**COSC3796\_Assignment1\_HarmanPreetKaurGrewal**

**server.py**

#!/usr/bin/env python3

import threading

import socket

import json

import struct

import argparse

import base64

from datetime import datetime

# Utility functions to send/recv length-prefixed JSON over sockets

def send\_json(conn, obj):

data = json.dumps(obj).encode()

conn.sendall(struct.pack(">I", len(data)) + data)

def recv\_json(conn):

raw\_len = recvall(conn, 4)

if not raw\_len:

return None

length = struct.unpack(">I", raw\_len)[0]

data = recvall(conn, length)

if not data:

return None

return json.loads(data.decode())

def recvall(conn, n):

data = b""

while len(data) < n:

packet = conn.recv(n - len(data))

if not packet:

return None

data += packet

return data

class Server:

def \_\_init\_\_(self, host, port):

self.addr = (host, port)

self.sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

self.sock.bind(self.addr)

self.sock.listen(10)

print(f"[SERVER] Listening on {host}:{port}")

self.clients = {} # username -> (conn, addr)

self.pubkeys = {} # username -> public\_key\_pem (str)

self.lock = threading.Lock()

self.server\_log\_file = "server\_log.json"

self.ensure\_server\_log()

def ensure\_server\_log(self):

try:

with open(self.server\_log\_file, "r") as f:

pass

except FileNotFoundError:

with open(self.server\_log\_file, "w") as f:

json.dump([], f, indent=2)

def append\_server\_log(self, entry):

with self.lock:

with open(self.server\_log\_file, "r+") as f:

data = json.load(f)

data.append(entry)

f.seek(0)

json.dump(data, f, indent=2)

f.truncate()

def client\_thread(self, conn, addr):

try:

# first handshake: client sends {"type":"register","name":..., "pubkey":...}

msg = recv\_json(conn)

if not msg:

conn.close()

return

if msg.get("type") != "register":

send\_json(conn, {"type":"error","error":"First message must be register"})

conn.close()

return

name = msg.get("name")

pubkey = msg.get("pubkey")

if not name or not pubkey:

send\_json(conn, {"type":"error","error":"register requires name and pubkey"})

conn.close()

return

with self.lock:

self.clients[name] = (conn, addr)

self.pubkeys[name] = pubkey

print(f"[SERVER] Registered user '{name}' from {addr}")

send\_json(conn, {"type":"registered","ok":True})

# Accept messages from this client

while True:

msg = recv\_json(conn)

if not msg:

break

# Basic expected message shapes:

# 1) {"type":"request\_pubkey","target":"Bob"}

# 2) {"type":"session\_key","from":"Alice","to":"Bob","payload": base64 bytes}

# 3) {"type":"message","from":"Alice","to":"Bob","ciphertext":base64, "nonce":base64, "tag":base64}

mtype = msg.get("type")

if mtype == "request\_pubkey":

target = msg.get("target")

pk = self.pubkeys.get(target)

if pk:

send\_json(conn, {"type":"pubkey\_resp","target":target,"pubkey":pk})

else:

send\_json(conn, {"type":"error","error":"no such user"})

elif mtype in ("session\_key","message"):

# Log incoming encrypted data (we store the raw JSON in server\_log)

entry = {

"timestamp": datetime.utcnow().isoformat() + "Z",

"type": mtype,

"from": msg.get("from"),

"to": msg.get("to"),

"payload": msg.get("payload") if mtype=="session\_key" else {

"ciphertext": msg.get("ciphertext"),

"nonce": msg.get("nonce"),

"tag": msg.get("tag")

}

}

print(f"[SERVER] Received {mtype} from {entry['from']} -> {entry['to']}")

self.append\_server\_log(entry)

# Forward to recipient if connected

recipient = msg.get("to")

with self.lock:

rec = self.clients.get(recipient)

if rec:

rec\_conn, \_ = rec

try:

send\_json(rec\_conn, msg)

except Exception as e:

print(f"[SERVER] Error forwarding to {recipient}: {e}")

else:

print(f"[SERVER] Recipient {recipient} not connected. (message queued not implemented)")

elif mtype == "list\_users":

with self.lock:

users = list(self.pubkeys.keys())

send\_json(conn, {"type":"users\_list","users":users})

else:

send\_json(conn, {"type":"error","error":"unknown message type"})

except Exception as e:

print(f"[SERVER] Exception for client {addr}: {e}")

finally:

# cleanup

# find which username had this conn and remove

with self.lock:

found = None

for name,(c,a) in list(self.clients.items()):

if c==conn:

found = name

del self.clients[name]

del self.pubkeys[name]

break

if found:

print(f"[SERVER] {found} disconnected.")

conn.close()

def serve\_forever(self):

try:

while True:

conn, addr = self.sock.accept()

t = threading.Thread(target=self.client\_thread, args=(conn, addr), daemon=True)

t.start()

except KeyboardInterrupt:

print("\n[SERVER] Shutting down.")

self.sock.close()

if \_\_name\_\_ == "\_\_main\_\_":

parser = argparse.ArgumentParser()

parser.add\_argument("--host", default="127.0.0.1")

parser.add\_argument("--port", type=int, default=9000)

args = parser.parse\_args()

srv = Server(args.host, args.port)

srv.serve\_forever()

**client.py**

#!/usr/bin/env python3

import socket

import argparse

import struct

import json

import threading

import os

import base64

from cryptography.hazmat.primitives import serialization, hashes

from cryptography.hazmat.primitives.asymmetric import rsa, padding

from cryptography.hazmat.primitives.ciphers.aead import AESGCM

from datetime import datetime

# Utility functions for length-prefixed JSON

def send\_json(conn, obj):

data = json.dumps(obj).encode()

conn.sendall(struct.pack(">I", len(data)) + data)

def recv\_json(conn):

raw\_len = recvall(conn, 4)

if not raw\_len:

return None

length = struct.unpack(">I", raw\_len)[0]

data = recvall(conn, length)

if not data:

return None

return json.loads(data.decode())

def recvall(conn, n):

data = b""

while len(data) < n:

packet = conn.recv(n - len(data))

if not packet:

return None

data += packet

return data

# File helpers

def save\_pem\_private(key, filename):

pem = key.private\_bytes(

encoding=serialization.Encoding.PEM,

format=serialization.PrivateFormat.TraditionalOpenSSL,

encryption\_algorithm=serialization.NoEncryption()

)

with open(filename, "wb") as f:

f.write(pem)

def load\_pem\_private(filename):

with open(filename, "rb") as f:

return serialization.load\_pem\_private\_key(f.read(), password=None)

def save\_pem\_public(key, filename):

pem = key.public\_bytes(

encoding=serialization.Encoding.PEM,

format=serialization.PublicFormat.SubjectPublicKeyInfo

)

with open(filename, "wb") as f:

f.write(pem)

def load\_pem\_public\_data(pem\_bytes):

return serialization.load\_pem\_public\_key(pem\_bytes)

class Client:

def \_\_init\_\_(self, name, server\_host, server\_port):

self.name = name

self.server\_addr = (server\_host, server\_port)

self.sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

self.private\_key\_file = f"private\_{name}.pem"

self.public\_key\_file = f"public\_{name}.pem"

self.messages\_file = f"messages\_{name}.json"

self.ensure\_messages\_file()

self.session\_keys = {} # peername -> bytes (AES key)

self.read\_lock = threading.Lock()

self.generate\_or\_load\_keys()

self.connect\_and\_register()

# background listener

t = threading.Thread(target=self.listen\_loop, daemon=True)

t.start()

def ensure\_messages\_file(self):

if not os.path.exists(self.messages\_file):

with open(self.messages\_file, "w") as f:

json.dump([], f, indent=2)

def append\_message\_log(self, entry):

with self.read\_lock:

with open(self.messages\_file, "r+") as f:

data = json.load(f)

data.append(entry)

f.seek(0)

json.dump(data, f, indent=2)

f.truncate()

def generate\_or\_load\_keys(self):

if os.path.exists(self.private\_key\_file):

self.priv = load\_pem\_private(self.private\_key\_file)

with open(self.public\_key\_file, "rb") as f:

self.pub\_pem = f.read()

else:

# generate RSA 2048

self.priv = rsa.generate\_private\_key(public\_exponent=65537, key\_size=2048)

self.pub\_pem = self.priv.public\_key().public\_bytes(

encoding=serialization.Encoding.PEM,

format=serialization.PublicFormat.SubjectPublicKeyInfo

)

save\_pem\_private(self.priv, self.private\_key\_file)

save\_pem\_public(self.priv.public\_key(), self.public\_key\_file)

print(f"[{self.name}] Generated RSA keypair and saved to {self.private\_key\_file} / {self.public\_key\_file}")

def connect\_and\_register(self):

self.sock.connect(self.server\_addr)

# send registration

send\_json(self.sock, {"type":"register", "name": self.name, "pubkey": self.pub\_pem.decode()})

resp = recv\_json(self.sock)

if resp and resp.get("type")=="registered":

print(f"[{self.name}] Registered with server.")

else:

print(f"[{self.name}] Registration failed: {resp}")

raise SystemExit(1)

def listen\_loop(self):

try:

while True:

msg = recv\_json(self.sock)

if not msg:

print(f"[{self.name}] Server closed connection.")

break

mtype = msg.get("type")

if mtype == "pubkey\_resp":

# rare: client requested pubkey and got response

print(f"[{self.name}] Received pubkey for {msg.get('target')}.")

elif mtype == "session\_key":

# payload: base64 of RSA-encrypted session key

b64 = msg.get("payload")

enc = base64.b64decode(b64)

# decrypt with own private key

session\_key = self.priv.decrypt(

enc,

padding.OAEP(

mgf=padding.MGF1(algorithm=hashes.SHA256()),

algorithm=hashes.SHA256(),

label=None

)

)

from\_user = msg.get("from")

self.session\_keys[from\_user] = session\_key

print(f"\n[{self.name}] Received SESSION KEY from {from\_user}. (stored)")

# log this event in messages file

self.append\_message\_log({

"timestamp": datetime.utcnow().isoformat()+"Z",

"direction": "received\_session",

"from": from\_user,

"session\_key\_b64": base64.b64encode(session\_key).decode()

})

elif mtype == "message":

# Forwarded ciphertext

ciphertext\_b64 = msg.get("ciphertext")

nonce\_b64 = msg.get("nonce")

tag\_b64 = msg.get("tag")

from\_user = msg.get("from")

print(f"\n[{self.name}] Encrypted message received from {from\_user}:")

print(f" ciphertext (base64): {ciphertext\_b64}")

# server requirement: immediately show ciphertext then plaintext

# Attempt to decrypt using stored session key for this sender

session\_key = self.session\_keys.get(from\_user)

if not session\_key:

print(f" No session key for {from\_user}. Cannot decrypt.")

# still log ciphertext

self.append\_message\_log({

"timestamp": datetime.utcnow().isoformat()+"Z",

"direction": "received",

"from": from\_user,

"ciphertext": ciphertext\_b64,

"decrypted": None

})

continue

try:

aesgcm = AESGCM(session\_key)

nonce = base64.b64decode(nonce\_b64)

ciphertext = base64.b64decode(ciphertext\_b64)

# AESGCM expects ciphertext that already includes tag; but we sent tag separately.

# We'll reconstruct: AESGCM expects ciphertext + tag combined as 'data'

tag = base64.b64decode(tag\_b64)

ct\_plus\_tag = ciphertext + tag

plaintext = aesgcm.decrypt(nonce, ct\_plus\_tag, None)

plaintext\_str = plaintext.decode()

print(f" plaintext: {plaintext\_str}\n")

# log

self.append\_message\_log({

"timestamp": datetime.utcnow().isoformat()+"Z",

"direction": "received",

"from": from\_user,

"ciphertext": ciphertext\_b64,

"nonce": nonce\_b64,

"tag": tag\_b64,

"decrypted": plaintext\_str

})

except Exception as e:

print(f" Decryption failed: {e}")

self.append\_message\_log({

"timestamp": datetime.utcnow().isoformat()+"Z",

"direction": "received",

"from": from\_user,

"ciphertext": ciphertext\_b64,

"decrypted": None,

"error": str(e)

})

elif mtype == "users\_list":

users = msg.get("users",[])

print(f"[{self.name}] Users on server: {users}")

elif mtype == "error":

print(f"[{self.name}] Server error: {msg.get('error')}")

else:

print(f"[{self.name}] Unknown msg from server: {msg}")

except Exception as e:

print(f"[{self.name}] listener exception: {e}")

# helper to request pubkey of someone

def request\_pubkey(self, target):

send\_json(self.sock, {"type":"request\_pubkey","target":target})

resp = recv\_json(self.sock)

if resp and resp.get("type")=="pubkey\_resp" and resp.get("target")==target:

pk\_pem = resp.get("pubkey").encode()

return pk\_pem

else:

print(f"[{self.name}] error getting pubkey: {resp}")

return None

# create session key for 'target' and send encrypted session key to them

def create\_and\_send\_session\_key(self, target):

# ask server for public key (server returns immediately or in another message)

# We'll directly send request and wait for response

send\_json(self.sock, {"type":"request\_pubkey","target":target})

resp = recv\_json(self.sock)

if not resp or resp.get("type")!="pubkey\_resp":

print(f"[{self.name}] Could not get public key for {target}")

return False

target\_pub\_pem = resp.get("pubkey").encode()

target\_pub = serialization.load\_pem\_public\_key(target\_pub\_pem)

# generate AES-256 key

from os import urandom

session\_key = urandom(32) # AES-256

# encrypt session key with recipient's public RSA key

enc = target\_pub.encrypt(

session\_key,

padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()), algorithm=hashes.SHA256(), label=None)

)

b64 = base64.b64encode(enc).decode()

# store locally

self.session\_keys[target] = session\_key

# send to server with type session\_key

send\_json(self.sock, {"type":"session\_key","from":self.name,"to":target,"payload":b64})

print(f"[{self.name}] Sent encrypted session key to {target}.")

# log locally

self.append\_message\_log({

"timestamp": datetime.utcnow().isoformat()+"Z",

"direction":"sent\_session",

"to":target,

"session\_key\_b64": base64.b64encode(session\_key).decode()

})

return True

def send\_message\_to(self, target, text):

# ensure we have a session key for target

session\_key = self.session\_keys.get(target)

if not session\_key:

print(f"[{self.name}] No session key for {target}. Create session first using '/session {target}'")

return

aesgcm = AESGCM(session\_key)

nonce = os.urandom(12)

plaintext = text.encode()

ct\_plus\_tag = aesgcm.encrypt(nonce, plaintext, None)

# split tag: AESGCM returns ciphertext||tag; tag is last 16 bytes for GCM

tag = ct\_plus\_tag[-16:]

ciphertext = ct\_plus\_tag[:-16]

# base64 encode each

ct\_b64 = base64.b64encode(ciphertext).decode()

nonce\_b64 = base64.b64encode(nonce).decode()

tag\_b64 = base64.b64encode(tag).decode()

# send to server

send\_json(self.sock, {"type":"message","from":self.name,"to":target,"ciphertext":ct\_b64,"nonce":nonce\_b64,"tag":tag\_b64})

print(f"[{self.name}] Sent encrypted message to {target}.")

# log locally

self.append\_message\_log({

"timestamp": datetime.utcnow().isoformat()+"Z",

"direction":"sent",

"to":target,

"ciphertext":ct\_b64,

"nonce":nonce\_b64,

"tag":tag\_b64,

"decrypted": text

})

def interactive\_loop(self):

print(f"Hello {self.name}! Type commands or '@username message' to send.")

print("Commands:")

print(" /session <username> # create and send AES session key to username")

print(" /users # list users on server")

print(" /quit # exit")

while True:

try:

line = input(f"[{self.name}]> ").strip()

if not line:

continue

if line.startswith("/session"):

parts = line.split()

if len(parts) != 2:

print("Usage: /session <username>")

continue

target = parts[1]

self.create\_and\_send\_session\_key(target)

elif line == "/users":

send\_json(self.sock, {"type":"list\_users"})

# the server will reply asynchronously

elif line == "/quit":

print("Quitting.")

self.sock.close()

break

elif line.startswith("@"):

# @Bob Hello Bob

try:

target, msg = line[1:].split(" ", 1)

except ValueError:

print("Usage: @username your message")

continue

self.send\_message\_to(target, msg)

else:

print("Unknown command. Use /session, /users, /quit, or @username message.")

except KeyboardInterrupt:

print("\nKeyboard interrupt. Exiting.")

break

if \_\_name\_\_ == "\_\_main\_\_":

import os

parser = argparse.ArgumentParser()

parser.add\_argument("--name", required=True)

parser.add\_argument("--server", default="127.0.0.1")

parser.add\_argument("--port", type=int, default=9000)

args = parser.parse\_args()

client = Client(args.name, args.server, args.port)

client.interactive\_loop()

**README.md**

# COSC3796 Assignment 1 – End-to-End Encrypted Chat Application

\*\*Author:\*\* Harman Preet Kaur Grewal

\*\*Course:\*\* COSC3796 – Cryptography & Network Security

---

## Objective

This project demonstrates end-to-end encryption (E2EE) in a simple chat system where two users (Alice and Bob) securely exchange messages through a central server.

## Requirements

- Python 3.8+

- cryptography library