1. Crypto\_driver.c

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/fs.h>

#include <linux/cdev.h>

#include <linux/uaccess.h>

#include <linux/crypto.h>

#include <crypto/skcipher.h>

#include <crypto/hash.h>

#include <linux/scatterlist.h>

#include <linux/ioctl.h>

#define DEVICE\_NAME "crypto\_driver"

#define BUFFER\_SIZE 1024

#define SHA1\_DIGEST\_SIZE 20 // Define SHA1 digest size

#define IOCTL\_ENCODE \_IOW('k', 1, struct ioctl\_data)

#define IOCTL\_DECODE \_IOW('k', 2, struct ioctl\_data)

#define IOCTL\_HASH \_IOW('k', 3, struct ioctl\_data)

struct ioctl\_data {

void \_\_user \*input;

void \_\_user \*output;

size\_t len;

void \_\_user \*iv;

};

static struct cdev cdev;

static dev\_t device\_number;

static struct class \*class = NULL;

static int device\_open(struct inode \*inode, struct file \*file) {

printk(KERN\_INFO "crypto\_driver: Device opened\n");

return 0;

}

static int device\_release(struct inode \*inode, struct file \*file) {

printk(KERN\_INFO "crypto\_driver: Device closed\n");

return 0;

}

static long device\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg) {

struct ioctl\_data data;

void \_\_user \*input, \*output, \*iv;

size\_t len;

int ret = -ENOMEM;

char \*input\_buf = NULL, \*output\_buf = NULL;

printk(KERN\_INFO "crypto\_driver: IOCTL cmd=%u (hex: %x), arg=%lx\n", cmd, cmd, arg);

printk(KERN\_INFO "crypto\_driver: Expected IOCTL\_ENCODE=%u (hex: %x), IOCTL\_DECODE=%u (hex: %x), IOCTL\_HASH=%u (hex: %x)\n",

(unsigned int)IOCTL\_ENCODE, (unsigned int)IOCTL\_ENCODE,

(unsigned int)IOCTL\_DECODE, (unsigned int)IOCTL\_DECODE,

(unsigned int)IOCTL\_HASH, (unsigned int)IOCTL\_HASH);

printk(KERN\_INFO "crypto\_driver: Size of ioctl\_data=%zu\n", sizeof(struct ioctl\_data));

if (copy\_from\_user(&data, (void \_\_user \*)arg, sizeof(struct ioctl\_data))) {

printk(KERN\_ERR "crypto\_driver: Failed to copy ioctl\_data from user\n");

return -EFAULT;

}

input = data.input;

output = data.output;

len = data.len;

iv = data.iv;

printk(KERN\_INFO "crypto\_driver: Received len=%zu, input=%p, output=%p, iv=%p\n",

len, input, output, iv);

if (len > BUFFER\_SIZE || len == 0) {

printk(KERN\_ERR "crypto\_driver: Invalid length: %zu\n", len);

return -EINVAL;

}

if (!access\_ok(input, len)) {

printk(KERN\_ERR "crypto\_driver: Invalid access to input buffer at %p\n", input);

return -EFAULT;

}

if (!access\_ok(output, len)) {

printk(KERN\_ERR "crypto\_driver: Invalid access to output buffer at %p\n", output);

return -EFAULT;

}

input\_buf = kmalloc(len, GFP\_KERNEL);

output\_buf = kmalloc(len, GFP\_KERNEL);

if (!input\_buf || !output\_buf) {

printk(KERN\_ERR "crypto\_driver: Failed to allocate kernel buffers\n");

ret = -ENOMEM;

goto out\_free;

}

if (copy\_from\_user(input\_buf, input, len)) {

printk(KERN\_ERR "crypto\_driver: Failed to copy input from user\n");

ret = -EFAULT;

goto out\_free;

}

if (cmd == IOCTL\_ENCODE || cmd == IOCTL\_DECODE) {

struct crypto\_skcipher \*tfm = NULL;

struct skcipher\_request \*req = NULL;

struct scatterlist sg[2];

char iv\_copy[8]; // DES uses 8-byte IV

if (len % 8 != 0) { // DES block size is 8 bytes

printk(KERN\_ERR "crypto\_driver: Invalid length for DES: %zu\n", len);

ret = -EINVAL;

goto out\_free;

}

if (!access\_ok(iv, 8)) { // DES IV size is 8 bytes

printk(KERN\_ERR "crypto\_driver: Invalid access to iv buffer at %p\n", iv);

ret = -EFAULT;

goto out\_free;

}

if (copy\_from\_user(iv\_copy, iv, 8)) { // DES IV size is 8 bytes

printk(KERN\_ERR "crypto\_driver: Failed to copy IV from user\n");

ret = -EFAULT;

goto out\_free;

}

printk(KERN\_INFO "crypto\_driver: Copied IV: %\*ph\n", 8, iv\_copy);

printk(KERN\_INFO "crypto\_driver: Input data: %\*ph\n", (int)len, input\_buf);

tfm = crypto\_alloc\_skcipher("cbc(des)", 0, 0);

if (IS\_ERR(tfm)) {

printk(KERN\_ERR "crypto\_driver: Failed to allocate skcipher handle: %ld\n", PTR\_ERR(tfm));

ret = PTR\_ERR(tfm);

goto out\_free;

}

u8 key[] = {0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe}; // DES uses 8-byte key

printk(KERN\_INFO "crypto\_driver: Setting key: %\*ph\n", 8, key);

if (crypto\_skcipher\_setkey(tfm, key, 8)) { // DES key size is 8 bytes

printk(KERN\_ERR "crypto\_driver: Failed to set key\n");

ret = -EINVAL;

goto out\_free\_tfm;

}

req = skcipher\_request\_alloc(tfm, GFP\_KERNEL);

if (!req) {

printk(KERN\_ERR "crypto\_driver: Failed to allocate skcipher request\n");

ret = -ENOMEM;

goto out\_free\_tfm;

}

sg\_init\_table(sg, 2);

sg\_set\_buf(&sg[0], input\_buf, len);

sg\_set\_buf(&sg[1], output\_buf, len);

skcipher\_request\_set\_crypt(req, &sg[0], &sg[1], len, iv\_copy);

if (cmd == IOCTL\_ENCODE) {

printk(KERN\_INFO "crypto\_driver: Processing IOCTL\_ENCODE\n");

ret = crypto\_skcipher\_encrypt(req);

if (ret == 0)

printk(KERN\_INFO "crypto\_driver: Encryption completed\n");

else

printk(KERN\_ERR "crypto\_driver: Encryption failed with error %d\n", ret);

} else if (cmd == IOCTL\_DECODE) {

printk(KERN\_INFO "crypto\_driver: Processing IOCTL\_DECODE\n");

ret = crypto\_skcipher\_decrypt(req);

if (ret == 0)

printk(KERN\_INFO "crypto\_driver: Decryption completed\n");

else

printk(KERN\_ERR "crypto\_driver: Decryption failed with error %d\n", ret);

}

if (ret == 0) {

printk(KERN\_INFO "crypto\_driver: Output data: %\*ph\n", (int)len, output\_buf);

if (copy\_to\_user(output, output\_buf, len)) {

printk(KERN\_ERR "crypto\_driver: Failed to copy output to user\n");

ret = -EFAULT;

}

}

skcipher\_request\_free(req);

out\_free\_tfm:

crypto\_free\_skcipher(tfm);

} else if (cmd == IOCTL\_HASH) {

struct crypto\_ahash \*tfm = NULL;

struct ahash\_request \*req = NULL;

struct scatterlist sg[1];

u8 hash[SHA1\_DIGEST\_SIZE]; // SHA1 output is 20 bytes

tfm = crypto\_alloc\_ahash("sha1", 0, 0);

if (IS\_ERR(tfm)) {

printk(KERN\_ERR "crypto\_driver: Failed to allocate ahash handle: %ld\n", PTR\_ERR(tfm));

ret = PTR\_ERR(tfm);

goto out\_free;

}

req = ahash\_request\_alloc(tfm, GFP\_KERNEL);

if (!req) {

printk(KERN\_ERR "crypto\_driver: Failed to allocate ahash request\n");

ret = -ENOMEM;

goto out\_free\_ahash;

}

sg\_init\_table(sg, 1);

sg\_set\_buf(&sg[0], input\_buf, len);

ahash\_request\_set\_crypt(req, sg, hash, len);

ret = crypto\_ahash\_digest(req);

if (ret) {

printk(KERN\_ERR "crypto\_driver: SHA1 hash failed with error %d\n", ret);

goto out\_free\_ahash\_req;

}

if (copy\_to\_user(output, hash, SHA1\_DIGEST\_SIZE)) {

printk(KERN\_ERR "crypto\_driver: Failed to copy hash to user\n");

ret = -EFAULT;

goto out\_free\_ahash\_req;

}

printk(KERN\_INFO "crypto\_driver: SHA1 hash completed: %\*ph\n", SHA1\_DIGEST\_SIZE, hash);

ret = 0;

out\_free\_ahash\_req:

ahash\_request\_free(req);

out\_free\_ahash:

crypto\_free\_ahash(tfm);

} else {

printk(KERN\_ERR "crypto\_driver: Invalid ioctl command: %u (hex: %x)\n", cmd, cmd);

ret = -EINVAL;

}

out\_free:

kfree(input\_buf);

kfree(output\_buf);

return ret;

}

static struct file\_operations fops = {

.owner = THIS\_MODULE,

.open = device\_open,

.release = device\_release,

.unlocked\_ioctl = device\_ioctl,

};

static int \_\_init crypto\_driver\_init(void) {

int ret;

struct device \*device;

ret = alloc\_chrdev\_region(&device\_number, 0, 1, DEVICE\_NAME);

if (ret < 0) {

printk(KERN\_ERR "crypto\_driver: Failed to allocate device number\n");

return ret;

}

cdev\_init(&cdev, &fops);

ret = cdev\_add(&cdev, device\_number, 1);

if (ret < 0) {

printk(KERN\_ERR "crypto\_driver: Failed to add cdev\n");

unregister\_chrdev\_region(device\_number, 1);

return ret;

}

class = class\_create(DEVICE\_NAME);

if (IS\_ERR(class)) {

printk(KERN\_ERR "crypto\_driver: Failed to create class\n");

cdev\_del(&cdev);

unregister\_chrdev\_region(device\_number, 1);

return PTR\_ERR(class);

}

device = device\_create(class, NULL, device\_number, NULL, DEVICE\_NAME);

if (IS\_ERR(device)) {

printk(KERN\_ERR "crypto\_driver: Failed to create device\n");

class\_destroy(class);

cdev\_del(&cdev);

unregister\_chrdev\_region(device\_number, 1);

return PTR\_ERR(device);

}

printk(KERN\_INFO "crypto\_driver: Initialized with DES-CBC and SHA1 support\n");

printk(KERN\_INFO "crypto\_driver: IOCTL\_ENCODE=%u (hex: %x), IOCTL\_DECODE=%u (hex: %x), IOCTL\_HASH=%u (hex: %x)\n",

(unsigned int)IOCTL\_ENCODE, (unsigned int)IOCTL\_ENCODE,

(unsigned int)IOCTL\_DECODE, (unsigned int)IOCTL\_DECODE,

(unsigned int)IOCTL\_HASH, (unsigned int)IOCTL\_HASH);

return 0;

}

static void \_\_exit crypto\_driver\_exit(void) {

device\_destroy(class, device\_number);

class\_destroy(class);

cdev\_del(&cdev);

unregister\_chrdev\_region(device\_number, 1);

printk(KERN\_INFO "crypto\_driver: Exited\n");

}

module\_init(crypto\_driver\_init);

module\_exit(crypto\_driver\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Your Name");

MODULE\_DESCRIPTION("A crypto driver with DES-CBC and SHA1 support");

1. Client.py

import tkinter as tk

from tkinter import messagebox, ttk

import socket

import sys

import threading

import fcntl

import os

import struct

import ctypes

from ctypes import c\_void\_p, c\_size\_t, create\_string\_buffer, addressof, Structure

# Constants

BUFFER\_SIZE = 8 # DES block size

MAX\_DATA\_SIZE = 1024

DEVICE\_PATH = "/dev/crypto\_driver"

MAX\_USERNAME = 20

SHA1\_DIGEST\_SIZE = 20 # SHA1 output size

# Color scheme

COLORS = {

'bg': '#2C3E50',

'secondary': '#34495E',

'primary': '#3498DB',

'success': '#27AE60',

'danger': '#E74C3C',

'warning': '#F39C12',

'light': '#ECF0F1',

'dark': '#2C3E50',

'white': '#FFFFFF',

'input\_bg': '#34495E',

'input\_border': '#52A5B7',

'chat\_bg': '#FAFAFA',

'sent\_msg': '#E8F5E8',

'recv\_msg': '#F0F8FF'

}

# Dynamic IOCTL calculation

\_IOC\_NRBITS = 8

\_IOC\_TYPEBITS = 8

\_IOC\_SIZEBITS = 14

\_IOC\_NRSHIFT = 0

\_IOC\_TYPESHIFT = \_IOC\_NRSHIFT + \_IOC\_NRBITS

\_IOC\_SIZESHIFT = \_IOC\_TYPESHIFT + \_IOC\_TYPEBITS

\_IOC\_DIRSHIFT = \_IOC\_SIZESHIFT + \_IOC\_SIZEBITS

\_IOC\_WRITE = 1

def \_IOW(type\_, nr, size):

return (\_IOC\_WRITE << \_IOC\_DIRSHIFT) | (ord(type\_) << \_IOC\_TYPESHIFT) | \

(nr << \_IOC\_NRSHIFT) | (size << \_IOC\_SIZESHIFT)

class IoctlData(Structure):

\_fields\_ = [

("input", c\_void\_p),

("output", c\_void\_p),

("len", c\_size\_t),

("iv", c\_void\_p),

]

IOCTL\_ENCODE = \_IOW('k', 1, ctypes.sizeof(IoctlData))

IOCTL\_DECODE = \_IOW('k', 2, ctypes.sizeof(IoctlData))

IOCTL\_HASH = \_IOW('k', 3, ctypes.sizeof(IoctlData))

def pad\_data(data, block\_size=8): # DES block size

padding\_len = block\_size - (len(data) % block\_size)

return data + bytes([padding\_len] \* padding\_len)

def unpad\_data(padded\_data):

if not padded\_data:

return padded\_data

padding\_len = padded\_data[-1]

if padding\_len < 1 or padding\_len > 8 or len(padded\_data) < padding\_len:

print(f"Invalid padding length: {padding\_len}")

return padded\_data

if not all(padded\_data[-i] == padding\_len for i in range(1, padding\_len + 1)):

print("Invalid padding detected")

return padded\_data

return padded\_data[:-padding\_len]

class AuthWindow:

def \_\_init\_\_(self, root):

self.root = root

self.root.title("🔒 Crypto Chat - Login")

self.root.geometry("420x500")

self.root.configure(bg=COLORS['bg'])

self.root.resizable(False, False)

# Center the window

self.root.update\_idletasks()

width = self.root.winfo\_width()

height = self.root.winfo\_height()

x = (self.root.winfo\_screenwidth() // 2) - (width // 2)

y = (self.root.winfo\_screenheight() // 2) - (height // 2)

self.root.geometry(f'{width}x{height}+{x}+{y}')

# Create main frame

main\_frame = tk.Frame(root, bg=COLORS['bg'])

main\_frame.pack(fill="both", expand=True, padx=20, pady=20)

# Header

header\_frame = tk.Frame(main\_frame, bg=COLORS['bg'])

header\_frame.pack(fill="x", pady=(0, 30))

title\_label = tk.Label(header\_frame, text="🔐 Crypto Chat",

font=("Arial", 24, "bold"),

fg=COLORS['white'],

bg=COLORS['bg'])

title\_label.pack()

subtitle\_label = tk.Label(header\_frame, text="Secure messaging with encryption",

font=("Arial", 10),

fg=COLORS['light'],

bg=COLORS['bg'])

subtitle\_label.pack()

# Login form

form\_frame = tk.Frame(main\_frame, bg=COLORS['secondary'], relief="flat", bd=1)

form\_frame.pack(fill="x", pady=(0, 20))

# Add padding to form

form\_inner = tk.Frame(form\_frame, bg=COLORS['secondary'])

form\_inner.pack(fill="both", expand=True, padx=20, pady=30)

# Username field

username\_frame = tk.Frame(form\_inner, bg=COLORS['secondary'])

username\_frame.pack(fill="x", pady=(0, 20))

username\_label = tk.Label(username\_frame, text="👤 Username:",

font=("Arial", 12, "bold"),

fg=COLORS['white'],

bg=COLORS['secondary'])

username\_label.pack(anchor="w")

self.username\_entry = tk.Entry(username\_frame,

font=("Arial", 12),

bg=COLORS['input\_bg'],

fg=COLORS['white'],

insertbackground=COLORS['white'],

relief="flat",

bd=1)

self.username\_entry.pack(fill="x", pady=(5, 0), ipady=8)

# Password field

password\_frame = tk.Frame(form\_inner, bg=COLORS['secondary'])

password\_frame.pack(fill="x", pady=(0, 20))

password\_label = tk.Label(password\_frame, text="🔑 Password:",

font=("Arial", 12, "bold"),

fg=COLORS['white'],

bg=COLORS['secondary'])

password\_label.pack(anchor="w")

self.password\_entry = tk.Entry(password\_frame,

font=("Arial", 12),

bg=COLORS['input\_bg'],

fg=COLORS['white'],

insertbackground=COLORS['white'],

show="\*",

relief="flat",

bd=1)

self.password\_entry.pack(fill="x", pady=(5, 0), ipady=8)

# Buttons frame

button\_frame = tk.Frame(form\_inner, bg=COLORS['secondary'])

button\_frame.pack(fill="x", pady=(10, 0))

login\_btn = tk.Button(button\_frame, text="🔓 Login",

font=("Arial", 12, "bold"),

bg=COLORS['success'],

fg=COLORS['white'],

relief="flat",

cursor="hand2",

command=self.login)

login\_btn.pack(side="left", fill="x", expand=True, padx=(0, 10), pady=5, ipady=8)

register\_btn = tk.Button(button\_frame, text="📝 Register",

font=("Arial", 12, "bold"),

bg=COLORS['primary'],

fg=COLORS['white'],

relief="flat",

cursor="hand2",

command=self.register)

register\_btn.pack(side="right", fill="x", expand=True, padx=(10, 0), pady=5, ipady=8)

# Footer

footer\_frame = tk.Frame(main\_frame, bg=COLORS['bg'])

footer\_frame.pack(fill="x", pady=(20, 0))

footer\_label = tk.Label(footer\_frame, text="🔒 End-to-end encryption enabled",

font=("Arial", 9),

fg=COLORS['light'],

bg=COLORS['bg'])

footer\_label.pack()

# Bind Enter key

self.username\_entry.bind('<Return>', lambda e: self.password\_entry.focus())

self.password\_entry.bind('<Return>', lambda e: self.login())

def hash\_password(self, password):

try:

fd = os.open(DEVICE\_PATH, os.O\_RDWR)

input\_buf = create\_string\_buffer(password.encode())

output\_buf = create\_string\_buffer(SHA1\_DIGEST\_SIZE)

data = IoctlData(

input=addressof(input\_buf),

output=addressof(output\_buf),

len=len(password),

iv=0

)

print(f"Calling IOCTL\_HASH with len={len(password)}, cmd={hex(IOCTL\_HASH)}")

fcntl.ioctl(fd, IOCTL\_HASH, data)

os.close(fd)

hash\_hex = output\_buf.raw.hex()

print(f"Hashed password: {hash\_hex}")

return hash\_hex

except Exception as e:

print(f"Error hashing password: {e}")

if 'fd' in locals():

os.close(fd)

return None

def login(self):

self.connect(self.username\_entry.get()[:19], self.password\_entry.get()[:19], "L")

def register(self):

self.connect(self.username\_entry.get()[:19], self.password\_entry.get()[:19], "R")

def connect(self, username, password, mode):

if not username or not password:

messagebox.showerror("Error", "Please enter username and password")

return

hash\_str = self.hash\_password(password)

if not hash\_str:

messagebox.showerror("Error", "Failed to hash password")

return

try:

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

sock.settimeout(10)

sock.connect(("127.0.0.1", 8080))

credentials = f"{username}:{hash\_str}"

sock.send(credentials.encode())

response = sock.recv(1024).decode().strip()

if response == "REGISTER\_PROMPT":

sock.send(f"{mode}{credentials}".encode())

response = sock.recv(1024).decode().strip()

if response in ["LOGIN\_SUCCESS", "REGISTER\_SUCCESS"]:

self.root.destroy()

root = tk.Tk()

app = CryptoClient(root, username, sock)

root.mainloop()

else:

messagebox.showerror("Error", f"Authentication failed: {response}")

sock.close()

except Exception as e:

messagebox.showerror("Connection Error", str(e))

class CryptoClient:

def \_\_init\_\_(self, root, client\_id, sock):

self.root = root

self.root.title(f"💬 Crypto Chat - {client\_id}")

self.root.geometry("600x700")

self.root.configure(bg=COLORS['bg'])

self.root.resizable(True, True)

# Center the window

self.root.update\_idletasks()

width = self.root.winfo\_width()

height = self.root.winfo\_height()

x = (self.root.winfo\_screenwidth() // 2) - (width // 2)

y = (self.root.winfo\_screenheight() // 2) - (height // 2)

self.root.geometry(f'{width}x{height}+{x}+{y}')

self.root.protocol("WM\_DELETE\_WINDOW", self.on\_closing)

self.client\_id = client\_id.encode()[:19].ljust(19)

self.sock = sock

self.running = True

self.last\_sent = b''

# Create main layout

self.create\_widgets()

def create\_widgets(self):

# Main container

main\_frame = tk.Frame(self.root, bg=COLORS['bg'])

main\_frame.pack(fill="both", expand=True, padx=10, pady=10)

# Header

header\_frame = tk.Frame(main\_frame, bg=COLORS['secondary'], relief="flat", bd=1)

header\_frame.pack(fill="x", pady=(0, 10))

header\_inner = tk.Frame(header\_frame, bg=COLORS['secondary'])

header\_inner.pack(fill="x", padx=20, pady=15)

title\_label = tk.Label(header\_inner, text="💬 Secure Chat",

font=("Arial", 18, "bold"),

fg=COLORS['white'],

bg=COLORS['secondary'])

title\_label.pack(side="left")

status\_label = tk.Label(header\_inner, text="🟢 Online",

font=("Arial", 12),

fg=COLORS['success'],

bg=COLORS['secondary'])

status\_label.pack(side="right")

# Chat area

chat\_frame = tk.Frame(main\_frame, bg=COLORS['bg'])

chat\_frame.pack(fill="both", expand=True, pady=(0, 10))

# Chat title

chat\_title = tk.Label(chat\_frame, text="💬 Chat History",

font=("Arial", 14, "bold"),

fg=COLORS['white'],

bg=COLORS['bg'])

chat\_title.pack(anchor="w", pady=(0, 5))

# Chat box with scrollbar

chat\_container = tk.Frame(chat\_frame, bg=COLORS['bg'])

chat\_container.pack(fill="both", expand=True)

# Create scrollable text widget

self.chat\_text = tk.Text(chat\_container,

font=("Courier", 10),

bg=COLORS['chat\_bg'],

fg=COLORS['dark'],

wrap=tk.WORD,

state=tk.DISABLED,

relief="flat",

bd=1)

scrollbar = tk.Scrollbar(chat\_container, orient="vertical", command=self.chat\_text.yview)

self.chat\_text.configure(yscrollcommand=scrollbar.set)

self.chat\_text.pack(side="left", fill="both", expand=True)

scrollbar.pack(side="right", fill="y")

# Message input area

input\_frame = tk.Frame(main\_frame, bg=COLORS['secondary'], relief="flat", bd=1)

input\_frame.pack(fill="x", pady=(0, 10))

input\_inner = tk.Frame(input\_frame, bg=COLORS['secondary'])

input\_inner.pack(fill="x", padx=20, pady=20)

# Recipient field

recip\_frame = tk.Frame(input\_inner, bg=COLORS['secondary'])

recip\_frame.pack(fill="x", pady=(0, 15))

recip\_label = tk.Label(recip\_frame, text="👤 To:",

font=("Arial", 12, "bold"),

fg=COLORS['white'],

bg=COLORS['secondary'])

recip\_label.pack(side="left")

self.recip\_entry = tk.Entry(recip\_frame,

font=("Arial", 12),

bg=COLORS['input\_bg'],

fg=COLORS['white'],

insertbackground=COLORS['white'],

relief="flat",

bd=1)

self.recip\_entry.pack(side="left", fill="x", expand=True, padx=(10, 0), ipady=5)

# Default recipient

default\_recip = "lien" if self.client\_id.decode().strip() != "lien" else "minh"

self.recip\_entry.insert(0, default\_recip)

# Message input with send button

msg\_frame = tk.Frame(input\_inner, bg=COLORS['secondary'])

msg\_frame.pack(fill="x")

msg\_label = tk.Label(msg\_frame, text="💬 Message:",

font=("Arial", 12, "bold"),

fg=COLORS['white'],

bg=COLORS['secondary'])

msg\_label.pack(anchor="w", pady=(0, 5))

input\_row = tk.Frame(msg\_frame, bg=COLORS['secondary'])

input\_row.pack(fill="x")

self.msg\_entry = tk.Entry(input\_row,

font=("Arial", 12),

bg=COLORS['input\_bg'],

fg=COLORS['white'],

insertbackground=COLORS['white'],

relief="flat",

bd=1)

self.msg\_entry.pack(side="left", fill="x", expand=True, ipady=8)

self.msg\_entry.insert(0, "Hello")

send\_btn = tk.Button(input\_row, text="🚀 Send",

font=("Arial", 12, "bold"),

bg=COLORS['primary'],

fg=COLORS['white'],

relief="flat",

cursor="hand2",

command=self.send\_message)

send\_btn.pack(side="right", padx=(10, 0), ipady=8, ipadx=15)

# Status bar

status\_frame = tk.Frame(main\_frame, bg=COLORS['bg'])

status\_frame.pack(fill="x")

self.status\_label = tk.Label(status\_frame, text="📊 Ready to send messages (max 8 bytes)",

font=("Arial", 9),

fg=COLORS['light'],

bg=COLORS['bg'])

self.status\_label.pack(anchor="w")

# Bind Enter key to send message

self.msg\_entry.bind('<Return>', lambda e: self.send\_message())

# Start message receiving

self.start\_receive\_thread()

def update\_status(self, message, color=None):

if color is None:

color = COLORS['light']

self.status\_label.config(text=f"📊 {message}", fg=color)

def add\_message\_to\_chat(self, message, message\_type="received"):

self.chat\_text.config(state=tk.NORMAL)

if message\_type == "sent":

self.chat\_text.insert(tk.END, f"📤 {message}\n")

else:

self.chat\_text.insert(tk.END, f"📥 {message}\n")

self.chat\_text.config(state=tk.DISABLED)

self.chat\_text.see(tk.END)

def on\_closing(self):

self.running = False

if self.sock and self.sock.fileno() != -1:

try:

self.sock.close()

except Exception as e:

print(f"Error closing socket: {e}")

if messagebox.askokcancel("Quit", "Do you want to quit?"):

self.root.destroy()

def start\_receive\_thread(self):

def receive\_messages():

while self.running:

if not self.sock or self.sock.fileno() == -1:

print("Socket is invalid, stopping receive thread")

break

try:

self.sock.settimeout(1.0)

data = self.sock.recv(MAX\_USERNAME + 2 \* BUFFER\_SIZE)

if not data:

print("Receive error: No data received, connection closed")

break

print(f"Received data: {data.hex()} ({data.decode(errors='ignore')})")

if data == b"SEND\_SUCCESS":

print("Message sent successfully")

self.update\_status("✅ Message sent successfully", COLORS['success'])

continue

elif data == b"RECIPIENT\_NOT\_FOUND":

self.show\_error("Error", "Recipient not found")

self.update\_status("❌ Recipient not found", COLORS['danger'])

continue

elif data == b"SEND\_FAILED":

self.show\_error("Error", "Failed to send message")

self.update\_status("❌ Failed to send message", COLORS['danger'])

continue

elif data == b"INVALID\_FORMAT":

self.show\_error("Error", "Invalid message format")

self.update\_status("❌ Invalid message format", COLORS['danger'])

continue

if len(data) != MAX\_USERNAME + 2 \* BUFFER\_SIZE:

print(f"Received invalid data length: {len(data)}")

continue

sender = data[:MAX\_USERNAME].decode(errors='ignore').rstrip('\0')

iv = data[MAX\_USERNAME:MAX\_USERNAME + BUFFER\_SIZE]

ciphertext = data[MAX\_USERNAME + BUFFER\_SIZE:MAX\_USERNAME + 2 \* BUFFER\_SIZE]

if len(ciphertext) != BUFFER\_SIZE or len(iv) != BUFFER\_SIZE:

print(f"Invalid ciphertext or IV length: ciphertext={len(ciphertext)}, iv={len(iv)}")

continue

if ciphertext != self.last\_sent:

decrypted = self.decrypt\_data(ciphertext, iv)

if decrypted:

message = decrypted.decode(errors='ignore')

if message:

print(f"Decrypted message: {message}")

self.add\_message\_to\_chat(f"From {sender}: {message}", "received")

self.update\_status(f"📨 New message from {sender}", COLORS['success'])

else:

print(f"Empty message after decoding")

else:

print(f"Decryption failed for data: {ciphertext.hex()}, IV: {iv.hex()}")

else:

print(f"Ignored duplicate message: {ciphertext.hex()}")

except socket.timeout:

continue

except socket.error as e:

if e.errno != 9:

print(f"Receive error: {e}")

break

except Exception as e:

print(f"Unexpected receive error: {e}")

continue

if self.running and (not self.sock or self.sock.fileno() == -1):

self.show\_error("Connection Error", "Connection to server lost")

self.running = False

if self.sock and self.sock.fileno() != -1:

try:

self.sock.close()

except Exception as e:

print(f"Error closing socket: {e}")

threading.Thread(target=receive\_messages, daemon=True).start()

def encrypt\_data(self, plaintext):

try:

print("Opening device: /dev/crypto\_driver")

fd = os.open(DEVICE\_PATH, os.O\_RDWR)

print(f"Device opened, fd: {fd}")

plaintext = plaintext[:8] # DES block size

padded = pad\_data(plaintext.encode())

print(f"Padded plaintext: {padded.hex()}")

if len(padded) > MAX\_DATA\_SIZE or len(padded) % 8 != 0:

print(f"Invalid length: {len(padded)}")

os.close(fd)

return None, None

input\_buf = create\_string\_buffer(padded)

output\_buf = create\_string\_buffer(len(padded))

iv\_data = os.urandom(8) # DES IV size

iv\_buf = create\_string\_buffer(iv\_data)

print(f"Generated IV: {iv\_data.hex()}")

data = IoctlData(

input=addressof(input\_buf),

output=addressof(output\_buf),

len=len(padded),

iv=addressof(iv\_buf)

)

print(f"Size of IoctlData: {ctypes.sizeof(IoctlData)}")

print(f"Calling IOCTL\_ENCODE with len={len(padded)}, cmd={hex(IOCTL\_ENCODE)}")

print(f"IoctlData: input={data.input}, output={data.output}, len={data.len}, iv={data.iv}")

fcntl.ioctl(fd, IOCTL\_ENCODE, data)

print(f"IOCTL\_ENCODE completed, output: {output\_buf.raw.hex()}")

print(f"IV after ioctl: {iv\_buf.raw.hex()}")

os.close(fd)

return iv\_data, bytes(output\_buf.raw[:len(padded)])

except Exception as e:

print(f"Unexpected error in encrypt\_data: {e}")

import traceback

traceback.print\_exc()

if 'fd' in locals():

os.close(fd)

return None, None

def decrypt\_data(self, ciphertext, iv):

try:

if len(ciphertext) != BUFFER\_SIZE or len(iv) != BUFFER\_SIZE:

print(f"Invalid ciphertext or IV length: ciphertext={len(ciphertext)}, iv={len(iv)}")

return None

fd = os.open(DEVICE\_PATH, os.O\_RDWR)

input\_buf = create\_string\_buffer(ciphertext)

output\_buf = create\_string\_buffer(len(ciphertext))

iv\_buf = create\_string\_buffer(iv)

data = IoctlData(

input=addressof(input\_buf),

output=addressof(output\_buf),

len=len(ciphertext),

iv=addressof(iv\_buf)

)

print(f"Calling IOCTL\_DECODE with len={len(ciphertext)}, cmd={hex(IOCTL\_DECODE)}")

print(f"IoctlData: input={data.input}, output={data.output}, len={data.len}, iv={data.iv}")

fcntl.ioctl(fd, IOCTL\_DECODE, data)

print(f"IOCTL\_DECODE completed, output: {output\_buf.raw.hex()}")

os.close(fd)

return unpad\_data(bytes(output\_buf.raw[:len(ciphertext)]))

except Exception as e:

print(f"Unexpected error in decrypt\_data: {e}")

if 'fd' in locals():

os.close(fd)

return None

def send\_message(self):

message = self.msg\_entry.get().strip()

recipient = self.recip\_entry.get().strip()

if not message:

self.show\_error("Error", "Message cannot be empty")

self.update\_status("❌ Message cannot be empty", COLORS['danger'])

return

if len(message.encode()) > 8:

self.show\_error("Error", "Message must not exceed 8 bytes")

self.update\_status("❌ Message too long (max 8 bytes)", COLORS['danger'])

return

if not recipient:

self.show\_error("Error", "Please enter Recipient ID")

self.update\_status("❌ Please enter recipient", COLORS['danger'])

return

if not self.sock or self.sock.fileno() == -1:

self.show\_error("Connection Error", "Socket is invalid, please restart the client")

self.update\_status("❌ Connection error", COLORS['danger'])

return

try:

self.update\_status("🔄 Encrypting message...", COLORS['warning'])

iv, encrypted = self.encrypt\_data(message)

if encrypted is None or len(encrypted) != BUFFER\_SIZE:

self.show\_error("Encryption Error", f"Encryption failed, got {len(encrypted) if encrypted else 'None'} bytes")

self.update\_status("❌ Encryption failed", COLORS['danger'])

return

print(f"Sending IV: {iv.hex()}, Ciphertext: {encrypted.hex()}")

self.update\_status("📤 Sending message...", COLORS['warning'])

self.sock.send(b'S')

self.sock.send(recipient.encode()[:19].ljust(19))

self.sock.send(iv)

self.sock.send(encrypted)

self.last\_sent = encrypted

self.add\_message\_to\_chat(f"To {recipient}: {message}", "sent")

self.msg\_entry.delete(0, tk.END)

self.update\_status("✅ Message sent", COLORS['success'])

except socket.error as e:

self.show\_error("Connection Error", f"Failed to send: {str(e)}")

self.update\_status("❌ Connection error", COLORS['danger'])

self.running = False

if self.sock and self.sock.fileno() != -1:

try:

self.sock.close()

except Exception as e:

print(f"Error closing socket: {e}")

except Exception as e:

self.show\_error("Error", f"Failed to send: {str(e)}")

self.update\_status("❌ Send failed", COLORS['danger'])

def show\_error(self, title, message):

try:

messagebox.showerror(title, message)

except tk.TclError:

print(f"Error (GUI closed): {title} - {message}")

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

print(f"IOCTL\_ENCODE value: {hex(IOCTL\_ENCODE)}")

print(f"IOCTL\_DECODE value: {hex(IOCTL\_DECODE)}")

print(f"IOCTL\_HASH value: {hex(IOCTL\_HASH)}")

print(f"Size of IoctlData: {ctypes.sizeof(IoctlData)}")

root = tk.Tk()

auth = AuthWindow(root)

root.mainloop()

1. Makefile

obj-m += crypto\_driver.o

all:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

gcc -o server server.c -pthread

clean:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

rm -f server

4.server.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#include <sys/socket.h>

#include <sys/types.h>

#include <netinet/in.h>

#include <pthread.h>

#include <errno.h>

#define PORT 8080

#define MAX\_CLIENTS 10

#define MAX\_USERNAME 20

#define MAX\_HASH 41 // SHA1 hash in hex (40 chars) + null terminator

#define MAX\_CRED 60 // username + ':' + hash

#define BUFFER\_SIZE 8 // DES block size

#define MAX\_DATA\_SIZE 1024

struct Client {

int socket;

char username[MAX\_USERNAME];

};

struct Client clients[MAX\_CLIENTS];

int client\_count = 0;

pthread\_mutex\_t clients\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

void log\_message(const char \*recipient, const unsigned char \*iv, const unsigned char \*ciphertext) {

FILE \*file = fopen("messages.txt", "a");

if (file == NULL) {

fprintf(stderr, "Failed to open messages.txt: %s\n", strerror(errno));

return;

}

fprintf(file, "To: %s, IV: ", recipient);

for (int i = 0; i < BUFFER\_SIZE; i++)

fprintf(file, "%02x", iv[i]);

fprintf(file, ", Ciphertext: ");

for (int i = 0; i < BUFFER\_SIZE; i++)

fprintf(file, "%02x", ciphertext[i]);

fprintf(file, "\n");

if (fclose(file) != 0) {

fprintf(stderr, "Failed to close messages.txt: %s\n", strerror(errno));

}

}

int find\_client\_by\_username(const char \*username) {

char padded\_username[MAX\_USERNAME];

strncpy(padded\_username, username, MAX\_USERNAME - 1);

padded\_username[MAX\_USERNAME - 1] = '\0';

for (int i = strlen(padded\_username) - 1; i >= 0 && padded\_username[i] == ' '; i--) {

padded\_username[i] = '\0';

}

pthread\_mutex\_lock(&clients\_mutex);

for (int i = 0; i < client\_count; i++) {

char client\_username[MAX\_USERNAME];

strncpy(client\_username, clients[i].username, MAX\_USERNAME - 1);

client\_username[MAX\_USERNAME - 1] = '\0';

for (int j = strlen(client\_username) - 1; j >= 0 && client\_username[j] == ' '; j--) {

client\_username[j] = '\0';

}

if (strcmp(client\_username, padded\_username) == 0) {

int socket = clients[i].socket;

pthread\_mutex\_unlock(&clients\_mutex);

return socket;

}

}

pthread\_mutex\_unlock(&clients\_mutex);

return -1;

}

void \*handle\_client(void \*client\_socket\_ptr) {

int client\_socket = (int)(intptr\_t)client\_socket\_ptr;

struct sockaddr\_in client\_addr;

socklen\_t addr\_len = sizeof(client\_addr);

getpeername(client\_socket, (struct sockaddr \*)&client\_addr, &addr\_len);

char buffer[MAX\_DATA\_SIZE];

char credentials[MAX\_CRED];

char username[MAX\_USERNAME];

int bytes\_read;

bytes\_read = recv(client\_socket, credentials, MAX\_CRED - 1, 0);

if (bytes\_read <= 0) {

printf("Failed to receive initial data from socket %d: %s\n", client\_socket, strerror(errno));

close(client\_socket);

return NULL;

}

credentials[bytes\_read] = '\0';

printf("Received initial data: %s, bytes read: %d\n", credentials, bytes\_read);

char \*colon = strchr(credentials, ':');

if (!colon || colon == credentials || \*(colon + 1) == '\0') {

send(client\_socket, "INVALID\_CRED", 12, 0);

close(client\_socket);

return NULL;

}

strncpy(username, credentials, colon - credentials);

username[colon - credentials] = '\0';

// Kiểm tra và tạo file users.txt nếu chưa tồn tại

FILE \*file = fopen("users.txt", "a+");

if (!file) {

fprintf(stderr, "Failed to open or create users.txt: %s\n", strerror(errno));

send(client\_socket, "SERVER\_ERROR", 12, 0);

close(client\_socket);

return NULL;

}

// Đọc file để kiểm tra user tồn tại

char stored\_user[MAX\_CRED];

int user\_exists = 0;

rewind(file); // Đảm bảo con trỏ file ở đầu

while (fgets(stored\_user, MAX\_CRED, file)) {

stored\_user[strcspn(stored\_user, "\n")] = '\0';

if (strcmp(stored\_user, credentials) == 0) {

user\_exists = 1;

break;

}

}

if (!user\_exists) {

send(client\_socket, "REGISTER\_PROMPT", 15, 0);

bytes\_read = recv(client\_socket, buffer, MAX\_CRED - 1, 0);

if (bytes\_read <= 0) {

fprintf(stderr, "Failed to receive registration response: %s\n", strerror(errno));

fclose(file);

close(client\_socket);

return NULL;

}

buffer[bytes\_read] = '\0';

if (buffer[0] == 'R') {

fseek(file, 0, SEEK\_END);

fprintf(file, "%s\n", credentials);

send(client\_socket, "REGISTER\_SUCCESS", 16, 0);

} else {

send(client\_socket, "REGISTER\_FAIL", 13, 0);

fclose(file);

close(client\_socket);

return NULL;

}

} else {

send(client\_socket, "LOGIN\_SUCCESS", 13, 0);

}

fclose(file);

pthread\_mutex\_lock(&clients\_mutex);

if (client\_count >= MAX\_CLIENTS) {

pthread\_mutex\_unlock(&clients\_mutex);

send(client\_socket, "SERVER\_FULL", 11, 0);

close(client\_socket);

return NULL;

}

strncpy(clients[client\_count].username, username, MAX\_USERNAME - 1);

clients[client\_count].username[MAX\_USERNAME - 1] = '\0';

for (int i = strlen(clients[client\_count].username); i < MAX\_USERNAME - 1; i++) {

clients[client\_count].username[i] = ' ';

}

clients[client\_count].socket = client\_socket;

client\_count++;

pthread\_mutex\_unlock(&clients\_mutex);

printf("Client %s logged in successfully\n", username);

while (1) {

bytes\_read = recv(client\_socket, buffer, 1, 0);

if (bytes\_read <= 0) {

printf("Client %s disconnected, socket %d: %s\n", username, client\_socket, strerror(errno));

break;

}

buffer[bytes\_read] = '\0';

printf("Received data: %s, bytes read: %d from socket %d\n", buffer, bytes\_read, client\_socket);

if (buffer[0] == 'S') {

char recipient[MAX\_USERNAME];

char sender\_username[MAX\_USERNAME];

unsigned char iv[BUFFER\_SIZE];

unsigned char ciphertext[BUFFER\_SIZE];

bytes\_read = recv(client\_socket, recipient, MAX\_USERNAME - 1, 0);

if (bytes\_read != MAX\_USERNAME - 1) {

printf("Invalid recipient length: %d from socket %d\n", bytes\_read, client\_socket);

send(client\_socket, "INVALID\_FORMAT", 14, 0);

continue;

}

recipient[bytes\_read] = '\0';

printf("Received recipient: %s\n", recipient);

strncpy(sender\_username, username, MAX\_USERNAME - 1);

sender\_username[MAX\_USERNAME - 1] = '\0';

bytes\_read = recv(client\_socket, iv, BUFFER\_SIZE, 0);

if (bytes\_read != BUFFER\_SIZE) {

printf("Invalid IV length: %d from socket %d\n", bytes\_read, client\_socket);

send(client\_socket, "INVALID\_FORMAT", 14, 0);

continue;

}

printf("Received IV: ");

for (int i = 0; i < BUFFER\_SIZE; i++) printf("%02x", iv[i]);

printf("\n");

bytes\_read = recv(client\_socket, ciphertext, BUFFER\_SIZE, 0);

if (bytes\_read != BUFFER\_SIZE) {

printf("Invalid ciphertext length: %d from socket %d\n", bytes\_read, client\_socket);

send(client\_socket, "INVALID\_FORMAT", 14, 0);

continue;

}

printf("Received ciphertext: ");

for (int i = 0; i < BUFFER\_SIZE; i++) printf("%02x", ciphertext[i]);

printf("\n");

int target\_socket = find\_client\_by\_username(recipient);

if (target\_socket == -1) {

printf("Recipient %s not found\n", recipient);

send(client\_socket, "RECIPIENT\_NOT\_FOUND", 19, 0);

continue;

}

log\_message(recipient, iv, ciphertext);

unsigned char message[MAX\_USERNAME + 2 \* BUFFER\_SIZE];

memcpy(message, sender\_username, MAX\_USERNAME - 1);

message[MAX\_USERNAME - 1] = '\0';

memcpy(message + MAX\_USERNAME, iv, BUFFER\_SIZE);

memcpy(message + MAX\_USERNAME + BUFFER\_SIZE, ciphertext, BUFFER\_SIZE);

if (send(target\_socket, message, MAX\_USERNAME + 2 \* BUFFER\_SIZE, 0) != MAX\_USERNAME + 2 \* BUFFER\_SIZE) {

printf("Failed to forward message to %s: %s\n", recipient, strerror(errno));

send(client\_socket, "SEND\_FAILED", 11, 0);

continue;

}

printf("Forwarded message to %s (socket %d)\n", recipient, target\_socket);

send(client\_socket, "SEND\_SUCCESS", 12, 0);

} else {

printf("Invalid message format from socket %d\n", client\_socket);

send(client\_socket, "INVALID\_FORMAT", 14, 0);

}

}

pthread\_mutex\_lock(&clients\_mutex);

for (int i = 0; i < client\_count; i++) {

if (clients[i].socket == client\_socket) {

for (int j = i; j < client\_count - 1; j++) {

clients[j] = clients[j + 1];

}

client\_count--;

break;

}

}

pthread\_mutex\_unlock(&clients\_mutex);

close(client\_socket);

return NULL;

}

int main() {

int server\_socket, client\_socket;

struct sockaddr\_in server\_addr, client\_addr;

socklen\_t addr\_len = sizeof(client\_addr);

server\_socket = socket(AF\_INET, SOCK\_STREAM, 0);

if (server\_socket < 0) {

perror("Socket creation failed");

exit(1);

}

int opt = 1;

if (setsockopt(server\_socket, SOL\_SOCKET, SO\_REUSEADDR, &opt, sizeof(opt))) {

perror("Setsockopt failed");

exit(1);

}

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_addr.s\_addr = INADDR\_ANY;

server\_addr.sin\_port = htons(PORT);

if (bind(server\_socket, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) < 0) {

perror("Bind failed");

exit(1);

}

if (listen(server\_socket, MAX\_CLIENTS) < 0) {

perror("Listen failed");

exit(1);

}

printf("Server listening on port %d...\n", PORT);

while (1) {

client\_socket = accept(server\_socket, (struct sockaddr \*)&client\_addr, &addr\_len);

if (client\_socket < 0) {

perror("Accept failed");

continue;

}

printf("New client connected, socket fd: %d, ip: %s, port: %d\n",

client\_socket, inet\_ntoa(client\_addr.sin\_addr), ntohs(client\_addr.sin\_port));

pthread\_t thread;

if (pthread\_create(&thread, NULL, handle\_client, (void \*)(intptr\_t)client\_socket) != 0) {

perror("Thread creation failed");

close(client\_socket);

}

pthread\_detach(thread);

}

close(server\_socket);

return 0;s

}