$ y el verificador responde con $f(y)$ para cada caso.

Finalmente el adversario indica si $b = 0$ o $b = 1$, y **gana** si su elección es correcta.

Lo que nos dice este juego, es que **si** al encriptar una cantidad (normalmente polinomial en n) de mensajes, la probabilidad que el adversario pueda responder correctamente el valor de b es significativamente mayor a $\frac{1}{2}$, decimos que nuestro esquema es indistinguible de una permutación al azar (es una **PRP**).

$$\begin{aligned} \Pr(\text{Adversario gane}) &= \\ &\Pr(\text{Adversario gane} \mid b = 0) \cdot \Pr(b = 0) + \\ &\Pr(\text{Adversario gane} \mid b = 1) \cdot \Pr(b = 1) \\ &= \frac{1}{2} \cdot \Pr(\text{Adversario gane} \mid b = 0) + \frac{1}{2} \Pr(\text{Adversario gane} \mid b = 1) \end{aligned}$$

Lamentablemente, OTP no es PRP, ya que si puedo obtener dos mensajes junto con sus respectivos cifrados, puedo usar un mensaje junto con su cifrado (los resto) para obtener una posible llave, si no funciona con el segundo mensaje, entonces puedo decir con certeza que se trata de una permutación, mientras que de lo contrario puedo decir que se trata de la encriptación (La probabilidad de ganar depende de la cantidad de rondas, donde con 2 rondas ya es $\frac{5}{6}$).

Funciones de Hash

Formalmente, llamamos función de hash a un par (Gen, h) de algoritmos tal que, dado un factor de seguridad 1^n , Gen toma el factor y retorna una llave s y h toma la llave y el mensaje como entrada y retorna un hash $h^s(m) \in \{0, 1\}^{l(n)}$.

Una función de hash lleva el espacio completo de posibles mensajes a un espacio fijo

$$h: \mathcal{M} \rightarrow \mathcal{H}$$

Decimos que $h(m)$ es el hash de un mensaje m .

Para esta función, debe ser eficiente el cálculo de la imagen, es decir el cálculo $h(m)$. Sin embargo no debe existir un algoritmo eficiente que dado un hash x encuentre el mensaje tal que $h(m) = x$. Esto se conoce como **resistencia a preimagen**.

Por otro lado, también necesitamos que no exista un algoritmo eficiente que nos deje encontrar dos mensajes $m_1 \neq m_2$ tal que su hash sea el mismo. Esto se conoce como **resistencia a colisiones**.

Generalmente pedimos que (Gen, h) sean un par de algoritmos aleatorizados de tiempo polinomial.

Función despreciable

Una función $f: \mathbb{N} \rightarrow \mathbb{N}$ es despreciable si cumple:

$$(\forall \text{ polinomio } p)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0) \left(f(n) < \frac{1}{p(n)} \right)$$

La definición anterior dice que la función eventualmente va a ser más pequeña que el inverso de cualquier polinomio. La suma y multiplicación de funciones despreciables es despreciable.

Resistencia a Colisiones

Una función de hash será resistente a colisiones si y sólo si para todo **adversario** que funciona como un algoritmo aleatorizado en tiempo polinomial, dado una llave

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

la probabilidad que el adversario elija dos mensajes tal que $h^s(m_1) = h^s(m_2)$ es menor o igual a alguna función despreciable $f(n)$.

$$\Pr(\text{Adversario gane}) \leq f(n)$$

MACs

Definimos un esquema de autenticación de mensajes como una tupla $(Gen, Mac, Verify)$ Definidos sobre los espacios

$$\mathcal{K}, \mathcal{M}, \mathcal{T}$$

- \mathcal{K} corresponde al espacio de llaves, usualmente de la forma Σ^* .
- \mathcal{M} corresponde al espacio de mensajes, usualmente de la forma Σ^* .
- \mathcal{T} corresponde al espacio de *tags*, usualmente de forma Σ^* .
- Gen es un algoritmo aleatorizado, tal que, dado 1^n , genera una llave $k \in \mathcal{K}$ de largo mayor o igual a n .
- $Mac: \mathcal{K} \times \mathcal{M} \rightarrow \mathcal{T}$ es un algoritmo para generar *tags*.
- $Verify: \mathcal{K} \times \mathcal{T} \times \mathcal{M} \rightarrow \{0,1\}$ es un algoritmo para verificar *tags*.

Para todo $k \in \mathcal{K}$ y $m \in \mathcal{M}$, se cumple que:

$$Verify(k, Mac(k, m), m) = 1$$

1. El verificador (nosotros) invoca $Gen(1^n)$ para generar k .
2. El adversario envía $m_0 \in \mathcal{M}$.
3. El verificador responde $Mac(k, m_0)$.
4. Los pasos 2 y 3 se repiten tantas veces como quiera el adversario.
5. El adversario envía (m, t) , siendo m un mensaje que no había enviado antes.
6. El adversario gana si $Verify(k, t, m) = 1$.

Un problema de este esquema es un **replay attack**, donde el mismo mensaje se envía varias veces, sin embargo el esquema no tiene por qué mitigarlo, debe hacerse por parte de la aplicación.

AES

Buscamos una construcción Enc que se vea como una PRP.

$$Enc: \{0,1\}^n \times \{0,1\}^l \rightarrow \{0,1\}^l$$

Lo anterior define 2^n permutaciones, mientras que para $\{0,1\}^l$ existen $2^l!$ permutaciones. Un atacante no puede saber en tiempo polinomial si saco al azar una permutación de una o la otra.

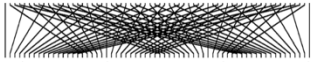
Las tablas de permutaciones son muy grandes, por lo que es razonable usar permutaciones de 8 bits.

Si aplicamos las permutaciones de *byte* en *byte*, tenemos el problema que el cambio en una *bit* sólo se refleja en una de las permutaciones, mientras que si simplemente permutamos los *bits* finales, tenemos un problema similar, son siempre los mismos bits que cambian.

$$m = B_0 \ B_1 \cdots B_6 \ B_7$$

$$m^c = f_0(B_0) \ f_1(B_1) \cdots f_6(B_6) \ f_7(B_7)$$

$$m^c = f_0(B_0) \ f_1(B_1) \cdots f_6(B_6) \ f_7(B_7)$$

$$D(m^c) = B_0^1 \ B_1^1 \ \cdots \ B_6^1 \ B_7^1$$


Para resolver esto, simplemente repetimos varias veces las permutaciones. Esto causa lo conocido como efecto avalancha, donde el cambio en un bit produce un resultado totalmente distinto, cambiando de forma aleatoria.

$$m_0 = B_0^0 \ B_1^0 \ \cdots \ B_6^0 \ B_7^0$$

$$m_0^c = f_0(B_0^0) \ f_1(B_1^0) \cdots f_6(B_6^0) \ f_7(B_7^0)$$

$$m_1 = D(m_0^c) = B_0^1 \ B_1^1 \ \cdots \ B_6^1 \ B_7^1$$

$$m_1^c = f_0(B_0^1) \ f_1(B_1^1) \cdots f_6(B_6^1) \ f_7(B_7^1)$$

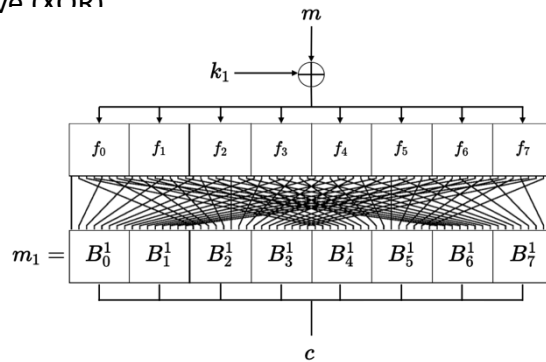
$$m_2 = D(m_1^c) = B_0^2 \ B_1^2 \ \cdots \ B_6^2 \ B_7^2$$

$$m_2^c = f_0(B_0^2) \ f_1(B_1^2) \cdots f_6(B_6^2) \ f_7(B_7^2)$$

⋮

1. Permutamos byte a byte.
2. Al resultado le permutamos todos sus bits.
3. Se repite proceso.

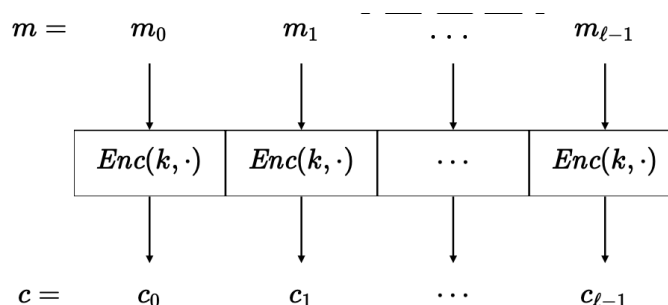
Entre cada uno de estos pasos usamos un *key schedule*, un algoritmo determinista que genera varias llaves a partir de una inicial, donde antes de las permutaciones combinamos el mensaje con la llave (XOR)



Todas estas operaciones (XOR), permutación, son invertibles, por lo que podemos descifrar el mensaje cifrado usando la llave.

ECB (Electronic Code Block)

Se utiliza para cifrar mensajes o datos en bloques independientes utilizando una clave criptográfica. En este modo, cada bloque de datos se cifra por separado utilizando la misma clave, y no hay interdependencia entre los bloques. Esto significa que si un mismo bloque de datos se repite en diferentes partes del mensaje original, los bloques cifrados correspondientes serán idénticos.



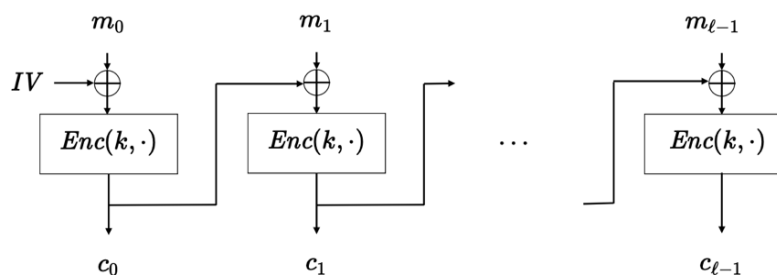
Problema:

Si cambio solo el primer bloque, cambia solo el primer bloque.

CBC (Cipher Block Chaining)

Se utiliza para cifrar mensajes o datos en bloques. A diferencia del modo ECB, en el que cada bloque de datos se cifra de manera independiente, en el modo CBC, se introduce un proceso de retroalimentación en la operación de cifrado para mejorar la seguridad y mitigar las vulnerabilidades del modo ECB.

1. Se divide el mensaje original en bloques de tamaño fijo.
2. Cada bloque se combina con el bloque cifrado anterior (o un vector de inicialización, llamado IV, para el primer bloque) mediante una operación XOR (*bit a bit*).
3. El resultado de la operación XOR se cifra utilizando una clave criptográfica y un algoritmo de cifrado, como AES.
4. El bloque cifrado resultante se convierte en el bloque "anterior" para el siguiente bloque en el proceso.



IV = vector de inicialización aleatorio

Construcción de una función de hash

Se basa en 2 pasos:

1. Definición de una función de hash para mensajes de largo fijo (**función de compresión**).
2. Definir un método que usa la función de compresión iterativamente para construir hashes de mensajes de largo arbitrario.

Función de compresión

Queremos construir una función de hash de largo fijo

$$h: \{0,1\}^{2n} \rightarrow \{0,1\}^n$$

Para esto suponemos que tenemos un esquema criptográfico $(GenEnc, Dec)$ con espacios

$$\mathcal{K} = \mathcal{M} = \mathcal{C} = \{0,1\}^n$$

- Tenemos que $Enc: \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^n$

Dados $u, v \in \{0,1\}^n$, definimos $h(u || v) = Enc(u, v)$

- Recordamos que $u || v$ es la concatenación de u con v .

Problema: Debido a la existencia de Dec , puedo crear muy fácilmente preimágenes para cualquier resultado, por lo que esta definición no es resistente a preimágenes (y por lo tanto, tampoco a colisiones).

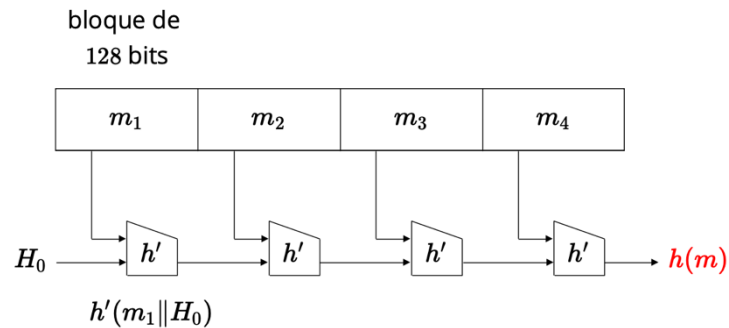
Davies-Meyer

Definimos algo distinta la función de compresión, esta vez como:

$$h(u || v) = Enc(u, v) \oplus v$$

Si (Gen, Enc, Dec) es un esquema criptográfico ideal, entonces la función de hash que se basa en ella con Davies-Meyer es **resistente a colisiones**.

Extensión de largo arbitrario



Dividimos el mensaje en bloques, pasamos cada uno de esos bloques como la llave de nuestra función de encriptación, y el resultado anterior o el vector de inicialización (H_0) como el mensaje.

Para el vector de inicialización H_0 usamos un “*nothing up my sleeve number*” a modo de evitar algún valor cuya preimagen pueda ser ya conocida.

Por ejemplo: considere los primeros 128 bits de la representación de π en binario.

- ¿Cree que alguien conoce la preimagen de este valor H_0 ?

¿Qué hacemos si el largo de m no es divisible por 128?

Padding

Para un mensaje m tal que $m \bmod n \neq 0$ definiremos el mensaje con *padding* $Pad(m)$ tal que:

- m es prefijo de $Pad(m)$, es decir, que agreguemos cosas al final y nunca entre medio ni antes.
- Si 2 mensajes tienen el mismo largo, el *padding* que se les hace es del mismo largo.
- Similarmente, si 2 mensajes tienen distinto largo, el último bloque de ambos mensajes debe ser distinto.

Función de padding útil usada en SHA-256

- Agrego un 1 al final.
- Agrego ceros hasta terminar el bloque completo.
- Agrego como último bloque el largo del mensaje, módulo 2^{64} .

$Pad(m)$	m_1	m_2	m_3 10...0	$ m \bmod 2^n$
----------	-------	-------	--------------	-----------------

Vector inicialización SHA-256

- $H_0 = H_0^1 H_0^2 H_0^3 H_0^4 H_0^5 H_0^6 H_0^7 H_0^8$
- H_0^i tiene los primeros 32 bits de la parte decimal de la raíz cuadrada del i -ésimo número primo.

Por ejemplo, H_0^1 tiene los primeros 32 bits de la parte decimal de $\sqrt{2}$.

Merkle-Damgård

La construcción de **Merkle-Damgård** toma la función de compresión definida con *Davies-Meyer* y una función de *padding* que satisface los puntos anteriores.

Si la función h' es resistente a colisiones, entonces h también lo es. Esto se debe a que si h no es resistente a colisiones entonces puedo fácilmente encontrar dos mensajes, si estos tienen el mismo tamaño, $h(m1) = h(m2)$, existe una colisión también en alguna llamada a h' , por lo que h' no puede ser resistente a colisiones.

Si tienen distinto tamaño, sé que el último bloque debe ser distinto por lo que debe existir una colisión en la última función de compresión.

HMAC

La función principal de HMAC es generar un código de autenticación que se adjunta al mensaje original. Este código se calcula utilizando una función hash criptográfica y una clave secreta compartida entre el remitente y el receptor. El receptor puede verificar la autenticidad del mensaje recalculando el código HMAC usando la misma clave y la misma función hash, y luego comparándolo con el código HMAC recibido. Si coinciden, el receptor puede tener un alto grado de confianza en que el mensaje no ha sido alterado y proviene de la fuente esperada.

Tengamos por ejemplo la función $MAC(k, m) = h(k, m)$.

Un **ataque de extensión** se refiere a el poder calcular el hash de $h(k||m||m')$ dado solamente $h(k||m)$.

Un ejemplo se puede dar con $pad(k||m)$ ya que $pad(pad(k||m))$ será dividido en los mismos bloques, por lo que en la penúltima aplicación de h' me dará el resultado de $h(k||m)$.

Una función que es más segura es la siguiente:

$$MAC(k, m) = h(k_1 || h(k_2 || m))$$

Donde k_1 y k_2 ocupan exactamente un bloque, son distintas, se derivan de manera determinista a partir de k y no se pueden obtener sin k .

$$k' = \begin{cases} h(k) & k \text{ usa mas de un bloque} \\ k & \text{en otro caso} \end{cases}$$

$$k_1 = k' \oplus 5c \dots 5c \quad k_2 = k' \oplus 36 \dots 36$$

$$\begin{aligned} 5c &= 01011100 \\ 36 &= 00110110 \end{aligned}$$

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

KDF (Key Derivation Functions)

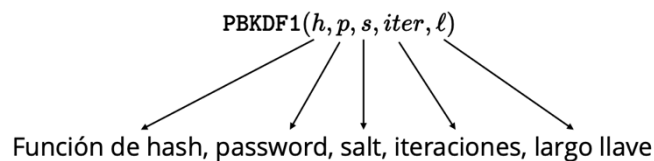
Quiero buscar una llave, pero el atacante puede tener una lista con muchísimas de las contraseñas más comunes. Lo mismo ocurre si usamos el hash, esta vez con **rainbow tables** (*pass, hash*) pre-calculadas.

Para evitar el precálculo, genero el hash con **salt**, que es un string aleatorio que envío con mi mensaje cifrado.

Con una *key-derivation function*, no nos cuesta tanto calcular una vez, sin embargo cuesta muchísimo calcular para todos los posibles valores de una supuesta *rainbow table*.

PBKDF 1

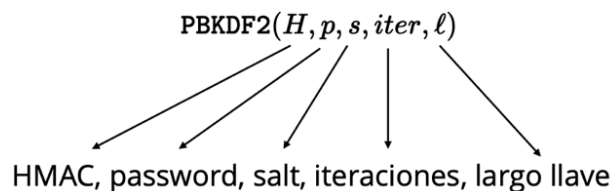
La función saca el *hash* de $p||s$, repitiendo el *hash* del resultado anterior una cantidad fija de veces, sacando los primeros *key-len* bits del resultado final.



PBKDF 2

La función saca HMAC de $p, s||1$ varias veces y retorna el XOR de todos los resultados.

WPA2 usa PBKDF2 con 4096 iteraciones, un resultado de 256 bits, HMAC-SHA1 y el SSID como salt.



Aritmética modular

Dados $a, n \in \mathbb{Z}$, existe un único par de elementos $(q, r) \in \mathbb{Z}^2$ tal que:

$$0 \leq r < |n|$$

$$a = q \cdot n + r$$

Donde q es el cociente, y r el resto.

Decimos entonces que:

$$a \bmod n = r$$

Si $a \equiv_n b \wedge c \equiv_n d$, entonces se cumple:

$$(a + c) \equiv_n (b + d)$$

$$(a \cdot c) \equiv_n (b \cdot d)$$

Máximo común divisor

$$MCD(a, b) = \begin{cases} a & \text{si } b = 0 \\ MCD(b, a \bmod b) & \text{si } b > 0 \end{cases}$$

Algoritmo extendido de Euclides

Suponga que $a \geq b$, y defina la siguiente **sucesión**:

$$r_0 = a$$

$$r_1 = b$$

$$r_i = r_{i-2} \bmod r_{i-1} \quad (i \geq 2)$$

y calculamos esta sucesión hasta un número k tal que $r_k = 0$.

Luego, $r_{k-1} = \mathbf{gcd}(a, b)$.

Al mismo tiempo, podemos ir calculando dos sucesiones s_i, t_i tales que:

$$r_i = s_i \cdot a + t_i \cdot b$$

Tenemos que:

$$\mathbf{gcd}(a, b) = r_{k-1} = s_{k-1} \cdot a + t_{k-1} \cdot b$$

Sean:

$$s_0 = 1$$

$$t_0 = 0$$

$$s_1 = 0$$

$$t_1 = 1$$

Se tiene que:

$$\begin{aligned} r_0 &= s_0 \cdot a + t_0 \cdot b = a \\ r_1 &= s_1 \cdot a + t_1 \cdot b = b \end{aligned}$$

Dado que $r_{i-1} = \overbrace{\left\lfloor \frac{r_{i-1}}{r_i} \right\rfloor}^{\text{División parte entera}} \cdot r_i + \overbrace{r_{i-1} \bmod r_i}^{\text{Resto}}$, tenemos que:

$$r_{i-1} = \left\lfloor \frac{r_{i-1}}{r_i} \right\rfloor \cdot r_i + r_{i+1}$$

Por lo tanto:

$$s_{i-1} \cdot a + t_{i-1} \cdot b = \left\lfloor \frac{r_{i-1}}{r_i} \right\rfloor \cdot (s_i \cdot a + t_i \cdot b) + r_{i+1}$$

Concluimos que:

$$r_{i+1} = \left(s_{i-1} - \left\lfloor \frac{r_{i-1}}{r_i} \right\rfloor \cdot s_i \right) \cdot a + \left(t_{i-1} - \left\lfloor \frac{r_{i-1}}{r_i} \right\rfloor \cdot t_i \right) \cdot b$$

Definimos entonces:

$$\begin{aligned} s_{i+1} &= s_{i-1} - \left\lfloor \frac{r_{i-1}}{r_i} \right\rfloor \cdot s_i \\ t_{i+1} &= t_{i-1} - \left\lfloor \frac{r_{i-1}}{r_i} \right\rfloor \cdot t_i \end{aligned}$$

Ejemplo

Vamos a usar el algoritmo para $a = 60$ y $b = 13$

Inicialmente:

$$\begin{array}{lll} r_0 = 60 & s_0 = 1 & t_0 = 0 \\ r_1 = 13 & s_1 = 0 & t_1 = 1 \end{array}$$

Entonces tenemos que:

$$\begin{aligned} r_2 &= r_0 \bmod r_1 \\ s_2 &= s_0 - \left\lfloor \frac{r_0}{r_1} \right\rfloor \cdot s_1 \\ t_2 &= t_0 - \left\lfloor \frac{r_0}{r_1} \right\rfloor \cdot t_1 \end{aligned}$$

Por lo tanto:

$$r_2 = 8 \quad s_2 = 1 \quad t_2 = -4$$

Y el proceso continúa:

$$\begin{array}{lll}
 r_3 = 5 & s_3 = -1 & t_3 = 5 \\
 r_4 = 3 & s_4 = 2 & t_4 = -9 \\
 r_5 = 2 & s_5 = -3 & t_5 = 14 \\
 r_6 = 1 & s_6 = 5 & t_6 = -23 \\
 r_7 = 0 & s_7 = -13 & t_7 = 60
 \end{array}$$

Tenemos que: $1 = 5 \cdot 60 + (-23) \cdot 13$

Ejemplo para mortales:

$$\gcd(888, 54) = 6:$$

$$888 = 54(16) + 24$$

$$54 = 24(2) + 6$$

$$24 = 6(4) + 0$$

¿Cuántas veces
cabe el 54 en 888?

Nuestro último resto
 $\neq 0$ será el gcd

Inverso modular

b es inverso de a en módulo n si:

$$a \cdot b \equiv_n 1$$

Un número es invertible en módulo n si y sólo si $MCD(a, n) = 1$. Si n es primo, todos los números menores a n son invertibles en módulo n . Para encontrar los inversos, usamos el algoritmo extendido de Euclides.

RSA

- Generar 2 números primos distintos P y Q . Y diremos que $N = P \cdot Q$. Para esto eliges un número al azar y chequeas si es o no primo. La probabilidad que el numero sea primo es aproximadamente $\frac{1}{\ln(n)}$.
- Definimos $\phi(N) = (P - 1) \cdot (Q - 1)$.
- Generar un número d tal que sea coprimo con $\phi(N)$. Es decir, se tiene que cumplir que $MCD(d, \phi(N)) = 1$.
- Obtener e , el inverso multiplicativo de d en modulo $\phi(N)$, es decir, se tiene que cumplir que $e \cdot d \equiv 1 \pmod{\phi(N)}$. Para esto, usaremos el algoritmo extendido de Euclides.
- Definimos $S_A = (d, N)$ y $P_A = (e, N)$.

La función de encriptación es entonces

$$Enc(P_A, m) = m^e \bmod N$$

$$Dec(S_A, c) = c^d \bmod N$$

Ejemplo:

1. Sean $P = 19$ y $Q = 31$. Tenemos que $N = 589$.
2. $\phi(N) = \phi(589) = 18 \cdot 30 = 540$.
3. Sea $d = 77$, para el cual se tiene que $MCD(77, 540) = 1$.
4. Calculamos inverso $e = 533$ de 77 en módulo 540 .
$$533 \cdot 77 \bmod 540 = 1$$
5. Llave pública: $P_A = (533, 589)$ y llave secreta: $S_A = (77, 589)$.

Tenemos que:

$$Enc(P_A, m) = m^{533} \bmod 589$$

$$Dec(S_A, c) = c^{77} \bmod 589$$

Para $m = 121$ tenemos:

$$Enc(P_A, 121) = 121^{533} \bmod 589 = 144$$

$$Dec(S_A, 144) = 144^{77} \bmod 589 = 121$$

En general, tendremos que:

$$Dec(S_A, c) = (m^e)^d \bmod N = m^{ed} \bmod N = m \bmod N$$

Esto se debe ya que, como e y d son inversos en $\bmod \phi(N)$, $ed = \alpha \cdot \phi(N) + 1$.

Por el pequeño teorema de Fermat,

$$m^{\phi(N)} \bmod N = 1 \rightarrow m^{k \cdot \phi(N)} \bmod N = 1 \rightarrow m^{k \cdot \phi(N) + 1} \bmod N = m$$

$$m^{ed} \bmod N = m^{\alpha \cdot \phi(N) + 1} \bmod N = 1 \cdot m \bmod N$$

La seguridad de RSA depende de que tan difícil sea descomponer $N = P \cdot Q$.

Test de primalidad

Si un número es primo, todos los números menores a él cumplen que $a^{n-1} \equiv 1 \pmod{n}$, el tamaño del conjunto de todos estos números es $p - 1$. En el caso contrario, el tamaño es siempre menor a $p - 1$.

Si n es compuesto, el tamaño es menor o igual a $\frac{n-1}{2}$ (falso pero sirve).

Consideramos testigos de forma:

$$a^{\frac{n-1}{2}} \pmod{n} = 1 \vee -1 \pmod{n}$$

Si p es primo, entonces el tamaño del conjunto de testigos tal que lo anterior da 1 es igual al tamaño del conjunto de testigos que dan $-1 \pmod{p}$.

Para revisar si un número p' es primo, elegimos k números al azar. Si vemos que el número elegido divide a p' , entonces no es primo. Si observamos un número tal que $a^{\frac{n-1}{2}} \pmod{n} = -1$, decimos que el número es primo. Si el número no es 1 o -1 , no puede ser primo. Finalmente si ningún testigo nos entrega -1 , decimos que el número no es primo, con un error de 1.

Teoría de Grupos

Un grupo es un par $(G, *)$ donde G es un conjunto y $*$ es una operación $*: G \times G \rightarrow G$ que satisface que existe un **neutro**, un **inverso** para todo elemento de G y además que es **asociativo**. El orden de un grupo se define como el tamaño del conjunto que lo conforma.

Llamamos \mathbb{Z}_n^* Al grupo multiplicativo de los enteros módulo n , donde **los elementos de G son todos los coprimos de n** . Por esto, si **n es primo entonces G son todos los números menores a n** .

Ejemplo:

Si queremos \mathbb{Z}_{10}^* . Los números coprimos a 10 (elementos que no comparten factores primos con 10) serían 1, 3, 7 y 9. Por lo tanto:

$$\mathbb{Z}_{10}^* = \{1, 3, 7, 9\}$$

Un subgrupo de $(G, *)$ es un subconjunto H de G tal que $(H, *)$ también es un grupo. El orden de un subgrupo **siempre divide al orden del grupo original**.

Ahora definimos, usando notación multiplicativa,

$$a^n = a * a * \dots * a$$

Dada una secuencia $\{1, a, a^2, a^3, \dots\}$ eventualmente tendremos una repetición, ya que el grupo es finito. Eventualmente debe existir entonces un k tal que $a^k = 1$. Dado un elemento cualquiera de la secuencia, buscamos cuánto se demora en repetirse y ese número es nuestro k . La primera repetición siempre será 1 (neutro).

Definimos ahora

$$\langle a \rangle = \{a^j \mid 0 \leq j < k\}$$

Esto es un subgrupo de tamaño k . Por lo tanto, k divide a $|G|$.

Dado \mathbb{Z}_n^* y $|\langle a \rangle| = k$, \mathbb{Z}_n^* tiene $\phi(n)$ elementos y k divide a ese número.

$$a^{\phi(n)} = a^{\alpha \cdot k} = (a^k)^\alpha = 1^\alpha = 1$$

Ejemplo:

Considere $(\mathbb{Z}_{10}^*, \cdot)$

$$\begin{aligned}\langle 1 \rangle &= \{1\} \\ \langle 3 \rangle &= \{1, 3, 7, 9\} \\ \langle 7 \rangle &= \{1, 3, 7, 9\} \\ \langle 9 \rangle &= \{1, 9\}\end{aligned}$$

Grupos cíclicos

Un grupo $(G, *)$ es cíclico si existe $a \in G$ tal que $\langle a \rangle = G$

$(\mathbb{Z}_{10}^*, +)$ es cíclico ya que $\langle 3 \rangle = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

$(\mathbb{Z}_{10}^*, \cdot)$ es cíclico ya que $\langle 7 \rangle = \{1, 3, 7, 9\}$

Dado un grupo, si existe algún elemento tal que $\langle a \rangle$ es el grupo original, el grupo es cíclico. \mathbb{Z}_p^* siempre es cíclico.

Ejemplo: (\mathbb{Z}_8^*, \cdot) NO es cíclico

En primer lugar, tenemos que:

$$\mathbb{Z}_8^* = \{1, 3, 5, 7\}$$

Luego,

$$\begin{aligned}\langle 1 \rangle &= \{1\} \\ \langle 3 \rangle &= \{1, 3\} \\ \langle 5 \rangle &= \{1, 5\} \\ \langle 7 \rangle &= \{1, 7\}\end{aligned}$$

Por lo tanto, como no existe un $a \in G$ tal que $\langle a \rangle = G$, el grupo NO es cíclico.

Diffie-Hellman y ElGamal

Diffie-Hellman

Al igual que calcular exponentes en RSA, tenemos que dado un grupo G y un elemento $g \in G$, el valor $y = g^x$ es difícil calcular x si $|\langle g \rangle|$ es grande.

Queremos usar elementos de orden primo, ya que a priori sabemos el orden del subgrupo generado y si $|\langle g \rangle|$ es compuesto puedo usar el teorema Chino del resto para ayudarme a calcular x . Esto no pasa con, por ejemplo, curvas elípticas.

Para *Diffie-Hellman*, se elige un G y $g \in G$, donde cada persona elige también un número entre 1 y $|\langle g \rangle|$, la clave secreta compartida es entonces

$$g^x, g^y, g^{xy}$$

La persona que sabe x le comparten g^y y puede calcular g^{xy} y vice versa. Se asume que dado g , g^x y g^y un atacante no puede adivinar x o y en tiempo polinomial.

Forward secrecy se refiere a que si una llave es comprometida, no compromete información antigua. Por ejemplo generando una nueva llave cada sesión.

ElGamal

Se genera primero una llave pública g^x , luego cuando otra persona quiera enviar un mensaje esta debe generar una llave efímera y y enviar $(m * g^{xy}, g^y)$.

Si el orden del grupo es q , entonces podemos hacer

$$g^{xy} \cdot g^{y(q-x)} = g^{xy+yq-xy} = g^{yq}$$

Como yq es múltiplo del orden del grupo, g^{yq} es el neutro, por lo tanto

$$m \cdot g^{xy} \cdot g^{y(q-x)} = m \cdot g^{yq} = m$$

Para ver el orden de un subgrupo $\langle g \rangle$ de Z_p^* , puedo obtener los divisores de $p - 1$, donde el divisor mas pequeño k tal que $g^k \bmod p = 1$.

Firmas digitales

Podemos generar firmas para cualquier persona que necesite cerciorar que el mensaje fue enviado por nosotros.

Con RSA , supongamos que tenemos ya generadas las llaves públicas y privadas. Para cada mensaje $m \in \{0, \dots, N - 1\}$ sabemos que

$$Dec(S_A, Enc(P_A, m)) = m$$

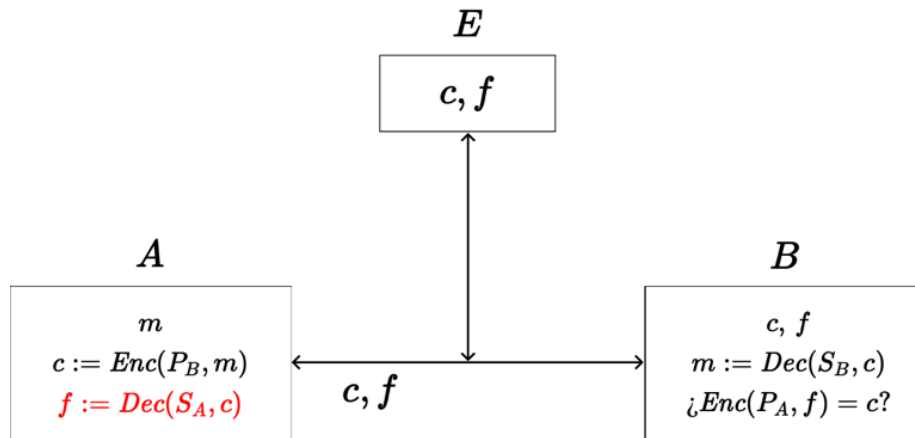
Debido a las propiedades de RSA, también se cumple que:

$$\begin{aligned} Enc(P_A, Dec(S_A, m)) &= (m^d \bmod N)^e \bmod N \\ &= (m^d)^e \bmod N \\ &= m^{d \cdot e} \bmod N \\ &= (m^e)^d \bmod N \\ &= (m^e \bmod N)^d \bmod N \\ &= Dec(S_A, Enc(P_A, m)) \\ &= m \end{aligned}$$

Definimos entonces la firma del mensaje m por el usuario A como:

$$f := Dec(S_A, m)$$

Solo A puede generar esta firma. Cualquier usuario puede verificar si A firmó un mensaje usando la llave pública P_A de A .



Problema:

Firmar un mensaje m puede ser lento si m es un mensaje muy largo.

Para solucionar este problema, se puede firmar $h(m)$ en lugar de firmar m , donde h es una función de hash.

Firmas de Schorr

Suponemos dado un grupo finito $(G, *)$ y un elemento $g \in G$ tal que $|\langle g \rangle| = q$.

- G, g y q son públicos.
- Como vimos antes, se debe tener que $|G|$ y q son números grandes.

Además, suponemos dada una función de hash h :

- La llave privada de un usuario A es $x \in \{1, \dots, q-1\}$ y su llave pública es g^x .

El usuario A tiene que firmar un mensaje m .

A firma de la siguiente forma:

1. Genera al azar $k \in \{1, \dots, q-1\}$ y calcula $r = g^k$.
2. Calcula $v = h(r || m)$ usando r como un *string*.
3. Calcula $s = k + v \cdot x$ interpretando v como un número natural.
4. La firma de m es (v, s) .

Se puede verificar que (v, s) es una firma de m generada por A de la siguiente forma:

1. Calcule $\alpha = g^s = g^{k+v \cdot x} = g^k * g^{v \cdot x} = g^k * (g^x)^v$
2. Calcule $\beta = \alpha * ((g^x)^v)^{-1} = g^k * (g^x)^v * ((g^x)^v)^{-1} = g^k$
3. Verifique si $v = h(\beta || m) = h(g^k || m)$

Lo bueno de estas firmas es que son más pequeñas y fáciles de combinar si se necesitan varias firmas.

Dados n usuarios con llaves privadas x_n y llaves públicas g^{x_n} , la llave pública para verificar la firma de m por todos los usuarios es:

$$g^{x_1} * g^{x_2} * \dots * g^{x_n} = g^{\sum x_n}$$

Se generan al azar k_n y se calculan $r_n = g^{k_n}$, luego todos calculan:

$$v = h((r_1 * \dots * r_n) || m)$$

y

$$s_n = k_n + v \cdot x_n$$

Se calcula $s = \sum s_n$ y la firma es (v, s) .

$$\alpha = g^s = g^{\sum s_n} = g^{\sum k_n} * g^{\sum v \cdot x_n} = g^{\sum k_n} * (g^{\sum x_n})^v$$
$$\beta = \alpha * \left(\left(\prod g^{x_n} \right)^v \right)^{-1} = g^{\sum k_n} * (g^{\sum x_n})^v * ((g^{\sum x_n})^v)^{-1} = g^{\sum k_n}$$

Luego verificamos al igual que para el caso inicial.

Seguridad Web

Puedo usar autenticación con la llave pública para ver que efectivamente los mensajes están firmados correctamente, sin embargo necesito alguien que me corrobore que efectivamente la llave obtenida es del sitio web de confianza.

Existen *Certificate Authorities* (**CAs**) que firman certificados que nos aseguran que efectivamente la llave pública es la correcta. Para obtener los certificados la página debe firmar algo con su llave privada, además de ingresar un valor específico en alguna de sus páginas a modo de verificar que efectivamente son ellos los dueños. Luego es posible certificar cualquier nueva llave pública con la llave pública anterior verificada.

TLS

TLS, que significa “*Transport Layer Security*” (Seguridad de la Capa de Transporte), es un protocolo de seguridad utilizado para cifrar las comunicaciones en Internet y garantizar la privacidad y la integridad de los datos transmitidos entre dos sistemas. TLS es la evolución y el sucesor del protocolo SSL (*Secure Sockets Layer*) y se utiliza principalmente para establecer conexiones seguras a través del protocolo HTTPS, que es la versión segura del protocolo HTTP utilizado para acceder a sitios web.

A continuación, te explico cómo funciona TLS de manera simplificada:

1. **Inicio de la Comunicación:** Cuando un cliente (por ejemplo, un navegador *web*) desea establecer una conexión segura con un servidor (por ejemplo, un sitio *web*), el proceso de TLS comienza con una solicitud del cliente al servidor. El servidor responde indicando su disposición para usar TLS y proporciona su certificado digital.
2. **Verificación del Certificado:** El cliente recibe el certificado digital del servidor. Este certificado contiene la clave pública del servidor y está firmado por una Autoridad de Certificación (como *Let's Encrypt* o una entidad certificadora comercial). El cliente verifica la autenticidad del certificado comprobando la firma y asegurándose de que el certificado no esté revocado.

3. **Generación de Claves de Sesión:** Una vez que se verifica el certificado del servidor, el cliente y el servidor generan claves de sesión únicas que se utilizarán para cifrar y descifrar los datos durante la comunicación. Estas claves son temporales y se generan para cada conexión.
4. **Intercambio de Claves y Parámetros Criptográficos:** El cliente envía al servidor una clave de sesión cifrada con la clave pública del servidor. Esto garantiza que solo el servidor pueda descifrar la clave de sesión. Además, durante este proceso, se acuerdan los parámetros criptográficos, como los algoritmos de cifrado y de intercambio de claves que se utilizarán para proteger la comunicación.
5. **Cifrado de Datos:** A partir de este punto, toda la comunicación entre el cliente y el servidor se cifra utilizando la clave de sesión. Esto garantiza que cualquier dato transmitido entre ellos esté protegido y no pueda ser interceptado ni entendido por terceros no autorizados.
6. **Integridad de los Datos:** TLS también proporciona mecanismos para verificar que los datos no se hayan alterado durante la transmisión. Esto se logra mediante el uso de funciones de hash y códigos de autenticación de mensajes (MAC) para garantizar que los datos lleguen sin cambios.
7. **Cierre de la Conexión:** Una vez que la comunicación ha terminado, ya sea porque el cliente ha finalizado la solicitud o porque se ha completado una transacción, se realiza un proceso de cierre de conexión seguro para asegurarse de que ambas partes estén al tanto de la finalización de la comunicación.

