Practica3SIDoubleRatchet: Hugo Freire Blanco (hugo.freire@udc.es)

Descripción

Este es el repositorio de la práctica 3, que consiste en implementar el protocolo de doble ratchet de Diffie-Hellman (Double Ratchet) en Python 3. La implementación actual permite que cualquier usuario pueda comenzar enviando mensajes y recalcula el ratchet de diffie-hellman para cada mensaje enviado (en lugar de enviar x mensajes con el mismo ratchet para tener mayor seguridad y simplicidad en el código).

Para ejecutar el programa tenemos que ejecutar ambas partes : el cliente y el servidor.

```
python3 client.py
python3 server.py
```

Commons.py (sección de funciones genéricas y variables de configuración)

Dentro de este fichero se encuentran algunos parámetros que son necesarios para establecer la conexión entre las dos partes y otras funciones relacionadas con el cifrado y descifrado de datos.

```
def safe_print(*args, **kwargs):
    """Thread-safe print function to avoid mixing output from different
threads"""
    with print_lock:
        print(*args, **kwargs)
```

Esta es una función que se utiliza para imprimir en pantalla de manera segura, evitando que los datos se mezclen entre los hilos. Se ha implementado esta función porque al utilizar threads diferentes para el envío y recepción de datos los prints no se muestran correctamente y desaparece el placeholder del input para meter el mensaje a enviar.

```
BUFFER_SIZE = 4096
```

Esta constante define el tamaño máximo de los buffers que se utilizarán para recibir y enviar datos. Se utiliza para evitar errores de buffer overflow.

```
COMUNICATION_PORT = 6027
DEFAULT_IP = "127.0.0.1"
```

Estas constantes definen el puerto y la dirección IP de la máquina en la que se ejecutará el servidor y el cliente. Se utilizan para establecer la conexión entre las dos partes.

Funciones genéricas de criptografía

```
def _generate_df_private_key():
    return X25519PrivateKey.generate()

def _generate_df_public_key(private_key : X25519PrivateKey):
    return private_key.public_key()
```

Estas funciones generan claves públicas y privadas para el algoritmo de Diffie-Hellman (Double Ratchet). Usamos una clave que implementa el algoritmo de X25519 (también conocido como Curve25519).

```
def serialize_public_key(public_key : X25519PublicKey):
    return public_key.public_bytes(
        encoding=serialization.Encoding.Raw,
        format=serialization.PublicFormat.Raw
    )

def deserialize_public_key(public_key_bytes : bytes) -> X25519PublicKey:
    return X25519PublicKey.from_public_bytes(public_key_bytes)
```

Estas funciones serializan y deserializan las claves públicas. Se utilizan para enviar y recibir claves públicas entre las dos partes.

```
def send_public_key(socket, public_key : X25519PublicKey):
    socket.send(serialize_public_key(public_key))

def receive_public_key(socket):
    key_bytes = socket.recv(32)
    return deserialize_public_key(key_bytes)
```

Estas funciones envían y reciben claves públicas entre las dos partes. Se utilizan en combinacion con las funciones de serialización y deresialización ya que la información enviada por el canal debe ser en forma de bytes y no en forma de string u objeto de Python.

```
def generate_df_key_pair():
    private_key = _generate_df_private_key()
    return private_key, _generate_df_public_key(private_key)

def obtain_shared_secret(private_key, public_key):
    return private_key.exchange(public_key)
```

Estas funciones son las que se utilizan para la implementación de Diffie-Hellman, donde la primera función genera un nuevo par de claves válidas y la segunda función coge la clave privada y la clave pública (que le llegaría desde el otro extremo) para obtener el secreto compartido.

Implementación de ratchet de Diffie-Hellman

```
def KDF_RK(rk: bytes, dh_out: bytes):
    """
    Deriva una nueva root key y una nueva chain key a partir del root key
actual y el DH output.
    """
    hkdf = HKDF(
        algorithm=hashes.SHA256(),
        length=32,         # 32 bytes para root key + 32 bytes para chain
key
        salt=rk,           # salt = root key actual
        info=b"DoubleRatchet_RK",  # aplicación específica
    )
    output = hkdf.derive(dh_out)
    new_root_key = output[:16]
    new_chain_key = output[16:]
    return new_root_key, new_chain_key
```

Esta función es la que implementa el ratchet de Diffie-Hellman. Se utiliza para generar la nueva root key y la nueva chain key a partir del root key actual y el DH output. La salida de esta función es una tupla con las claves nuevas. Utilizamos la funcion HKDF (HMAC-based Key Derivation Function) para derivar las claves.

La root key obtenida se utilizará como entrada para nuevas ejecuciones del ratchet, mientras que la chain_key es la entrada para el ratchet simetrico que se usa previo al cifrado y descifrado de los mensajes.

Implementación de ratchet simetrico

```
def KDF_CK(ck: bytes):
    """
    Ratchet simétrico de la chain key.
    Devuelve: message key y siguiente chain key.
    """
    # message key
    h = hmac.HMAC(ck, hashes.SHA256())
    h.update(b"\x01")
    mk = h.finalize()

    # next chain key
    h2 = hmac.HMAC(ck, hashes.SHA256())
    h2.update(b"\x02")
    next_ck = h2.finalize()

    return mk, next_ck
```

Esta función es la que implementa el ratchet simétrico. Se utiliza para obtener la clave de mensaje y la siguiente chain key a partir de la chain key actual. La salida de esta función es una tupla con las claves.

La message key se utiliza para cifrar y descifrar los mensajes, mientras que la siguiente chain key se utiliza para generar el siguiente ratchet simetrico (en caso de que haya varios dentro del mismo ratchet de diffie hellman según se haya configurado).

Implementación de cifrado y descifrado

```
def encrypt(mk : bytes, plaintext : bytes , asociated_data : bytes):
    nonce = os.urandom(12) # este es el tamaño recomendado para AESGCM
    cipher = AESGCM(mk)
    ciphertext = cipher.encrypt(nonce, plaintext,
associated_data=asociated_data)
    return nonce + ciphertext

def decrypt(mk: bytes, cyphertext: bytes, associated_data: bytes):
    nonce = cyphertext[:12]       # extraemos los primeros 12 bytes como
nonce
    ciphertext = cyphertext[12:]  # el resto es el ciphertext
    aesgcm = AESGCM(mk)
    plaintext = aesgcm.decrypt(nonce, ciphertext, associated_data)
    return plaintext
```

Estas funciones son las que se utilizan para cifrar y descifrar los mensajes. Se utilizan para cifrar y descifrar los mensajes enviados y recibidos. La salida de estas funciones es un bytes con el mensaje cifrado.

Internamente estas funciones utilizan el algoritmo AES-GCM (Galois/Counter Mode) para cifrar y descifrar los mensajes , utilizando para ello la clave obtenida con el ratchet simetrico a traves de la función KDF_CK.

Establecimiento de la conexión entre cliente y servidor

```
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
try:
    client.connect((DEFAULT_IP, COMUNICATION_PORT))
except ConnectionRefusedError:
    safe_print("❌ Error: no se pudo conectar con el servidor")
    exit(1)

safe_print("✓ Conectado al servidor\n")
safe_print("🔄 Intercambiando claves públicas iniciales...")
```

Esta es la parte de código que se utiliza para establecer la conexión entre el cliente y el servidor. Se utiliza un socket para establecer la conexión y se comprueba si se pudo conectar con el servidor. Si no se pudo conectar, se imprime un mensaje de error y se sale del programa.

Generación e intercambio inicial de claves.

```
private_key , public_key = generate_df_key_pair()
send_public_key(client, public_key)
safe_print("  ✓ Clave pública enviada")

received_public_key = receive_public_key(client)
safe_print("  ✓ Clave pública recibida\n")
```

Esta es la parte de código que se utiliza para generar y intercambiar claves públicas entre el cliente y el servidor. Se utiliza la función generate_df_key_pair para generar una nueva clave pública y una nueva clave privada. La clave pública se envía al otro extremo y a su vez se recibe la clave pública del otro extremo.

Esta lógica es necesaria para que así ambas partes contengan un secreto compartido que puedan utilizar para en envío del primer mensaje. Asi cualquiera puede enviar el mensaje en cualquier orden inicial y no hay un orden fijo en la comunicación.

Bucle principal del envío de mensajes.

El proceso de envío y recepción de mensajes es el mismo en el nodo de envío y en el de recepción y se implementa en el siguiente bloque de código:

```
while True:
    data = input("Enviar(exit para salir): ")
    if data.lower() == "exit":
        break

    if other_public_key is not None:
        private_key, public_key = generate_df_key_pair()
        secret = obtain_shared_secret(private_key, other_public_key)

    new_root_key , sending_chain_key = KDF_RK(new_root_key, secret)
    message_key , sending_chain_key = KDF_CK(sending_chain_key)

    ciphertext = encrypt(message_key, data.encode(), None)
    public_key_bytes = serialize_public_key(public_key)
    client.send(public_key_bytes + ciphertext)
    safe_print("[EMISOR] 📤 Enviado: " + data)
```

En primer lugar se comprueba que no sea la primera vez que se envia en mensaje para así recalcular de nuevo las claves de diffie hellman que se van a utilizar posteriormente. En cualquier caso se obtienen la clave junto con la información del otro extremo para obtener el secret y posteriormente calcular el ratchet de diffie hellman antes del envío de cada mensaje.

En caso de querer utilizar la misma chain_key para múltiples envíos simplemente se podría añadir un contador y recalcular dicho ratchet solamente cada x envíos.

El valor obtenido de la chain_key se utiliza como entrada para la generación del siguiente ratchet (ratchet simetrico) para así generar la clave de mensaje y la siguiente chain_key para el siguiente ratchet simetrico.

Esta clave obtenida (message_key) es la que se utiliza para el cifrado del mensaje y envío del array de bytes al otro extremo, todo ello acompañado de la clave pública del nodo actual. Es necesario incluir la clave pública para que así el receptor obtenga esa clave y compruebe si es igual a la que tiene almacenada para determinar si debe recalcular el ratchet de diffie hellman.

Una vez enviado el mensaje se finaliza la lógica y el nodo queda a la espera de enviar otro mensaje o recibir uno.

Recibir mensajes.

El proceso de recibir y decodificar mensajes es el mismo en el nodo de recepción y se implementa en el siguiente bloque de código:

```
while True:
    data = socket.recv(BUFFER_SIZE)
    if not data:
        break
    received_public_key_bytes = data[:32]
    received_ciphertext = data[32:]
    if serialize_public_key(other_public_key) == received_public_key_bytes:
        print("Las claves públicas son iguales")
        #usas eso con el ratchet simetrico para obtener la message key
        message_key , receiving_chain_key = KDF_CK(chain_key)
    else:
        print("Las claves públicas son diferentes")
        # generar nueva ratchet de diffie hellman
        received_public_key =
deserialize_public_key(received_public_key_bytes)
        secret = obtain_shared_secret(private_key, received_public_key)
        new_root_key , chain_key = KDF_RK(new_root_key, secret)
        message_key , receiving_chain_key = KDF_CK(chain_key)
    plaintext = decrypt(message_key, received_ciphertext, None)
    print("\nRecibido:", plaintext.decode() , end="\n")
```

Una vez que se ha recibido el array de bytes se extrae la clave pública del otro extremo (son los primeros 32 bytes) y se compara con la clave pública almacenada. Si las claves son iguales sabemos que se trata del mismo ratchet de diffie hellman así que simplemente calculamos el ratchet simetrico para obtener la clave de mensaje y la siguiente chain_key para el siguiente ratchet simetrico.

En caso de que las claves no sean iguales se genera un nuevo ratchet de diffie hellman y se utiliza la clave pública del otro extremo para obtener la clave de mensaje y la siguiente chain_key para el siguiente ratchet simetrico.

En ambos casos se obtiene una message_key que se utiliza para descifrar el mensaje que es igual al mensaje original que se envió al otro extremo.

Al recibir el mensaje se actualiza la clave recibida por el otro extremo para así utilizarla en el proceso de envío para la nueva generación del secreto a partir de dicha clave pública y mantener así los dos nodos sincronizados.

Este proceso de recepción también es igual en ambos casos.