Benemérita Universidad Autónoma de Puebla

Facultad de Ciencias de la Computación

Ingeniería en Ciencias de la Computación

Materia: Programación Distribuida Aplicada

Tarea algoritmos de TLS

Profesor: Gustavo Emilio Mendoza Olguin

Alumnos:

Pérez Flores Ivonne 202141158

Primavera 2025

20 de enero de 2025

Algoritmos más usados por TLS

- **Encriptación.** Existen dos tipos de algoritmos de cifrado usados por TLS, simétrico y asimétrico.
  - *AES(Advanced Encryption Standard).*
    - Es un cifrado simétrico en bloques que se utiliza para cifrar datos confidenciales. Tanto por seguridad como por velocidad, AES se ha convertido en un estándar de seguridad para usuarios y aplicaciones que necesitan una encriptación fácil de usar.
    - Como todos los métodos de encriptación, el AES convierte el texto sin formato en un código que sólo puede descifrar quien tenga la clave (o cifrado). Sin embargo, la principal diferencia entre AES y otros métodos de cifrado es que AES utiliza varias rondas de transposición, sustitución y mezcla en lugar de una única fase de cifrado.
    - Pseudocodigo.

```
Funcion AES_Cifrado(mensaje, clave):
	Entrada:
		mensaje: Bloque de 128 bits a cifrar.
		clave: Clave de 128, 192 o 256 bits.
	Salida:
		mensaje_cifrado: Bloque cifrado de 128 bits.

	1. Generar las subclaves a partir de la clave original.  # KeyExpansion

	2. Inicializar el estado:
		estado = mensaje XOR subclave_inicial

	3. Realizar las rondas principales:
		Para cada ronda (excepto la última):
		estado = SustituirBytes(estado)      # SubBytes
		estado = DesplazarFilas(estado)      # ShiftRows
		estado = MezclarColumnas(estado)     # MixColumns
		estado = estado XOR subclave_ronda

	4. Realizar la ronda final (sin MezclarColumnas):
		estado = SustituirBytes(estado)
		estado = DesplazarFilas(estado)
		estado = estado XOR subclave_final

	5. Retornar estado como mensaje_cifrado.
```

- ***RC4 (Rivest Cipher 4).***
    - Un algoritmo de cifrado de flujo que fue ampliamente utilizado en versiones más antiguas de TLS.
    - Actualmente está **prohibido** debido a varias vulnerabilidades críticas (p.ej., ataques de sesgo de clave).

- Pseudocodigo.

```
Funcion RC4(clave, mensaje):
    Entrada:
        clave: Cadena de bytes de longitud variable (entre 1 y 256 bytes).
        mensaje: Cadena de bytes a cifrar o descifrar.
    Salida:
        resultado: Mensaje cifrado o descifrado.

    1. Inicialización del estado:
        S = [0, 1, 2, ..., 255]              # Array de permutación
        j = 0
        Para i desde 0 hasta 255:
            j = (j + S[i] + clave[i % longitud(clave)]) mod 256
            Intercambiar S[i] y S[j]

    2. Generación del keystream:
        i = 0
        j = 0
        Para cada byte en el mensaje:
            i = (i + 1) mod 256
            j = (j + S[i]) mod 256
            Intercambiar S[i] y S[j]
            K = S[(S[i] + S[j]) mod 256]      # Byte del keystream
            salida.append(K)

    3. Cifrado o descifrado:
        Para cada byte en el mensaje:
            resultado[byte] = mensaje[byte] XOR salida[byte]

    4. Retornar resultado.
```

- **ChaCha20.**
    - ChaCha20 es un algoritmo de cifrado por flujo, lo que indica que el proceso de cifrado se hace bit a bit sobre un mensaje o información que se desee cifrar. Los algoritmos de cifrado por flujo son de llave privada, por lo que se requiere la misma llave para cifrar y descifrar. El número 20 viene de que ChaCha20 realiza 20 rondas de funciones no lineales para poder cifrar información, es más rápido que el algoritmo de cifrado AES y su diseñador lo recomienda para aplicaciones criptográficas típicas. ChaCha20 está diseñado para ser soportado por dispositivos que no posean una capacidad de procesamiento tan grande como lo son dispositivos IoT, teléfonos celulares, relojes inteligentes, etc. ya que los algoritmos de cifrado por flujo están diseñados para ser rápidos y resistente a errores.

- Pseudocodigo

```
Funcion ChaCha20(clave, contador, nonce, mensaje):
    1. Inicializar el estado:
        estado = ["Chacha20", clave, contador, nonce]

    2. Realizar 20 rondas de mezclado sobre el estado:
        Para cada ronda:
            Actualizar el estado con rotaciones y XOR.

    3. Generar el keystream:
        Concatenar el estado después de las rondas para crear el keystream.

    4. Cifrar/Descifrar:
        Para cada byte del mensaje:
            mensaje[i] = mensaje[i] XOR keystream[i]

    5. Retornar el mensaje cifrado/descifrado.
```

- **Autenticación.**
  - **MAC**

- Propósito: Proporcionar autenticación de mensaje y garantizar que los datos no han sido modificados.
- Descripción: Similar a HMAC, pero sin necesidad de una función hash específica. El MAC utiliza una clave secreta para generar un código de autenticación a partir de un mensaje. El receptor debe tener la misma clave para verificar la autenticidad del mensaje.
- Funcionamiento: El emisor genera el MAC usando una clave secreta y lo envía junto con el mensaje. El receptor genera el MAC del mensaje recibido con la misma clave y compara con el MAC recibido.
- Pseudocodigo.

```
Funcion MAC(clave, mensaje):
    Entrada:
        clave: Clave secreta compartida.
        mensaje: Mensaje que se desea autenticar.
    Salida:
        mac: Código de autenticación del mensaje.

    1. Inicializar una variable 'mac' con un valor inicial (por ejemplo, 0).

    2. Para cada bloque del mensaje:
        - Concatenar el bloque del mensaje con la clave secreta.
        - Aplicar una función de hash (por ejemplo, SHA-256) al valor concatenado.
        - Actualizar 'mac' con el resultado del hash.

    3. Después de procesar todo el mensaje, el valor de 'mac' es el código de autenticación.
```

```
    4. Retornar el valor final de 'mac'.
```

- **HMAC (Hashed Message Authentication Code)**
  - Propósito: Verificar la integridad y autenticidad de un mensaje.
  - Descripción: HMAC utiliza una función hash criptográfica (como SHA-256) junto con una clave secreta para generar un código de autenticación. HMAC garantiza que el mensaje no haya sido alterado y que provenga de una fuente auténtica que conoce la clave secreta.
  - Pseudocodigo

```
Funcion HMAC(clave, mensaje):
    Entrada:
        clave: Clave secreta compartida.
        mensaje: Mensaje que se desea autenticar.
    Salida:
        hmac: Código de autenticación del mensaje.

    1. Definir dos valores de padding:
        ipad = 0x36 * 64  # Padding con el valor hexadecimal 0x36, de 64 bytes de longitud.
        opad = 0x5C * 64  # Padding con el valor hexadecimal 0x5C, de 64 bytes de longitud.

    2. Asegurarse de que la longitud de la clave es 64 bytes:
        Si la longitud de la clave es menor a 64 bytes:
            Extender la clave con ceros hasta 64 bytes.
        Si la longitud de la clave es mayor a 64 bytes:
            Aplicar una función hash (por ejemplo, SHA-256) a la clave para reducirla a 64
bytes.

    3. Preparar los valores internos para el cálculo de HMAC:
        K_ipad = clave XOR ipad  # XOR bit a bit de la clave con ipad.
        K_opad = clave XOR opad  # XOR bit a bit de la clave con opad.

    4. Calcular el valor intermedio:
        paso1 = H(K_ipad || mensaje)  # Aplicar la función hash (H) al concatenar K_ipad y el
mensaje.

    5. Calcular el HMAC final:
        hmac = H(K_opad || paso1)  # Aplicar la función hash (H) al concatenar K_opad y el
valor intermedio (paso1).

    6. Retornar el valor de 'hmac'.
```

- **SHA-1 / SHA-2 (con HMAC)**
  - Propósito: Verificación de la integridad del mensaje y autenticación utilizando funciones hash seguras.
  - Descripción: Estas funciones de hash (SHA-1 y SHA-2) pueden ser usadas junto con un MAC (Message Authentication Code) para garantizar la integridad y autenticidad del mensaje.
  - SHA-1 genera un valor hash de 160 bits y SHA-2 genera un valor hash de 224, 256, 384, o 512 bits.
  - Pseudocodigo

```
Funcion HMAC(clave, mensaje, hash_func):
    Entrada:
        clave: Clave secreta compartida.
        mensaje: Mensaje a autenticar.
        hash_func: Función hash (SHA-1 o SHA-2).
    Salida:
        hmac: Código de autenticación del mensaje.

    1. Si la clave es más corta que 64 bytes, extenderla con ceros.
    2. Si la clave es más larga que 64 bytes, aplicar hash_func a la clave para reducirla a 64
bytes.

    3. K_ipad = clave XOR 0x36 (64 bytes)
    4. K_opad = clave XOR 0x5C (64 bytes)

    5. paso1 = hash_func(K_ipad || mensaje)   # Hash de K_ipad + mensaje
    6. hmac = hash_func(K_opad || paso1)      # Hash de K_opad + paso1

    7. Retornar hmac
```

- **Integridad**
  - **CRC (Cyclic Redundancy Check)**
    - El CRC es un algoritmo que detecta cambios accidentales en los datos.
    - Utiliza una operación de división polinómica para generar un código de verificación basado en el contenido de los datos.
    - Pseudocodigo

```
Funcion CRC(mensaje, polinomio):
    Entrada:
        mensaje: Datos a verificar.
        polinomio: Polinomio divisor utilizado para calcular el CRC.
    Salida:
        crc: Valor CRC calculado.

    1. Inicializar crc = 0.
```

```
    2. Para cada byte en el mensaje:
        a. Realizar una división binaria (XOR) entre crc y el byte del mensaje.
        b. Desplazar crc a la derecha y realizar operaciones según el polinomio.
    3. Retornar crc
```

- **MD5 (Message Digest Algorithm 5)**
    - MD5 es una función de hash criptográfico que produce un valor de 128 bits (16 bytes) como resumen del mensaje. Aunque ha sido superado en términos de seguridad, sigue siendo un ejemplo común de un algoritmo de integridad.
    - Pseudocodigo

```
Funcion MD5(mensaje):
    Entrada:
        mensaje: Datos a verificar.
    Salida:
        md5: Resumen MD5 del mensaje.

    1. Inicializar los valores de los registros A, B, C, D.
    2. Dividir el mensaje en bloques de 512 bits.
    3. Para cada bloque:
        a. Realizar una serie de operaciones de mezcla y actualización de registros.
    4. Concatenar los registros A, B, C, D para formar el resumen MD5.
    5. Retornar md5.
```

- **Digital Signature Algorithm (DSA)**
    - El DSA es un algoritmo de firma digital que garantiza tanto la integridad de los datos como la autenticidad, utilizando un par de claves (pública/privada) y un algoritmo de hash (generalmente SHA-1 o SHA-256).
    - Pseudocodigo

```
Funcion DSA(clave_privada, mensaje):
    Entrada:
        clave_privada: Clave privada del firmante.
        mensaje: Datos a firmar.
    Salida:
        firma: Firma digital del mensaje.

    1. Calcular el hash del mensaje usando SHA-1 o SHA-256.
    2. Generar un número aleatorio k dentro de un rango.
    3. Calcular un valor r basado en k y el mensaje hash.
    4. Calcular el valor s basado en r, k, el hash del mensaje y la clave privada.
    5. La firma es (r, s).
    6. Retornar firma.
```

Bibliografía

101 Computing. (2018, octubre 18). Algoritmos hash para la validación de integridad. 101 Computing. https://www.101computing.net/hashing-algorithms-for-integrity-validation/

IBM. (2024, enero 19). Configuración de la autenticación y el cifrado de mensajes. IBM. https://www.ibm.com/docs/es/powerha-aix/7.2?topic=security-configuring-message-authentication-encryption

Universidad de Valencia. (n.d.). 3.1 Algoritmos de cifrado simétrico. Recuperado de https://www.uv.es/sto/cursos/seguridad.java/html/sjava-12.html

Zendesk. (2024, mayo 29). *Encriptación de datos: ¿qué es y qué técnicas usa?*. Zendesk. https://www.zendesk.es/blog/encriptacion-datos/