

SI - Práctica TOR

Samuel Vázquez Fernández
A Coruña, 13 de Octubre

Configuración Inicial	2
1. Instalar dependencias necesarias	2
2. Copiar plantilla de configuración y editar con tus datos	2
Variables de configuración:	2
3. Configurar claves RSA	2
Uso del Sistema	3
Componentes Principales	3
1. Listener (Receptor)	3
2. Sender (Emisor)	3
Nodos disponibles en la red	3
Estructura de archivos en el repositorio	3
Funciones clave del sistema	4
tor.py	4
Funciones para cargar el par de claves (Privada y Pública)	4
Formatear (padding) de user-id	4
Buscar clave pública de un usuario	5
Funciones de cifrado y descifrado (AES-GCM y RSA)	5
Cifrado y descifrado híbrido	6
Cifrado anidado	7
Recibir mensaje	8
mqtt-listener.py	8
Función get_next_hop()	8
Procesamiento de Mensajes (on_message())	9
Configuración e inicialización del Cliente MQTT	10
mqtt-sender.py	11
Pruebas	12
Envíos	12
Recibir	12
Recibido con una cadena estando de destino final y siendo anónimo	12
Recibiendo de un compañero en modo anónimo	13
Recibiendo de un compañero sin ser anónimo	13
Recibiendo con una cadena muy larga y siendo destinatario final	13

Configuración Inicial

1. Instalar dependencias necesarias

```
pip install paho-mqtt cryptography
```

2. Copiar plantilla de configuración y editar con tus datos

```
cp config_template.py config.py
```

Variables de configuración:

BROKER_HOST: Dirección del servidor MQTT

BROKER_USER y BROKER_PWD: Credenciales de autenticación (usuario y contraseña)

BROKER_PORT: Puerto de conexión (1883)

KEEPALIVE: Tiempo de keep-alive para mantener la conexión (60 por defecto en la configuración)

MY_ID: Identificador único del nodo en la red TOR

3. Configurar claves RSA

Nos aseguramos de tener en el directorio el par de claves.

En el código el sistema carga automáticamente las claves RSA, importante cambiar el nombre ya que si nos fijamos en la imagen, yo lo tengo hecho para que busque **id_rsaSlsamu**.

```
try:
    # Importante cambiar el nombre a como tengas guardada tu clave privada
    with open("id_rsaSlsamu", "rb") as kf:
        private_key = serialization.load_pem_private_key(
            kf.read(),
            password=None,
            backend=default_backend()
        )
except FileNotFoundError:
    private_key = None
    logging.warning("Private key file not found.")

try:
    # Importante cambiar el nombre a como tengas guardada tu clave pública
    with open("id_rsaSlsamu.pub", "rb") as kf:
        public_key = serialization.load_ssh_public_key(
            kf.read(),
            backend=default_backend()
        )
except FileNotFoundError:
    public_key = None
    logging.warning("Public key file not found.")
```

Uso del Sistema

Componentes Principales

El sistema se compone de dos aplicaciones complementarias:

1. Listener (Receptor)

```
python mqtt_listener.py
```

Al ejecutar el listener, el nodo se convierte en un punto activo de la red TOR, permaneciendo a la escucha de mensajes cifrados. Cuando recibe un mensaje, analiza su estructura para determinar si debe reenviarlo al siguiente salto de la ruta o si ha llegado a su destino final, momento en el cual descifra el contenido y lo presenta al usuario.

2. Sender (Emisor)

```
python mqtt_sender.py
```

El sender básicamente se utiliza para enviar mensajes a los nodos, desde este archivo puedes cambiar el path con la ruta de nodos que quieres que siga hasta el destino final, y aparte también el contenido del mensaje a enviar.

Nodos disponibles en la red

Los nodos están definidos en el archivo **tor.py** en el diccionario `pubkey_dictionary`.

Algunos ejemplos:

svf, ancr, svr, etc.

Estructura de archivos en el repositorio

```
tor_network/
├── tor.py                # Librería criptográfica
├── mqtt_listener.py      # Receptor de mensajes
├── mqtt_sender.py        # Emisor
├── config_template.py    # Plantilla de configuración
├── config.py             # Configuración real (.gitignore)
├── id_rsaISamu           # Clave privada (.gitignore)
├── id_rsaISamu.pub       # Clave pública (.gitignore)
└── README.md            # Este archivo
```

Funciones clave del sistema

Empezamos con el módulo principal, en este caso el archivo **tor.py**.

Todo el código está subido en el siguiente repositorio:

[SI-Munics/Lab1-RedTOR at main · samuelvazfez/SI-Munics](#)

tor.py

Las funciones más importantes son:

Funciones para cargar el par de claves (Privada y Pública)

```
try:
    # Importante cambiar el nombre a como tengas guardada tu clave privada
    with open("id_rsaSIsamu", "rb") as kf:

        private_key = serialization.load_pem_private_key(
            kf.read(),
            password=None,
            backend=default_backend()
        )
except FileNotFoundError:
    private_key = None
    logging.warning("Private key file not found.")

try:
    # Importante cambiar el nombre a como tengas guardada tu clave pública
    with open("id_rsaSIsamu.pub", "rb") as kf:

        public_key = serialization.load_ssh_public_key(
            kf.read(),
            backend=default_backend()
        )
except FileNotFoundError:
    public_key = None
    logging.warning("Public key file not found.")
```

Restart

Formatear (padding) de user-id

```
def pad_userid(uId: bytes) -> bytes:
    """Rellenar a la derecha con '\x00' hasta 5. Trunca si es mayor."""
    if not isinstance(uId, (bytes, bytearray)):
        raise TypeError("pad_userid espera bytes")
    if len(uId) >= 5:
        return uId[:5]
    return uId + b"\x00" * (5 - len(uId))

def unpad_userid(b: bytes) -> bytes:
    """Quita los ceros finales añadidos por pad_userid."""
    return b.rstrip(b"\x00")
```

Estas funciones ajustan el user-id a una longitud exacta de 5 bytes, ya sea rellenándolo con bytes nulos o truncándolo, y revierten el proceso eliminando dicho relleno.

Buscar clave pública de un usuario

```
def find_public_key_by_id(uid: str):
    if uid is None:
        return None
    p = pubkey_dictionary.get(uid)
    if not p:
        return None
    try:
        return serialization.load_ssh_public_key(('ssh-rsa ' + p).encode('ascii'), backend=default_backend())
    except Exception as e:
        logging.exception("Error cargando clave pública %s: %s", uid, e)
        return None
```

Esta función se utiliza para encontrar la clave pública asignada a un usuario con un ID específico, esto itera sobre la lista de claves públicas llamada "pubkey-dictionary".

Funciones de cifrado y descifrado (AES-GCM y RSA)

```
def encrypt_aesgcm(key, plaintext):
    aesgcm = AESGCM(key)
    nonce = key[:16] # para que sea iv = k -> nonce = key[:16]
    ciphertext = aesgcm.encrypt(nonce, plaintext, None)
    return ciphertext

# Decrypt el ciphertext con la key usando AESGCM cipher

def decrypt_aesgcm(key, ciphertext):
    aesgcm = AESGCM(key)
    nonce = key[:16]
    plaintext = aesgcm.decrypt(nonce, ciphertext, None)
    return plaintext
```

Estas funciones cifran y descifran datos utilizando el algoritmo AES-GCM, empleando los primeros 16 bytes de la propia clave como nonce (vector de inicialización). Actualmente, hacerlo de esta manera (nonce = key[:16]) es inseguro porque usa una parte de la clave como nonce, en lugar de generar uno aleatorio para cada mensaje. Esto es un parche temporal que debe corregirse.

```

def rsa_encrypt(pub_key, plaintext: bytes) -> bytes:
    return pub_key.encrypt(
        plaintext,
        padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()),
                    algorithm=hashes.SHA256(), label=None)
    )

# Decrypt el ciphertext con la private_key usando RSA cipher
def rsa_decrypt(ciphertext: bytes) -> bytes:
    if private_key is None:
        raise RuntimeError("Clave privada no cargada")
    return private_key.decrypt(
        ciphertext,
        padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()),
                    algorithm=hashes.SHA256(), label=None)
    )

```

Estas funciones implementan el cifrado y descifrado asimétrico RSA, usando una clave pública para cifrar y una privada para descifrar con el relleno seguro OAEP.

Cifrado y descifrado híbrido

Estas funciones implementan un cifrado híbrido, combinando la seguridad del cifrado asimétrico (RSA) con la velocidad del cifrado simétrico (AES-GCM) para cifrar datos de forma eficiente y segura.

```

def encrypt_hybrid(pub_key, plaintext: bytes) -> bytes:
    """
    Genera k (16 bytes), cifra plaintext con AES-GCM (iv=k),
    cifra k con RSA-OAEP y devuelve ek || ct || tag.
    """
    k = AESGCM.generate_key(bit_length=128)
    c2 = encrypt_aesgcm(k, plaintext)
    c1 = rsa_encrypt(pub_key, k)
    return c1 + c2

# Decrypt el plaintext usando cifrado híbrido, primero RSA para la clave AESGCM y luego AESGCM para los datos.
def decrypt_hybrid(ciphertext: bytes) -> bytes:
    """
    Separa el bloque RSA (keylen) y el resto (ct||tag), recupera k y descifra.
    Devuelve plaintext bytes o lanza excepciones si falla.
    """
    if private_key is None:
        raise RuntimeError("Clave privada no cargada")
    key_block_len = private_key.key_size // 8
    logging.debug("decrypt_hybrid: key_length=%d total_cipher_len=%d", key_block_len, len(ciphertext))
    if len(ciphertext) <= key_block_len + AES_TAG_LEN:
        raise ValueError("Ciphertext demasiado corto")
    ek = ciphertext[:key_block_len]
    ct_tag = ciphertext[key_block_len:]
    logging.debug("len ek=%d len ct_tag=%d", len(ek), len(ct_tag))
    k = rsa_decrypt(ek)
    pt = decrypt_aesgcm(k, ct_tag)
    return pt

```

Cifrado anidado

```
def encrypt_nested_hybrid(path: List[str], message: bytes, sender_id: str = "none") -> bytes:
    if not path:
        raise ValueError("path debe contener al menos el destinatario final")

    if isinstance(message, str):
        message = message.encode('utf-8')
    sender_b = sender_id.encode('ascii') if isinstance(sender_id, str) else sender_id
    # capa terminal: pad('end') || pad(sender) || message
    terminal_payload = pad_userid(b"end") + pad_userid(sender_b) + message
    # cifrar para el destinatario final (path[-1])
    recipient = path[-1]
    recipient_pub = find_public_key_by_id(recipient)
    if recipient_pub is None:
        raise KeyError(f"No encontré clave pública de '{recipient}'")
    c = encrypt_hybrid(recipient_pub, terminal_payload)
    # envolver desde n-1 hasta 0 (inclusive)
    # en cada paso: payload = pad(next_hop) || inner; Luego Encrypt(pk_i, payload)
    for idx in range(len(path) - 2, -1, -1):
        pk_i = find_public_key_by_id(path[idx])
        if pk_i is None:
            raise KeyError(f"No encontré clave pública de '{path[idx]}'")
        next_hop = path[idx + 1]
        next_b = pad_userid(next_hop.encode('ascii')) if isinstance(next_hop, str) else next_hop
        inner = next_b + c
        c = encrypt_hybrid(pk_i, inner)
    return c
```

Esta función, realiza un cifrado anidado o en capas, similar al que se usa en redes como Tor (Onion Routing). Crea una especie de "muñeca rusa" criptográfica donde cada capa solo puede ser abierta por un nodo específico de la ruta.

Los parámetros que se le pasan son:

- **path: List[str]** -> es la ruta que seguirá el mensaje, una lista con los IDs de los nodos.
- **messages: bytes** -> es el mensaje original que se quiere enviar.
- **sender_id: str** -> es el identificador de quién envía el mensaje, si se quiere ser anónimo, este parámetro será none, por ahora se cambia aquí de forma manual, pero se debería hacer diferente en un futuro, por ejemplo pasándolo como un booleano desde un menú interactivo que todavía no está creado, que sea para enviar.

Que hace esta función:

La función construye el mensaje cifrado desde dentro hacia fuera, empezando por el destinatario final y terminando en el primer nodo de la red.

Recibir mensaje

```
def receive_message_debug(ciphertext: bytes):
    try:
        decrypted = decrypt_hybrid(ciphertext)
    except Exception as e:
        logging.exception("Fallo al descifrar: %s", e)
        return
    if len(decrypted) < 5:
        logging.error("Descifrado demasiado corto (%d bytes)", len(decrypted))
        return
    next_hop = unpad_userid(decrypted[:5]).decode('ascii', errors='ignore')
    inner = decrypted[5:]
    if next_hop.lower() == "end":
        if len(inner) < 5:
            logging.error("Inner demasiado corto para contener sender")
            return
        sender = unpad_userid(inner[:5]).decode('ascii', errors='ignore')
        message_bytes = inner[5:]
        try:
            message_text = message_bytes.decode('utf-8')
        except Exception:
            message_text = message_bytes.decode('utf-8', errors='replace')
        print(f"[DEBUG] Mensaje final - De: {sender} - Contenido: {message_text}")
    else:
        print(f"[DEBUG] Reenvío - next_hop: {next_hop} - inner_bytes_len: {len(inner)}")
```

Esta función simula cómo un nodo de la red procesa un mensaje cifrado que recibe, decidiendo si es el destinatario final o si debe reenviarlo.

mqtt-listener.py

Función get_next_hop()

```
def get_next_hop(payload: bytes) -> str:
    # Devolvemos el next-hop decodificado (5 bytes, padded con \x00 a la derecha)
    if len(payload) < 5:
        return ""
    hop_bytes = payload[:5]
    return hop_bytes.rstrip(b'\x00').decode('ascii', errors='ignore')
```

Extrae y decodifica el siguiente destino de la ruta desde los primeros 5 bytes del payload, eliminando el padding de ceros.

Procesamiento de Mensajes (on_message())

```
def on_message(client, userdata, msg):
    logging.debug("Mensaje recibido en topic %s (len=%d)", msg.topic, len(msg.payload))
    try:
        decrypted = tor.decrypt_hybrid(msg.payload)
    except Exception as e:
        logging.exception("Fallo al descifrar la capa; descartando mensaje")

    if len(decrypted) < 5:
        logging.error("Payload descifrado demasiado corto: %d bytes", len(decrypted))
        return

    next_hop = get_next_hop(decrypted) # string, p.ej. 'svf' o 'end'
    inner = decrypted[5:]

    if next_hop.lower() == "end":
        # destinatario final: inner = sender(5 bytes) || message bytes
        if len(inner) < 5:
            logging.error("Inner demasiado corto para contener sender + message")
            return
        sender = inner[:5].rstrip(b'\x00').decode('ascii', errors='ignore')
        message_bytes = inner[5:]
        # decode seguro en utf-8
        message_text = message_bytes.decode('utf-8', errors='replace')

        logging.info("Mensaje final recibido. De: %s - Contenido: %s", sender, message_text)
    else:
        # reenvío al siguiente hop: publicamos EXACTAMENTE los bytes 'inner'
        try:
            client.publish(next_hop, payload=inner, qos=1)
            logging.info("🔄 Reenviado %d bytes a %s", len(inner), next_hop)
        except Exception:
            logging.exception("Error al reenviar a %s", next_hop)
```

Implementa la lógica de reenvío anidado.

Configuración e inicialización del Cliente MQTT

```
def mqtt_client(server, port, topic, user, password, keepalive):
    client = mqtt.Client()
    client.on_message = on_message
    client.username_pw_set(user, password)
    client.connect(server, port, keepalive)
    client.subscribe(topic)
    return client

if __name__ == "__main__":
    print("🚀 Iniciando Nodo TOR MQTT...")
    print(f"🔗 Conectando a {BROKER_HOST} como '{MY_ID}'")


    client = mqtt_client(BROKER_HOST, BROKER_PORT, MY_ID, BROKER_USER, BROKER_PWD, KEEPALIVE)

    # Iniciar loop MQTT en un hilo separado
    client.loop_start()

    # Banner de inicio
    print("✅ Conectado al broker MQTT")
    print("👂 Escuchando mensajes...")

    # ! Implementar a futuro
    # print("💡 Escribe 'help' para ver comandos disponibles")
    # y el menú con las funcionalidades

    try:
        menu_loop(client)
    finally:
        print("🛑 Desconectando del broker...")
        client.loop_stop()
        client.disconnect()
```

 Restart Vis

Establecemos la conexión con el broker MQTT.

mqtt-sender.py

```
def mqtt_client(server, port, topic, user, password, keepalive):
    client = mqtt.Client()
    client.username_pw_set(user, password)
    client.connect(server, port, keepalive)
    client.subscribe(topic)
    return client

# Creamos el cliente MQTT
if __name__ == "__main__":

    client = mqtt_client(BROKER_HOST, BROKER_PORT, MY_ID, BROKER_USER, BROKER_PWD, KEEPALIVE)

    # Ejemplo de mensaje y ruta (ajusta según necesites)

    cipher_message = b"Prueba para el trabajo siendo anonimo"

    path = ["svf", "acu"]

    encrypted_to_send = tor.encrypt_nested_hybrid(path, cipher_message)

    # Iniciar loop para que el hilo de red de paho procese la publicación
    client.loop_start()

    print("SENDER: len:", len(encrypted_to_send))
    print("SENDER: prefix hex:", encrypted_to_send[:64].hex())

    publish_result = client.publish(path[0], payload=encrypted_to_send, qos=1)

    publish_result.wait_for_publish()
    time.sleep(0.1)
    client.loop_stop()
    client.disconnect()

    print(f"Mensaje cifrado publicado en el topic '{path[0]}')"
```

Establece conexión MQTT similar al listener pero sin callback de mensajes entrantes.

Proceso de envío:

Cifrado anidado: Aplica `encrypt_nested_hybrid()` con la ruta especificada

Publicación: Envía el mensaje cifrado al primer nodo de la ruta (`path[0]`)

Confirmación: Espera confirmación de entrega al broker MQTT

Cierre ordenado: Libera recursos de conexión.

Pruebas

Envios

Así se ve cuando envío a un compañero. En este caso a svr.

```
C:\Users\USUARIO\munics\SI\SI-Munics\Lab1-RedTOR\mqtt_sender.py:8: DeprecationWarning: Callback API version 1 is deprecated, update to latest version
  client = mqtt.Client()
SENDER: len: 316
SENDER: prefix hex: 2be016c1170e1958e605da43601b843385362cca8d915f4e4465144e0f0d9472cf539567b4aae59296e8cf053b53fc22509eb4975f95ae92107729692bd713e4
Mensaje cifrado publicado en el topic 'svr'
PS C:\Users\USUARIO\munics\SI\SI-Munics\Lab1-RedTOR>
```

Recibir

Recibido con una cadena estando de destino final y siendo anónimo

```
PS C:\Users\USUARIO\munics\SI\SI-Munics\Lab1-RedTOR> py .\mqtt_listener.py
🚀 Iniciando Nodo TOR MQTT...
🐳 Conectando a 18.101.140.151 como 'svf'
C:\Users\USUARIO\munics\SI\SI-Munics\Lab1-RedTOR\mqtt_listener.py:166: DeprecationWarning: Callback API version 1 is deprecated, update to latest version
  client = mqtt.Client()
✅ Conectado al broker MQTT
🎧 Escuchando mensajes...

► Mensajes:
2025-10-09 23:08:19,652 INFO 🔄 Reenviado 1001 bytes a svr
2025-10-09 23:08:19,844 INFO Mensaje final recibido. De: none -
Contenido: Prueba para el trabajo siendo anonimo
```

Recibiendo de un compañero en modo anónimo

```
PS C:\Users\USUARIO\munics\SI\SI-Munics\Lab1-RedTOR> py .\mqtt_listener.py
🚀 Iniciando Nodo TOR MQTT...
🌐 Conectando a 18.101.140.151 como 'svf'
C:\Users\USUARIO\munics\SI\SI-Munics\Lab1-RedTOR\mqtt_listener.py:166: DeprecationWarning: Callback API version 1 is deprecated, update to latest version
  client = mqtt.Client()
✅ Conectado al broker MQTT
🎧 Escuchando mensajes...

► Mensajes:
2025-10-09 23:11:10,341 INFO 🔄 Reenviado 732 bytes a ancr
2025-10-09 23:11:10,459 INFO Mensaje final recibido. De: none -
Contenido: Esta es una prueba anónima desde el nodo svr
```

Recibiendo de un compañero sin ser anónimo

```
► Mensajes:
2025-10-09 23:09:53,142 INFO 🔄 Reenviado 723 bytes a ancr
2025-10-09 23:09:53,254 INFO Mensaje final recibido. De: svr -
Contenido: Esta es una prueba desde el nodo svr
```

Recibiendo con una cadena muy larga y siendo destinatario final

```
PS C:\Users\USUARIO\munics\SI\SI-Munics\Lab1-RedTOR> py .\mqtt_listener.py
🚀 Iniciando Nodo TOR MQTT...
🌐 Conectando a 18.101.140.151 como 'svf'
C:\Users\USUARIO\munics\SI\SI-Munics\Lab1-RedTOR\mqtt_listener.py:166: DeprecationWarning: Callback API version 1 is deprecated, update to latest version
  client = mqtt.Client()
✅ Conectado al broker MQTT
🎧 Escuchando mensajes...

► Mensajes:
2025-10-09 23:14:18,250 INFO 🔄 Reenviado 4283 bytes a svr
2025-10-09 23:14:18,452 INFO 🔄 Reenviado 3324 bytes a ancr
2025-10-09 23:14:18,565 INFO 🔄 Reenviado 2642 bytes a svr
2025-10-09 23:14:18,792 INFO 🔄 Reenviado 1683 bytes a ancr
2025-10-09 23:14:18,897 INFO 🔄 Reenviado 1001 bytes a svr
2025-10-09 23:14:19,099 INFO Mensaje final recibido. De: none -
Contenido: Prueba para el trabajo siendo anonimo
```