BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA

FACULTAD DE CIENCIAS DE LA COMPUTACIÓN

NOMBRE DE LA DEPENDENCIA:

Facultad de Ciencias de la Computación

REPORTE

NOMBRE DEL ALUMNO:

Inés Yaelin Rojas Huerta

MATRICULA:

201732130

NOMBRE DEL PROGRAMA:

SISTEMA DE CRIPTOGRAFÍA DE CURVAS ELÍPTICAS PARA RESGUARDAR INFORMACIÓN

FOLIO:

208356

Contenido

Objetivo General

Desarrollar un sistema que implemente criptografía de curvas elípticas, de forma que garantice la confidencialidad y seguridad de la información.

Objetivos Específicos

- Implementar un módulo criptográfico que genere claves y realice su intercambio seguro por medio del protocolo Diffie-Hellman.
- Diseñar un servidor que actúe como intermediario para reenviar mensajes cifrados sin conocer su contenido.
- Desarrollar un cliente que gestione las claves, cifre-descifre mensajes y presente al usuario una interfaz gráfica para poder chatear.
- Mostrar en el servidor únicamente los mensajes cifrados en formato legible (base64), sin comprometer la privacidad de los mensajes.
- Garantizar que las claves privadas nunca abandonen al cliente y que la seguridad se mantenga aun si el servidor es comprometido.

Introducción.

El cifrado de mensajes es una práctica con más de 4000 años de antigüedad. Muestra de ello son los jeroglíficos del Antiguo Egipto que suelen tomarse como uno de los primeros ejemplos de “escritura oculta”.

Otro ejemplo lo encontramos en la Antigua Roma, su gobernante Julio Cesar utilizo un sistema simple para mantener secretos militares. Sustituyo cada letra del alfabeto con una letra tres posiciones más adelante. Mas tarde, cualquier cifrado que utilizara el concepto de “desplazamiento” para la creación de un alfabeto cifrado, se le denomino cifrado Cesar. Con el paso del tiempo el cifrado de mensajes se convirtió en un elemento clave en el mundo. Un ejemplo de ello lo encontramos en la Segunda Guerra Mundial. El matemático inglés descifro los códigos que la Alemania Nazi enviaba con su máquina Enigma. Gracias a ello se salvaron millones de vidas.

Después de la Segunda Guerra Mundial, la computación se convirtió en un instrumento clave dentro del cifrado y descifrado de mensajes, por seguridad la mayoría de los países consideraron la criptografía como algo secreto y vinculado a las tareas de inteligencia y espionaje. Incluso la NASA bloqueo cualquier tipo de publicación o investigación que tuviera que ver con la criptografía en Estados Unidos. No fue hasta el 17 de Marzo de 1975 que llego el primer avance publico vinculado a la criptografía cuando IBM desarrollo el algoritmo de cifrado DES (Data Encryption Standard).

En la actualidad, la protección de la información a través de redes públicas es un requisito esencial para garantizar la privacidad y la integridad de las comunicaciones. Entre los avances mas significativos de la criptografía moderna se encuentra la criptografía de curvas elípticas (ECC). Esta técnica se basa en las propiedades matemáticas de las curvas elípticas sobre campos finitos para construir sistemas criptográficos muy seguros y eficientes. A diferencia de RSA u otros algoritmos clásicos, ECC permite obtener el mismo nivel de seguridad con claves mucho mas pequeñas. Lo que reduce el consumo de recursos y la latencia de redes.

Diseño de la aplicación

El sistema presentado en este proyecto es un prototipo de mensajería segura sobre una arquitectura cliente- servidor. Cada cliente genera y gestiona de manera local su propio par de claves basado en criptografía de curvas elípticas (ECC), lo que permite establecer claves simétricas únicas para cada sesión mediante el protocolo de intercambio Diffie-Hellman de Curva Elíptica (ECDH). Una vez derivada la clave compartida, los mensajes se cifran utilizando AES (Advanced Encription Standard), que proporcionan simultáneamente confidencialidad de los datos.

El servidor actúa únicamente como intermediario que distribuye las claves publicas entre los clientes y reenvía los mensajes cifrados a sus destinatarios. De esa manera, el servidor no tiene acceso a los mensajes en texto claro NI a las claves privadas, preservando la privacidad incluso en caso de que el servidor sea comprometido.

Seguridad y protocolos implementados

Curvas elípticas (ECC)

La criptografía con curvas elípticas se destaca por poder operar con claves pequeñas, pero con un alto nivel de seguridad. Ya que en lugar de usar operaciones con números enteros enormes (como RSA), ECC usa puntos en una curva elíptica y operaciones algebraicas entre esos puntos.

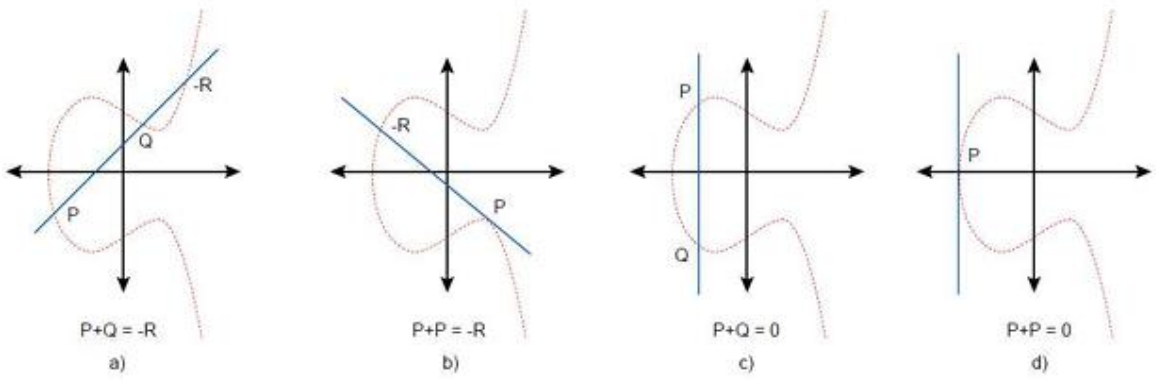
Una curva elíptica definida sobre un campo finito primario es el conjunto de soluciones (x, y) donde $x, y \in F_p$ de la ecuación de Weierstrass y su modulo p definen:

$$y^2 = x^3 + ax + b(mod\ p)$$

Las reglas de adición de puntos de una curva elíptica sobre un campo finito primo son:

- La adición del punto en el infinito más él mismo, tiene como resultado es el mismo punto en el infinito.
- La suma de un punto cualquiera más el punto en el infinito, tiene como resultado el mismo punto.
- La suma de dos puntos donde la coordenada "x" de ambos es igual y la coordenada "y" difiere o tiene el valor de cero (0), tiene como resultado el punto en el infinito.
- La suma de dos puntos con diferente coordenada "x". Este resultado da un punto con coordenadas diferentes a las coordenadas de los puntos sumados. Los puntos originales pertenecen a la curva.
- La suma de un punto por sí mismo, teniendo en cuenta que la coordenada "y" sea diferente de cero (0), tiene como resultado otro punto.

Los resultados de estas reglas de adición forman también un grupo. Este grupo es abeliano ya que cumple que $P1 + P2 = P2 + P1$, y estos puntos pertenecen a la curva.



1Suma de puntos en una curva elíptica

Los esquemas de criptografía basados en curva elíptica dependen de la multiplicación de un escalar por los puntos de la curva. Dado un entero y un punto de la curva, el proceso de multiplicación es la adición de ese punto tantas veces como lo indique el escalar. Esto es kP.

Curva elíptica Secp256r1

También conocida como P-256 o NIST P-256, es una curva elíptica estandarizada común, utilizada en el dominio criptográfico de firmas digitales e intercambio de claves, principalmente utilizada en la industria de blockchain.

Esta curva opera sobre un campo finito sobre números primos, proporcionando una seguridad de 256 bits, lo que significa que es resistente a ataques que intentan resolver el problema matemático del logaritmo discreto de la curva elíptica.
Su ecuación es:

$$y^2 = x^3 - 3x + b \ (mod\ p)$$

Con parámetros:

p	0xffffffff00000001000000000000000000000000ffffffffffffffffffffffff
a	0xffffffff00000001000000000000000000000000fffffffffffffffffffffffc
b	0x5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b

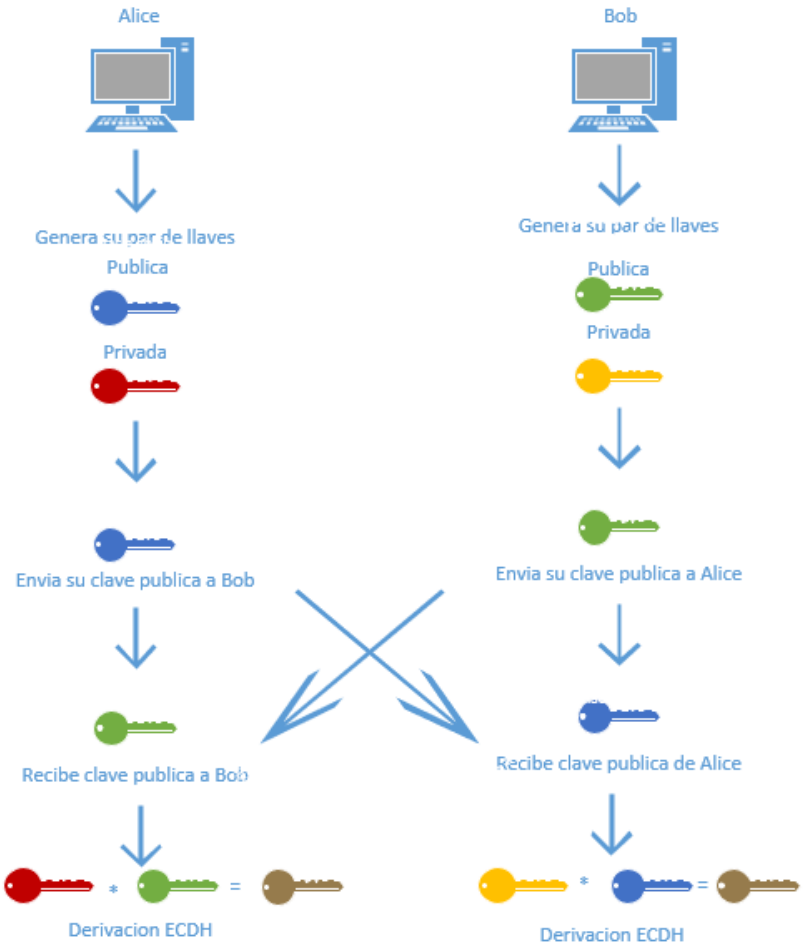
G	(0x6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296, 0x4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5)
n	0xffffffff00000000ffffffffffffffffbce6faada7179e84f3b9cac2fc632551
h	0x1

ECDH (Elliptic Curve Diffie-Hellman)

Ejemplo básico de ECDH

- Alice tiene clave privada a y clave pública $A = aG$
- Bob tiene clave privada b y clave pública $B = bG$
- Ambos usan el mismo punto base G de la curva
- Alice calcula $S = aB$, Bob calcula $S = bA$

  Ambos obtienen el mismo punto $S = abG$ que usan para derivar una clave simétrica AES.



AES-GCM(Advanced Encryption Standard en modo Galois/Counter Mode)

AES (Advanced Encryption Standard) es un algoritmo de cifrado simétrico que cifra bloques de 128 bits usando claves de 128, 192 o 256 bits.
Por sí solo, AES cifra un bloque fijo; para mensajes largos necesitas un modo de operación.

GCM es un modo de operación para AES que combina dos cosas:

- CTR (Counter Mode): convierte AES en un cifrador de flujo seguro.
    - Se genera un contador (nonce + contador) que se cifra con AES para producir una secuencia de claves ("keystream") que se XOR con el mensaje → así se cifra y descifra cualquier longitud.
- GHASH (Galois Hash): calcula un código de autenticación (MAC) sobre el texto cifrado y datos asociados para garantizar integridad y autenticidad.

En este proyecto se utiliza un sistema con curvas elípticas, ECC (ECDH) se usa para acordar una clave simétrica entre dos usuarios.

Esa clave simétrica se pasa a AES-GCM para cifrar y autenticar los mensajes reales que se envían por el servidor. Así se obtiene un cifrado de extremo a extremo.

Arquitectura general

La arquitectura se compone de tres módulos principales:

- **client.py**: gestiona la conexión del usuario, envía y recibe mensajes.
- **server.py**: actúa como puente, distribuyendo los mensajes cifrados entre los clientes.
- **crypto_utils.py**: contiene las funciones criptográficas para la generación de claves y el cifrado/descifrado.

A continuación, se hará una descripción de los módulos más importantes en el código, el cogido completo se encuentra en mi perfil de GitHub: https://github.com/IYRH/Cifrado-con-Curvas-El-pticas-

Cliente

Dirección y puerto a donde conectarse.

```
HOST = 'localhost'
PORT = 65432
```

Generación de claves y estructuras de datos

- **private.key y public.key** generan par de llaves ECC.
- **peer_public_keys**: almacenará las claves públicas de otros usuarios.
- **shared_keys**: almacena las claves simétricas derivadas con cada usuario (ECDH).

```
        self.private_key, self.public_key = generate_keypair()
        self.peer_public_keys = {}
        self.shared_keys = {}
```

Conexión al servidor

Crea un socket TCP (Protocolo de Control de Transmisión) y se conecta al servidor. Después, envía un paquete [USER | nombre | clave_publica_serializada] para registrarse.

```
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.connect((HOST, PORT))
        self.sock.sendall(b"USER|" + self.username.encode() + b"|" +
serialize_public_key(self.public_key))
```

Hilo de recepción

Lanza un hilo aparte que estará constantemente leyendo mensajes del servidor.

```
        threading.Thread(target=self.receive_loop, daemon=True).start()
```

Recepción de claves públicas

Cuando llega un paquete [USER | usuario | clave_publica], actualiza las estructuras:

- Guarda la clave pública del nuevo usuario.
- Deriva inmediatamente la clave simétrica compartida con él (ECDH).

```
    if data.startswith(b'USER|'):
                    parts = data.split(b'|', 2)
                    username = parts[1].decode()
                    pubkey_bytes = parts[2]
                    pubkey = deserialize_public_key(pubkey_bytes)
                    if username != self.username and username not in
self.peer_public_keys:
                        self.peer_public_keys[username] = pubkey
                        self.shared_keys[username] =
derive_shared_key(self.private_key, pubkey)
                        self.display_message(f"*** Clave pública recibida de
{username}", sent=False)
```

Recepción de mensajes cifrados

Cuando llega el paquete [MSG | emisor | destino | bloque_cifrado]:

- Comprueba el destinatario.
- Usa la clave simétrica que deriva de from_user para descifrar.

```
                elif data.startswith(b'MSG|'):
                    parts = data.split(b'|', 3)
                    from_user = parts[1].decode()
                    to_user = parts[2].decode()
                    encrypted_msg = parts[3]
                    if to_user == self.username and from_user in
self.shared_keys:
                        msg = decrypt_message(self.shared_keys[from_user],
encrypted_msg)
                        self.display_message(msg, sent=False)
```

Envío de mensajes

Se toma el mensaje, para cada usuario que ya tiene clave simétrica, cifra el mensaje con **encrypt_message**(shared_key,…). Crea el paquete [MSG | emisor | destino | mensaje_cifrado] y lo envia al servidor.

```
    def send_message(self, event=None):
        msg = self.entry.get()
        if not msg:
            return

        for username, shared_key in self.shared_keys.items():
            encrypted = encrypt_message(shared_key, f"{self.username}:
{msg}")
            payload = b"MSG|" + self.username.encode() + b"|" +
username.encode() + b"|" + encrypted
            self.sock.sendall(payload)

        self.display_message(f"{self.username}: {msg}", sent=True)
        self.entry.delete(0, tk.END)
```

Servidor

Las funciones principales del servidor son:

- Recibir las conexiones de los clientes.
- Registrar nombre de usuario y clave pública de cada cliente.
- Reenviar las claves públicas y los mensajes cifrados.
- Muestra en consola la versión **encriptada** de los mensajes (en base64).

El servidor no tiene acceso a las claves privadas, **ni descifra** mensajes en ningún momento.

Transmisión de Mensajes

Reenvía el paquete de bytes a los demás clientes, excepto al que lo envió.

```
def broadcast_message(sender_conn, data):
    with lock:
```

```
        for conn, addr in clients:
            if conn != sender_conn:
                try:
                    conn.sendall(data)
                except Exception as e:
                    print(f"¡Error enviando a {addr}: {e} !")
```

Manejo de Cliente

Se crea un hilo para cada cliente conectado, lo primero que hace es recibir el paquete [USER | usuario | clave_publica] y separarlo en variables username y pubkey, si el paquete no tiene ese formato, entonces aborta.

```
def handle_client(conn, addr):
    try:
        user_data = conn.recv(4096)
        if not user_data.startswith(b"USER|"):
            print(f"x Formato inválido de {addr}")
            return

        _, username, pubkey = user_data.split(b"|", 2)
```

Registrar al cliente

Guarda el nombre del cliente y su clave publica

```
        with lock:
            clients.append((conn, addr))
            public_keys[addr] = (username, pubkey)
```

Enviar claves existentes al nuevo cliente

El servidor envía al nuevo cliente las claves públicas del usuario que ya estaba conectado para que el nuevo cliente pueda derivar claves compartidas.

```
        with lock:
            for other_addr, (other_user, other_key) in public_keys.items():
                if other_addr != addr:
                    conn.sendall(b"USER|" + other_user + b"|" + other_key)
```

Recepción de mensajes

En esta parte, se reciben los mensajes del cliente en este formato [MSG | emisor | destino|bloque_cifrado], extrae el bloque_cifrado y lo imprime en consola en base64. Al final llama a broadcast_message para reenviarlo al cliente receptor.

```
        while True:
            data = conn.recv(4096)
            if not data:
                break
            try:
                if data.startswith(b"MSG|"):
                    parts = data.split(b'|', 3)
                    if len(parts) == 4:
                        from_user = parts[1].decode(errors='replace')
                        encrypted = parts[3]  # bytes (iv|tag|ciphertext)
                        # Mostrar versión printable (base64) del mensaje
encriptado
                        print(f"{from_user}:
{base64.b64encode(encrypted).decode()}")
            except Exception as e:
                print(f"x Error al registrar mensaje encriptado de {addr}:
{e}")

            broadcast_message(conn, data)
```

No descifra el mensaje, solo lo muestra en base64 para que se vea legible en consola.

Manejo de cierre y limpieza

Si hay un error o desconexión, se elimina al cliente de las listas y se cierra el socket.

```
    except Exception as e:
        print(f"x Error con {addr}: {e}")
    finally:
        with lock:
            clients[:] = [c for c in clients if c[0] != conn]
            public_keys.pop(addr, None)
        conn.close()
        print(f"@ Cliente {addr} desconectado.")
```

Iniciar servidor

Se crea un socket TCP, lo asocial a HOST:PORT y empieza a escuchar conexiones. Cada vez que llega un cliente (acept()), lanza un hilo handle_client para gestionarlo.

```
def start_server():
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind((HOST, PORT))
        s.listen()
        print(f"--> Servidor escuchando en {HOST}:{PORT}")
        while True:
            conn, addr = s.accept()
            threading.Thread(target=handle_client, args=(conn, addr),
daemon=True).start()

start_server()
```

Módulo Criptográfico

El archivo crypto_utils.py contiene utilidades criptográficas que implementa un esquema de cifrado usando curvas elípticas para cifrar y descifrar mensajes.

Generación de claves

Se genera un par de claves (pública y privada) usando curvas elípticas en el estándar **SECP256R1**.

La clave privada (k) se elige al azar sobre la curva **SECP256R1**. , mientras que la clave publica (P) se calcula $P = k * G$, donde G es un punto base estándar.

```
def generate_keypair():
    private_key = ec.generate_private_key(ec.SECP256R1())
    public_key = private_key.public_key()
    return private_key, public_key
```

Serialización y deserialización

Convierte la clave publica a formato PEM (Privacy Enchance Mail), para poder enviarla a otro usuario como texto.

```
def serialize_public_key(public_key):
    """Convierte una clave pública a formato PEM"""
    return public_key.public_bytes(
```

```
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )
```

Hace lo inverso: recibe una clave en PEM y la reconstruye como objeto usable.

```
def deserialize_public_key(pem_data):
    """recibe datos en PEM y devuelve un objeto clave pública usable."""
    return serialization.load_pem_public_key(pem_data)
```

Derivación de clave compartida

La siguiente función aplica ECDH.

```
shared_key = private_key.exchange(ec.ECDH(), peer_public_key)
```

No se obtiene texto legible ni una clave que se pueda usar directamente, sino una cadena de bytes aleatoria que solo los usuarios que estén chateado entre sí pueden calcular, porque el usuario Alice usa su clave privada más la clave publica de Bob. Mientras que Bob usa su clave privada más la clave publica de Alice. Al resultado lo llamaremos secreto compartido, es secreto porque nadie más lo puede derivar, aunque se conozcan las claves públicas.

```
def derive_shared_key(private_key, peer_public_key):
    """Usa ECDH conviertir en una clave simétrica de 32 bytes (256 bits) con
HKDF (función derivadora de clave con SHA-256)."""
    shared_key = private_key.exchange(ec.ECDH(), peer_public_key)
    return HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'handshake data',
    ).derive(shared_key)
```

Después, con HKDF (Función de derivación de claves basado en HMAC) transforma ese secreto compartido en una clave simétrica de 256 bits para usar con AES

(Estándar de Cifrado Avanzado con una clave de 256 bits). Esta clave ahora la pueden usar ambos extremos para cifrar y descifrar mensajes.

Cifrado de mensajes

Cifra el mensaje usando AES-GCM. Ahora que ambos tienen su clave derivada, se puede usar esa clave para cifrar y descifrar mensajes. Pero cada mensaje se cifra con un numero único y aleatorio de 12 bytes (en el código lo nombramos iv)

```
def encrypt_message(key, message):
    """Cifra mensaje usando AES CON GCM, usa un vector de inicializacion
aleatorio de 12 bytes"""
    iv = os.urandom(12)
    cipher = Cipher(algorithms.AES(key), modes.GCM(iv))
    encryptor = cipher.encryptor()
    ciphertext = encryptor.update(message.encode()) + encryptor.finalize()
    return iv + encryptor.tag + ciphertext
```

La función devuelve un paquete con

```
    iv (12 bytes) + tag (16 bytes) + ciphertext (resto)
```

Descifrado

El usuario que recibe ese paquete puede descifrarlo con la misma clave secreta. Extrae IV, etiqueta y datos cifrados para después devolver el texto original.

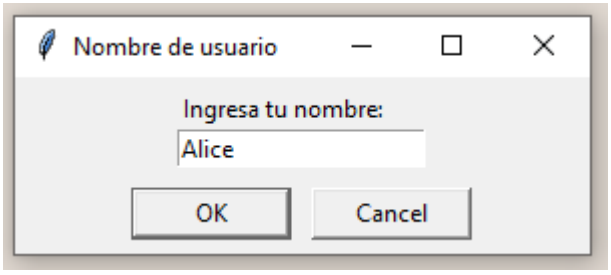
```
def decrypt_message(key, data):
    """Hace lo inverso al anterior funcion, extrae IV, tag y ciphertex con
AES-GCM"""
    iv = data[:12]
    tag = data[12:28]
    ciphertext = data[28:]
    cipher = Cipher(algorithms.AES(key), modes.GCM(iv, tag))
    decryptor = cipher.decryptor()
    plaintext = decryptor.update(ciphertext) + decryptor.finalize()
    return plaintext.decode()
```

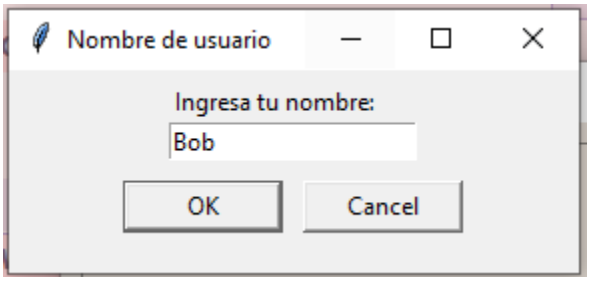
Pruebas de sistema

Al encender el servidor queda en espera de clientes

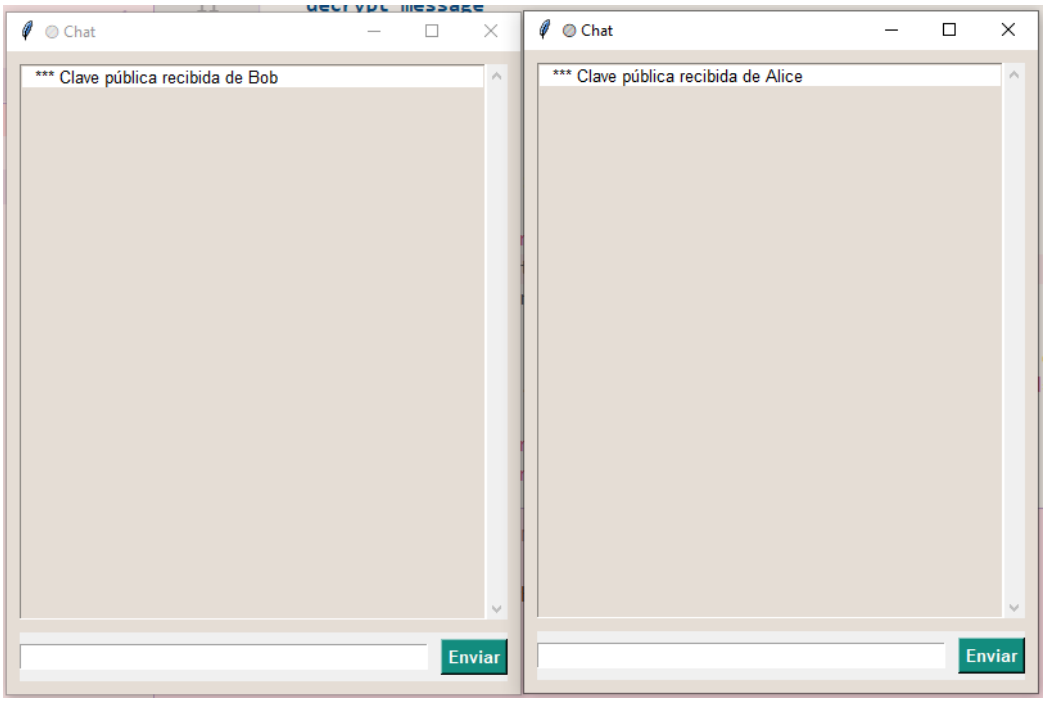
```
--> Servidor escuchando en localhost:65432
```

Al iniciar el cliente, arranca la interfaz gráfica que pide el nombre de usuario, en caso de no ingresar un nombre se quedara como “Anonimo”. Creamos los usuarios Alice y Bob para que interactúen en el chat.

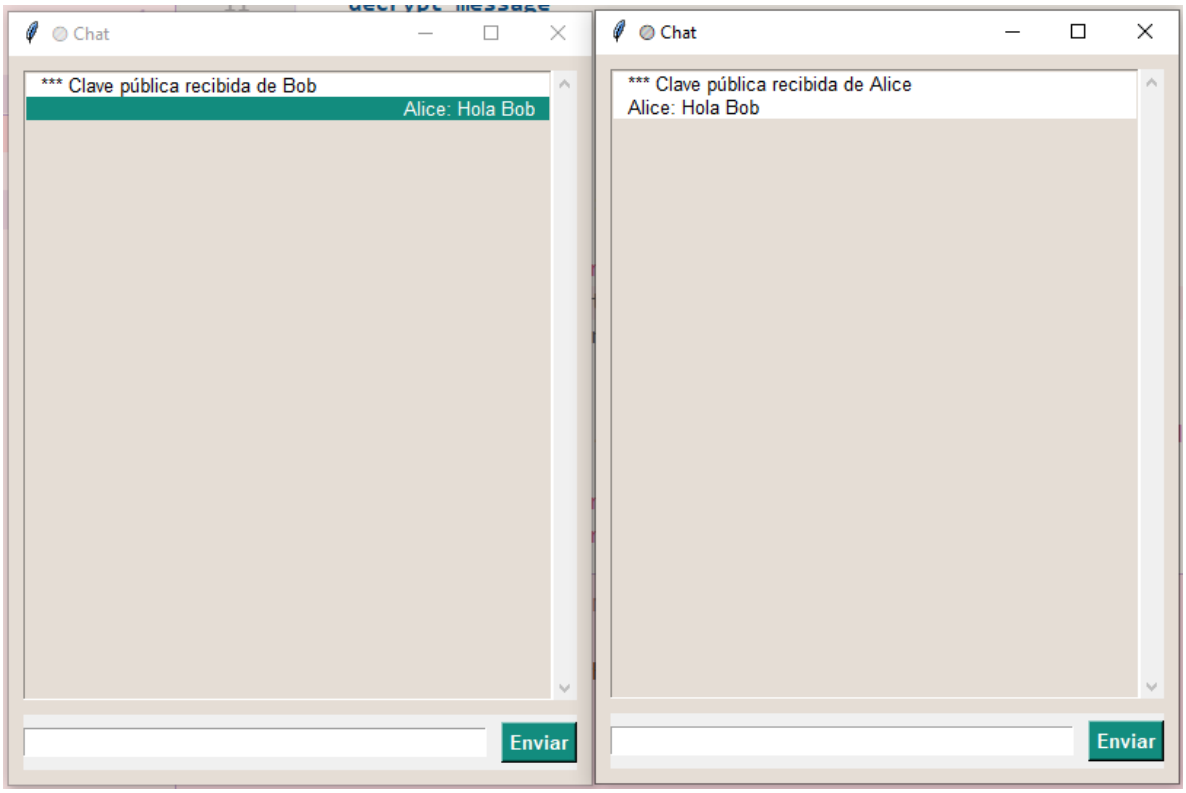




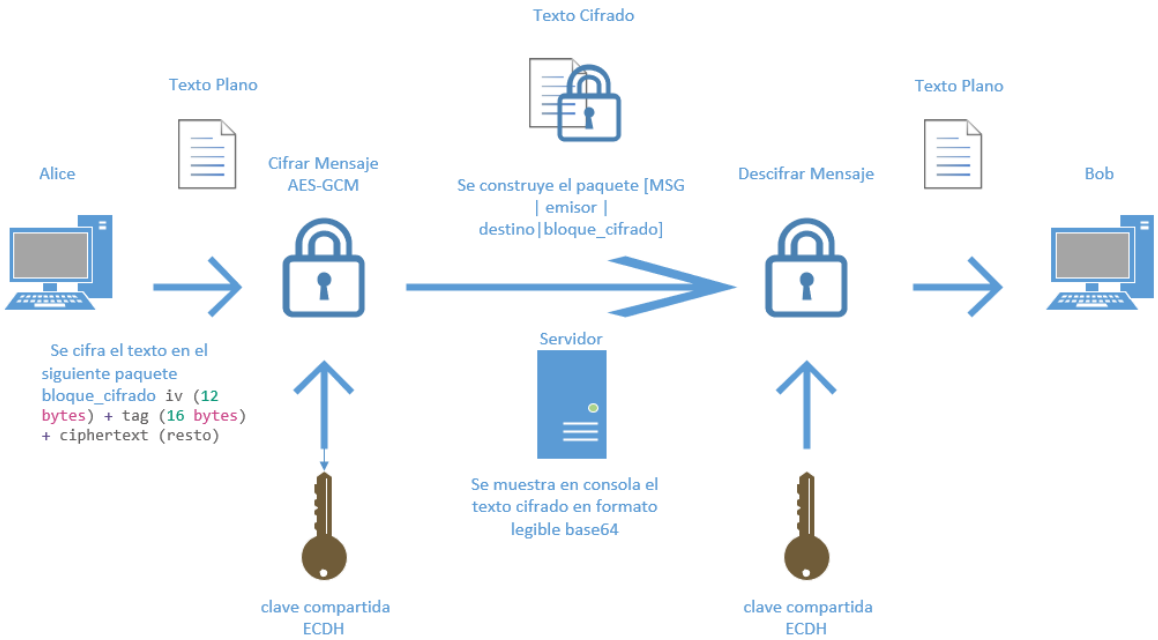
Ambos usuarios generan su par de claves ECC. Se conectan al servidor, se crea el paquete [USER | usuario | clave_publica] y se envía al servidor para registrarse. La clave privada se queda en secreto, pero la clave publica se comparte entre sí.



Ya con la clave pública del otro usuario se deriva la clave compartida ECDH. Ahora los usuarios están listos para intercambiar mensajes seguros.



A continuación, se muestra un diagrama con el flujo de comunicación.



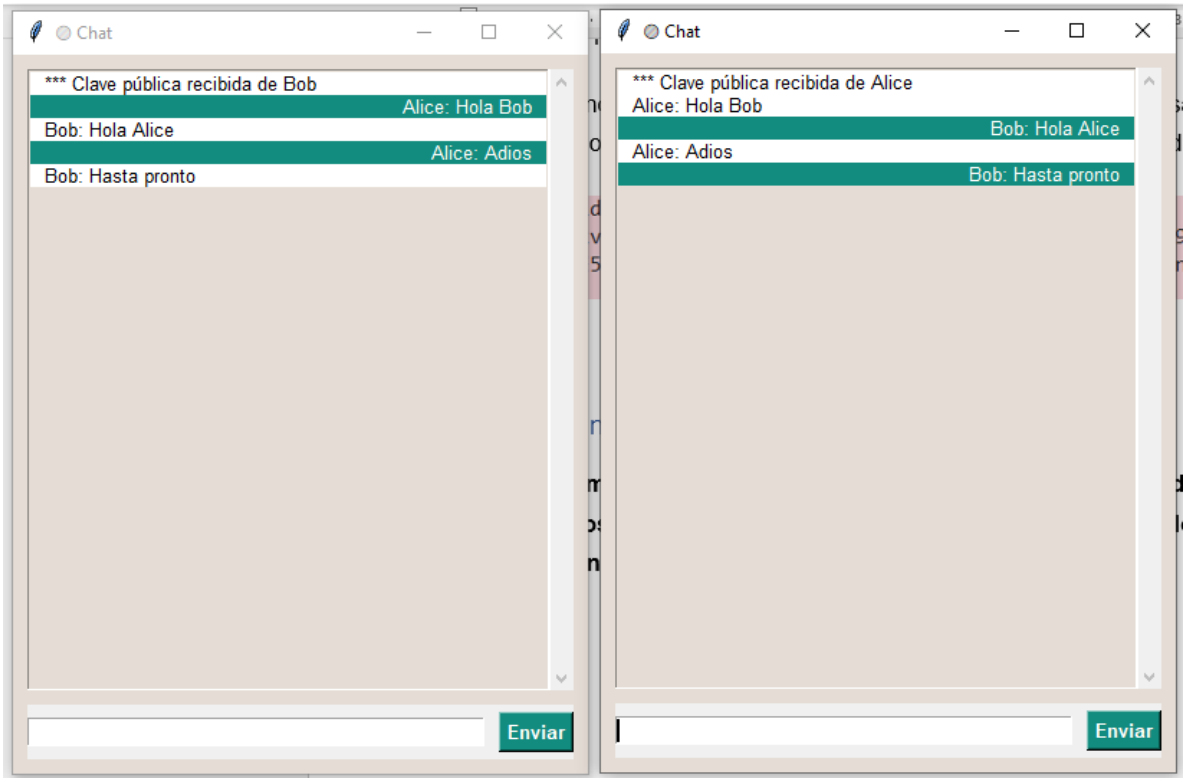
El cliente receptor recibe el paquete y descifra el bloque con su llave compartida ECDH para mostrar el mensaje en claro.

El servidor no es capaz de cifrar ni descifrar mensajes, recibe el mensaje ya cifrado y se puede observar en la consola como texto legible base64.

```
--> Servidor escuchando en localhost:65432
Alice: czvead9+KhisvAXXTMkwFcPcRIQtgMn5CXax3Jq+r2UAuQQSv799Bl0w8Q==
Bob: 9hPE5cOhQrgYxsU7jqpwi0fwTmxT/FzooiVyGeFLu3Sl9lTMl8qHIn9f8A==
```

Al salir uno de los clientes, el servidor limpia sus estructuras, elimina al cliente de la lista clients (para que no le intenten enviar más mensajes). Elimina su cklave publica de public_keys. Cierra el socket e imprime en consola:

```
--> Servidor escuchando en localhost:65432
Alice: czvead9+KhisvAXXTMkwFcPcRIQtgMn5CXax3Jq+r2UAuQQSv799Bl0w8Q==
Bob: 9hPE5cOhQrgYxsU7jqpwi0fwTmxT/FzooiVyGeFLu3Sl9lTMl8qHIn9f8A==
Alice: 8eiF250SvtGnBKCku9JEk9i7TbK91UZNiLWMn2g9fIyYeuypigxbsw==
Bob: ZaUPxj/qhedF5IEfcklm6LG+21T93afshqkkVxIz2ZheJUMNgGVz+Q9tOrIM
x Error con ('127.0.0.1', 57139): [WinError 10054] Se ha forzado la interrupción de una conexión existente por el host remoto
@ Cliente ('127.0.0.1', 57139) desconectado.
x Error con ('127.0.0.1', 57142): [WinError 10054] Se ha forzado la interrupción de una conexión existente por el host remoto
@ Cliente ('127.0.0.1', 57142) desconectado.
```



Conclusión

Este proyecto implementa un sistema de tipo chat básico pero seguro gracias al uso de protocolos y algoritmos de seguridad como: criptografía con curvas elípticas para generar aleatoriamente claves publicas y privadas, protocolo ECDH para derivar una clave compartida a partir del combo de llaves públicas y privadas y cifrado simétrico con AES-GCM para garantizar la seguridad y la autenticidad de los datos compartidos a través de una vía no confiable.

El servidor cuenta con una mínima seguridad y se limita a dos cosas: el servidor solo recibe bytes ya cifrados, nunca recibe llaves privadas. Además, cuenta con bloqueo de concurrencia con threading.Lock para evitar que dos hilos modifiquen al mismo tiempo las listas clients y public_keys, pero esto solo protege contra errores de concurrencia no contra ataques.

Sin embargo, este sistema no depende de lo confiable que sea el servidor, sino de mecanismos criptográficos robustos, ya que el servidor muestra los mensajes cifrados (en formato base64) sin exponer el contenido real.