Secure Communication Systems & Encrypted Storage

Table of Contents

End-to-End Encrypted Communications

Self-Hosted Secure Messaging Server

```
python
```

```python
#!/usr/bin/env python3
# secure_messaging_server.py - Self-hosted E2E encrypted messaging

import asyncio
import websockets
import json
import ssl
import hashlib
import secrets
from datetime import datetime
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from typing import Dict, Set, Optional
import sqlite3
import base64

class SecureMessagingServer:
    def __init__(self, port=8765):
        self.port = port
        self.clients = {}  # websocket -> user_id
        self.user_keys = {}  # user_id -> public_key
        self.pending_messages = {}  # user_id -> [messages]
        self.active_sessions = {}
        self.init_database()

    def init_database(self):
        """Initialize SQLite database for message storage"""
        self.conn = sqlite3.connect('secure_messages.db')
        self.cursor = self.conn.cursor()

        self.cursor.execute('''
            CREATE TABLE IF NOT EXISTS messages (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                sender TEXT NOT NULL,
                recipient TEXT NOT NULL,
                encrypted_content TEXT NOT NULL,
                timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                delivered BOOLEAN DEFAULT FALSE
            )
        ''')

        self.cursor.execute('''
            CREATE TABLE IF NOT EXISTS users (
                user_id TEXT PRIMARY KEY,
                public_key TEXT NOT NULL,
                created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                last_seen TIMESTAMP
            )
        ''')

        self.conn.commit()

    async def register_client(self, websocket, message):
        """Register new client with public key"""
        user_id = message['user_id']
        public_key = message['public_key']

        # Store user information
        self.clients[websocket] = user_id
        self.user_keys[user_id] = public_key

        # Save to database
        self.cursor.execute('''
            INSERT OR REPLACE INTO users (user_id, public_key, last_seen)
            VALUES (?, ?, ?)
        ''', (user_id, public_key, datetime.now()))
        self.conn.commit()

        # Send pending messages
        if user_id in self.pending_messages:
            for msg in self.pending_messages[user_id]:
                await websocket.send(json.dumps(msg))
            del self.pending_messages[user_id]

        # Send registration confirmation
        await websocket.send(json.dumps({
            'type': 'registered',
            'user_id': user_id,
            'timestamp': datetime.now().isoformat()
        }))

        print(f"Client registered: {user_id}")

    async def handle_message(self, websocket, message):
        """Handle encrypted message relay"""
        sender = self.clients.get(websocket)
        recipient = message['recipient']
        encrypted_content = message['encrypted_content']

        # Store message in database
        self.cursor.execute('''
            INSERT INTO messages (sender, recipient, encrypted_content)
            VALUES (?, ?, ?)
        ''', (sender, recipient, encrypted_content))
        self.conn.commit()

        # Forward to recipient if online
        recipient_ws = None
        for ws, uid in self.clients.items():
            if uid == recipient:
                recipient_ws = ws
                break

        if recipient_ws:
            await recipient_ws.send(json.dumps({
                'type': 'message',
                'sender': sender,
                'content': encrypted_content,
                'timestamp': datetime.now().isoformat()
            }))

            # Mark as delivered
            self.cursor.execute('''
                UPDATE messages SET delivered = TRUE
                WHERE sender = ? AND recipient = ? AND encrypted_content = ?
            ''', (sender, recipient, encrypted_content))
            self.conn.commit()
        else:
            # Store for later delivery
            if recipient not in self.pending_messages:
                self.pending_messages[recipient] = []

            self.pending_messages[recipient].append({
                'type': 'message',
                'sender': sender,
                'content': encrypted_content,
                'timestamp': datetime.now().isoformat()
            })

    async def handle_key_exchange(self, websocket, message):
        """Handle Diffie-Hellman key exchange for perfect forward secrecy"""
        sender = self.clients.get(websocket)
        recipient = message['recipient']
        dh_public = message['dh_public']

        # Forward DH public key to recipient
        recipient_ws = None
        for ws, uid in self.clients.items():
            if uid == recipient:
                recipient_ws = ws
                break

        if recipient_ws:
            await recipient_ws.send(json.dumps({
                'type': 'key_exchange',
                'sender': sender,
                'dh_public': dh_public,
                'timestamp': datetime.now().isoformat()
            }))

    async def handle_client(self, websocket, path):
        """Handle client connection"""
        try:
            async for message in websocket:
                data = json.loads(message)
                msg_type = data.get('type')

                if msg_type == 'register':
                    await self.register_client(websocket, data)
                elif msg_type == 'message':
                    await self.handle_message(websocket, data)
                elif msg_type == 'key_exchange':
                    await self.handle_key_exchange(websocket, data)
                elif msg_type == 'heartbeat':
                    await websocket.send(json.dumps({'type': 'heartbeat_ack'}))

        except websockets.exceptions.ConnectionClosed:
            pass
        finally:
            # Clean up on disconnect
            if websocket in self.clients:
                user_id = self.clients[websocket]
                del self.clients[websocket]

                # Update last seen
                self.cursor.execute('''
                    UPDATE users SET last_seen = ? WHERE user_id = ?
                ''', (datetime.now(), user_id))
                self.conn.commit()

                print(f"Client disconnected: {user_id}")

    async def start_server(self):
        """Start the secure messaging server"""
        # SSL context for WSS
        ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
        ssl_context.load_cert_chain('server.crt', 'server.key')

        async with websockets.serve(
            self.handle_client,
            'localhost',
            self.port,
            ssl=ssl_context
        ):
            print(f"Secure messaging server running on wss://localhost:{self.port}")
            await asyncio.Future()  # Run forever

class SecureMessagingClient:
    """Client for E2E encrypted messaging"""

    def __init__(self, user_id: str):
        self.user_id = user_id
        self.private_key, self.public_key = self.generate_keys()
        self.contacts = {}  # user_id -> public_key
        self.session_keys = {}  # user_id -> session_key

    def generate_keys(self):
        """Generate RSA key pair"""
        private_key = rsa.generate_private_key(
            public_exponent=65537,
            key_size=4096,
            backend=default_backend()
        )
        public_key = private_key.public_key()
        return private_key, public_key

    def generate_session_key(self):
        """Generate AES session key"""
        return secrets.token_bytes(32)  # 256-bit key

    def encrypt_message(self, message: str, recipient_public_key) -> str:
        """Encrypt message with hybrid encryption"""
        # Generate session key
        session_key = self.generate_session_key()

        # Encrypt session key with recipient's public key
        encrypted_session_key = recipient_public_key.encrypt(
            session_key,
            padding.OAEP(
                mgf=padding.MGF1(algorithm=hashes.SHA256()),
                algorithm=hashes.SHA256(),
                label=None
            )
        )

        # Encrypt message with session key (AES)
        iv = secrets.token_bytes(16)
        cipher = Cipher(
            algorithms.AES(session_key),
            modes.CBC(iv),
            backend=default_backend()
        )
        encryptor = cipher.encryptor()

        # Pad message
        message_bytes = message.encode()
        padding_length = 16 - (len(message_bytes) % 16)
        padded_message = message_bytes + bytes([padding_length] * padding_length)

        encrypted_message = encryptor.update(padded_message) + encryptor.finalize()

        # Combine encrypted session key, IV, and encrypted message
        combined = encrypted_session_key + iv + encrypted_message

        return base64.b64encode(combined).decode()

    def decrypt_message(self, encrypted_data: str) -> str:
        """Decrypt message"""
        data = base64.b64decode(encrypted_data)

        # Extract components
        encrypted_session_key = data[:512]  # 4096-bit RSA key
        iv = data[512:528]
        encrypted_message = data[528:]

        # Decrypt session key
        session_key = self.private_key.decrypt(
            encrypted_session_key,
            padding.OAEP(
                mgf=padding.MGF1(algorithm=hashes.SHA256()),
                algorithm=hashes.SHA256(),
                label=None
            )
        )

        # Decrypt message
        cipher = Cipher(
            algorithms.AES(session_key),
            modes.CBC(iv),
            backend=default_backend()
        )
        decryptor = cipher.decryptor()
        padded_message = decryptor.update(encrypted_message) + decryptor.finalize()

        # Remove padding
        padding_length = padded_message[-1]
        message = padded_message[:-padding_length]

        return message.decode()
    async def connect(self, server_url: str):
        """Connect to messaging server"""
        ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
        ssl_context.check_hostname = False
        ssl_context.verify_mode = ssl.CERT_NONE

        async with websockets.connect(server_url, ssl=ssl_context) as websocket:
            # Register with server
            public_key_pem = self.public_key.public_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PublicFormat.SubjectPublicKeyInfo
            ).decode()

            await websocket.send(json.dumps({
                'type': 'register',
                'user_id': self.user_id,
                'public_key': public_key_pem
            }))

            # Handle messages
            async for message in websocket:
                data = json.loads(message)
                await self.handle_received_message(data)

    async def handle_received_message(self, data: Dict):
        """Handle received message"""
        msg_type = data.get('type')

        if msg_type == 'message':
            sender = data['sender']
            encrypted_content = data['content']

            # Decrypt message
            decrypted = self.decrypt_message(encrypted_content)
            print(f"Message from {sender}: {decrypted}")

        elif msg_type == 'registered':
            print(f"Successfully registered as {data['user_id']}")

# Generate SSL certificates for server
def generate_ssl_certificates():
    """Generate self-signed SSL certificates"""
    from cryptography import x509
    from cryptography.x509.oid import NameOID
    from datetime import datetime, timedelta

    # Generate private key
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
        backend=default_backend()
    )

    # Generate certificate
    subject = issuer = x509.Name([
        x509.NameAttribute(NameOID.COUNTRY_NAME, "US"),
        x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, "State"),
        x509.NameAttribute(NameOID.LOCALITY_NAME, "City"),
        x509.NameAttribute(NameOID.ORGANIZATION_NAME, "SecureComm"),
        x509.NameAttribute(NameOID.COMMON_NAME, "localhost"),
    ])

    cert = x509.CertificateBuilder().subject_name(
        subject
    ).issuer_name(
        issuer
    ).public_key(
        private_key.public_key()
    ).serial_number(
        x509.random_serial_number()
    ).not_valid_before(
        datetime.utcnow()
    ).not_valid_after(
        datetime.utcnow() + timedelta(days=365)
    ).add_extension(
        x509.SubjectAlternativeName([x509.DNSName("localhost")]),
        critical=False,
    ).sign(private_key, hashes.SHA256(), default_backend())

    # Write private key
    with open("server.key", "wb") as f:
        f.write(private_key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.TraditionalOpenSSL,
            encryption_algorithm=serialization.NoEncryption()
        ))

    # Write certificate
    with open("server.crt", "wb") as f:
        f.write(cert.public_bytes(serialization.Encoding.PEM))

# Usage
if __name__ == "__main__":
    import sys

    if len(sys.argv) < 2:
        print("Usage: python secure_messaging.py [server|client]")
        sys.exit(1)

    if sys.argv[1] == "server":
        # Generate SSL certificates if not present
        import os
        if not os.path.exists("server.crt"):
            generate_ssl_certificates()

        # Start server
        server = SecureMessagingServer()
        asyncio.run(server.start_server())

    elif sys.argv[1] == "client":
        user_id = input("Enter your user ID: ")
        client = SecureMessagingClient(user_id)
        asyncio.run(client.connect("wss://localhost:8765"))
```

Secure Messaging Infrastructure

Matrix/Element Self-Hosted Setup

bash

```
#!/bin/bash
# matrix_server_setup.sh - Deploy self-hosted Matrix server for secure communications

# Install Docker and Docker Compose
install_docker() {
    curl -fsSL https://get.docker.com -o get-docker.sh
    sudo sh get-docker.sh
    sudo usermod -aG docker $USER

    # Install Docker Compose
    sudo curl -L "https://github.com/docker/compose/releases/latest/download/docker-compose-$(uname -s)-$(uname
    sudo chmod +x /usr/local/bin/docker-compose
}

# Setup Matrix Synapse server
setup_matrix_synapse() {
    # Create directory structure
    mkdir -p ~/matrix/{data,postgres,nginx,element}
    cd ~/matrix

    # Generate configuration
    cat > docker-compose.yml << 'EOF'
version: '3'

services:
  postgres:
    image: postgres:14
    restart: unless-stopped
    volumes:
      - ./postgres:/var/lib/postgresql/data
    environment:
      - POSTGRES_DB=synapse
      - POSTGRES_USER=synapse
      - POSTGRES_PASSWORD=STRONG_PASSWORD_HERE
      - POSTGRES_INITDB_ARGS=--encoding=UTF-8 --lc-collate=C --lc-ctype=C
    networks:
      - matrix

  synapse:
    image: matrixdotorg/synapse:latest
    restart: unless-stopped
    volumes:
      - ./data:/data
    environment:
      - SYNAPSE_CONFIG_PATH=/data/homeserver.yaml
    depends_on:
      - postgres
    networks:
      - matrix
    ports:
      - "8008:8008"
      - "8448:8448"

  element:
    image: vectorim/element-web:latest
    restart: unless-stopped
    volumes:
      - ./element/config.json:/app/config.json
    networks:
      - matrix
    ports:
      - "8080:80"

  nginx:
    image: nginx:alpine
    restart: unless-stopped
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf
      - ./nginx/ssl:/etc/nginx/ssl
    ports:
      - "80:80"
      - "443:443"
    depends_on:
      - synapse
      - element
    networks:
      - matrix

networks:
  matrix:
    driver: bridge
EOF

    # Generate Synapse configuration
    docker run -it --rm \
        -v $(pwd)/data:/data \
        -e SYNAPSE_SERVER_NAME=matrix.yourdomain.com \
        -e SYNAPSE_REPORT_STATS=no \
        matrixdotorg/synapse:latest generate

    # Configure Synapse for PostgreSQL
    cat >> data/homeserver.yaml << 'EOF'

# Database configuration
database:
  name: psycopg2
  args:
    user: synapse
    password: STRONG_PASSWORD_HERE
    database: synapse
    host: postgres
    cp_min: 5
    cp_max: 10

# Security settings
enable_registration: false
registration_shared_secret: "REGISTRATION_SECRET_HERE"
macaroon_secret_key: "MACAROON_SECRET_HERE"
form_secret: "FORM_SECRET_HERE"

# E2E Encryption
e2e_key_export: true
encryption_enabled_by_default_for_room_type: all

# Rate limiting
rc_message:
  per_second: 0.5
  burst_count: 10

rc_registration:
  per_second: 0.05
  burst_count: 2

rc_login:
  address:
    per_second: 0.2
    burst_count: 5
  account:
    per_second: 0.2
    burst_count: 5

# Federation (disable for private server)
federation_domain_whitelist: []
# Or enable federation with specific servers:
# federation_domain_whitelist:
#   - matrix.org
#   - matrix.friend-server.com

# Media storage
media_store_path: /data/media_store
max_upload_size: 50M

# URL preview (disable for privacy)
url_preview_enabled: false

# Metrics and monitoring
```

```
enable_metrics: true
metrics_port: 9000

# Retention policy
retention:
  enabled: true
  default_policy:
    min_lifetime: 1d
    max_lifetime: 365d
  allowed_lifetime_min: 1d
  allowed_lifetime_max: 365d
EOF

    # Configure Element
    cat > element/config.json << 'EOF'
{
    "default_server_config": {
        "m.homeserver": {
            "base_url": "https://matrix.yourdomain.com",
            "server_name": "yourdomain.com"
        },
        "m.identity_server": {
            "base_url": "https://vector.im"
        }
    },
    "disable_custom_urls": false,
    "disable_guests": true,
    "disable_login_language_selector": false,
    "disable_3pid_login": false,
    "brand": "SecureChat",
    "integrations_ui_url": "",
    "integrations_rest_url": "",
    "integrations_widgets_urls": [],
    "default_country_code": "US",
    "show_labs_settings": true,
    "features": {
        "feature_pinning": "labs",
        "feature_custom_status": "labs",
        "feature_custom_tags": "labs",
        "feature_state_counters": "labs"
    },
    "default_federate": false,
    "default_theme": "dark",
    "room_directory": {
        "servers": ["matrix.yourdomain.com"]
    },
    "enable_presence_by_hs_url": {
        "https://matrix.yourdomain.com": false
    },
    "setting_defaults": {
        "breadcrumbs": true
    },
    "jitsi": {
        "preferred_domain": "meet.jit.si"
    }
}
EOF

    # Configure Nginx
    cat > nginx/nginx.conf << 'EOF'
events {
    worker_connections 1024;
}

http {
    upstream synapse {
        server synapse:8008;
    }

    upstream element {
        server element:80;
    }
    # Rate limiting
    limit_req_zone $binary_remote_addr zone=global:10m rate=10r/s;
    limit_req_zone $binary_remote_addr zone=login:10m rate=1r/s;

    server {
        listen 80;
        server_name matrix.yourdomain.com;
        return 301 https://$server_name$request_uri;
    }

    server {
        listen 443 ssl http2;
        server_name matrix.yourdomain.com;

        ssl_certificate /etc/nginx/ssl/cert.pem;
        ssl_certificate_key /etc/nginx/ssl/key.pem;

        # Security headers
        add_header Strict-Transport-Security "max-age=31536000; includeSubDomains; preload" always;
        add_header X-Content-Type-Options "nosniff" always;
        add_header X-Frame-Options "SAMEORIGIN" always;
        add_header X-XSS-Protection "1; mode=block" always;
        add_header Referrer-Policy "strict-origin-when-cross-origin" always;

        # Matrix client-server API
        location /_matrix {
            limit_req zone=global burst=20 nodelay;
            proxy_pass http://synapse;
            proxy_set_header X-Forwarded-For $remote_addr;
            proxy_set_header X-Forwarded-Proto $scheme;
            proxy_set_header Host $host;

            # Increase timeout for long polling
            proxy_read_timeout 600s;
            proxy_send_timeout 600s;
        }

        # Matrix federation API
        location /_matrix/federation {
            proxy_pass http://synapse;
            proxy_set_header X-Forwarded-For $remote_addr;
            proxy_set_header X-Forwarded-Proto $scheme;
            proxy_set_header Host $host;
        }

        # Element web client
        location / {
            proxy_pass http://element;
            proxy_set_header X-Forwarded-For $remote_addr;
            proxy_set_header X-Forwarded-Proto $scheme;
            proxy_set_header Host $host;
        }
    }

    # Matrix federation port
    server {
        listen 8448 ssl http2;
        server_name matrix.yourdomain.com;

        ssl_certificate /etc/nginx/ssl/cert.pem;
        ssl_certificate_key /etc/nginx/ssl/key.pem;

        location / {
            proxy_pass http://synapse;
            proxy_set_header X-Forwarded-For $remote_addr;
            proxy_set_header X-Forwarded-Proto $scheme;
            proxy_set_header Host $host;
        }
    }
}
EOF

    # Generate SSL certificates with Let's Encrypt
    docker run -it --rm \
        -v $(pwd)/nginx/ssl:/etc/letsencrypt \
        certbot/certbot certonly \
        --standalone \
        -d matrix.yourdomain.com \
        --email your-email@domain.com \
        --agree-tos \
        --non-interactive

    # Start services
    docker-compose up -d

    echo "Matrix server deployed!"
    echo "Access Element at: https://matrix.yourdomain.com"
    echo "Create admin user: docker exec -it matrix_synapse_1 register_new_matrix_user -c /data/homeserver.yaml http:/
}

# Setup TURN server for voice/video calls
setup_turn_server() {
    # Install coturn
    sudo apt-get update
    sudo apt-get install -y coturn

    # Configure coturn
    sudo tee /etc/turnserver.conf << EOF
# TURN server configuration
listening-port=3478
tls-listening-port=5349

# Authentication
use-auth-secret
static-auth-secret=YOUR_TURN_SECRET_HERE

# Network
realm=turn.yourdomain.com
server-name=turn.yourdomain.com

# Security
no-multicast-peers
no-cli
no-loopback-peers
no-tcp-relay

# Certificates
cert=/etc/letsencrypt/live/turn.yourdomain.com/fullchain.pem
pkey=/etc/letsencrypt/live/turn.yourdomain.com/privkey.pem

# Logging
log-file=/var/log/turnserver.log
verbose
EOF

    # Enable and start coturn
    sudo systemctl enable coturn
    sudo systemctl start coturn

    # Add TURN configuration to Synapse
    cat >> ~/matrix/data/homeserver.yaml << EOF

# TURN configuration
turn_uris:
  - "turn:turn.yourdomain.com:3478?transport=udp"
  - "turn:turn.yourdomain.com:3478?transport=tcp"
  - "turns:turn.yourdomain.com:5349?transport=tcp"
turn_shared_secret: "YOUR_TURN_SECRET_HERE"
turn_user_lifetime: 86400000
turn_allow_guests: false
EOF

    # Restart Synapse
    cd ~/matrix && docker-compose restart synapse
}
```

```
# Main installation
main() {
  echo "Matrix/Element Secure Communication Server Setup"
  echo "=================================================="

  install_docker
  setup_matrix_synapse
  setup_turn_server

  echo ""
  echo "Setup complete! Next steps:"
  echo "1. Update DNS records for matrix.yourdomain.com"
  echo "2. Configure firewall to allow ports 80, 443, 8448, 3478, 5349"
  echo "3. Create admin user with provided command"
  echo "4. Access Element web client at https://matrix.yourdomain.com"
}

main
```

Encrypted Storage Solutions

Full Disk Encryption Setup

```
bash
```

```
#!/bin/bash
# encrypted_storage_setup.sh - Comprehensive encrypted storage configuration

# LUKS Full Disk Encryption
setup_luks_encryption() {
    echo "Setting up LUKS encryption..."

    # Check available disks
    lsblk

    read -p "Enter device to encrypt (e.g., /dev/sdb): " DEVICE

    # Secure wipe (optional but recommended)
    echo "Securely wiping disk (this will take time)..."
    sudo dd if=/dev/urandom of=$DEVICE bs=4M status=progress

    # Create LUKS container
    echo "Creating LUKS container..."
    sudo cryptsetup luksFormat \
        --type luks2 \
        --cipher aes-xts-plain64 \
        --key-size 512 \
        --hash sha512 \
        --iter-time 5000 \
        --use-random \
        $DEVICE

    # Open LUKS container
    sudo cryptsetup luksOpen $DEVICE encrypted_storage

    # Create filesystem
    sudo mkfs.ext4 /dev/mapper/encrypted_storage

    # Mount encrypted storage
    sudo mkdir -p /mnt/encrypted
    sudo mount /dev/mapper/encrypted_storage /mnt/encrypted

    # Create key file for auto-mounting (optional)
    sudo dd if=/dev/urandom of=/root/keyfile bs=1024 count=4
    sudo chmod 0400 /root/keyfile
    sudo cryptsetup luksAddKey $DEVICE /root/keyfile

    # Add to crypttab for auto-mounting
    echo "encrypted_storage $DEVICE /root/keyfile luks,discard" | sudo tee -a /etc/crypttab

    # Add to fstab
    echo "/dev/mapper/encrypted_storage /mnt/encrypted ext4 defaults 0 2" | sudo tee -a /etc/fstab

    echo "LUKS encryption setup complete!"
}

# VeraCrypt Hidden Volume Setup
setup_veracrypt() {
    echo "Setting up VeraCrypt with hidden volume..."

    # Install VeraCrypt
    wget https://launchpad.net/veracrypt/trunk/1.25.9/+download/veracrypt-1.25.9-Ubuntu-22.04-amd64.deb
    sudo dpkg -i veracrypt-*.deb
    sudo apt-get install -f -y

    # Create VeraCrypt volume with script
    cat > create_veracrypt.sh << 'EOF'
#!/usr/bin/expect -f

set volume_path [lindex $argv 0]
set volume_size [lindex $argv 1]
set outer_password [lindex $argv 2]
set hidden_password [lindex $argv 3]

# Create outer volume
spawn veracrypt -t -c

expect "Volume type:"
```

```
send "1\r"

expect "Enter volume path:"
send "$volume_path\r"

expect "Enter volume size:"
send "$volume_size\r"

expect "Encryption Algorithm:"
send "1\r"

expect "Hash algorithm:"
send "1\r"

expect "Filesystem:"
send "1\r"

expect "Enter password:"
send "$outer_password\r"

expect "Re-enter password:"
send "$outer_password\r"

expect "Enter PIM:"
send "\r"

expect "Enter keyfile path"
send "\r"

expect "Please type at least 320 randomly"
send "[exec dd if=/dev/urandom bs=1 count=320 2>/dev/null]\r"

expect "Done"

# Create hidden volume
spawn veracrypt -t -c --hidden

expect "Enter volume path:"
send "$volume_path\r"

expect "Enter password for the outer volume:"
send "$outer_password\r"

expect "Hidden volume size:"
send "[expr $volume_size / 2]\r"

expect "Encryption Algorithm:"
send "2\r"

expect "Hash algorithm:"
send "2\r"

expect "Filesystem:"
send "1\r"

expect "Enter password:"
send "$hidden_password\r"

expect "Re-enter password:"
send "$hidden_password\r"

expect "Please type at least 320 randomly"
send "[exec dd if=/dev/urandom bs=1 count=320 2>/dev/null]\r"

expect eof
EOF

  chmod +x create_veracrypt.sh

  # Create volume
  read -p "Enter volume path: " VOLUME_PATH
  read -p "Enter volume size (e.g., 10G): " VOLUME_SIZE
  read -s -p "Enter outer volume password: " OUTER_PASS
```

```
    echo
    read -s -p "Enter hidden volume password: " HIDDEN_PASS
    echo

    ./create_veracrypt.sh $VOLUME_PATH $VOLUME_SIZE "$OUTER_PASS" "$HIDDEN_PASS"

    echo "VeraCrypt hidden volume created!"
}

# ZFS Encrypted Dataset
setup_zfs_encryption() {
    echo "Setting up ZFS with native encryption..."

    # Install ZFS
    sudo apt-get update
    sudo apt-get install -y zfsutils-linux

    # Create encrypted ZFS pool
    read -p "Enter devices for ZFS pool (space-separated): " DEVICES
    read -p "Enter pool name: " POOL_NAME

    # Create encrypted pool with passphrase
    sudo zpool create \
        -o ashift=12 \
        -O encryption=aes-256-gcm \
        -O keylocation=prompt \
        -O keyformat=passphrase \
        -O compression=lz4 \
        -O atime=off \
        -O xattr=sa \
        -O normalization=formD \
        $POOL_NAME $DEVICES

    # Create encrypted datasets
    sudo zfs create -o encryption=on $POOL_NAME/private
    sudo zfs create -o encryption=on $POOL_NAME/secure

    # Set permissions
    sudo zfs allow -u $USER mount,create,destroy,snapshot,rollback $POOL_NAME

    # Enable auto-mounting
    sudo zfs set mountpoint=/mnt/$POOL_NAME $POOL_NAME
    sudo zfs mount $POOL_NAME

    # Create snapshot schedule
    cat > /etc/systemd/system/zfs-snapshot.service << EOF
[Unit]
Description=ZFS Snapshot Service
After=zfs.target

[Service]
Type=oneshot
ExecStart=/usr/sbin/zfs snapshot -r $POOL_NAME@\$(date +%%Y%%m%%d-%%H%%M%%S)
ExecStart=/usr/sbin/zfs list -t snapshot -o name,creation -s creation | tail -n +2 | head -n -10 | cut -f1 | xargs -n1 zfs des

[Install]
WantedBy=multi-user.target
EOF

    cat > /etc/systemd/system/zfs-snapshot.timer << EOF
[Unit]
Description=ZFS Snapshot Timer
Requires=zfs-snapshot.service

[Timer]
OnCalendar=hourly
Persistent=true

[Install]
WantedBy=timers.target
EOF

    sudo systemctl enable zfs-snapshot.timer
    sudo systemctl start zfs-snapshot.timer

    echo "ZFS encrypted pool created!"
}

# eCryptfs Home Directory Encryption
setup_ecryptfs() {
    echo "Setting up eCryptfs home directory encryption..."

    # Install eCryptfs
    sudo apt-get install -y ecryptfs-utils

    # Encrypt home directory
    read -p "Enter username to encrypt home: " USERNAME

    # Create backup
    sudo tar -czf /root/home_backup_$USERNAME.tar.gz /home/$USERNAME

    # Setup encrypted home
    sudo ecryptfs-migrate-home -u $USERNAME

    # Configure auto-mounting
    echo "auth required pam_ecryptfs.so unwrap" | sudo tee -a /etc/pam.d/common-auth
    echo "session optional pam_ecryptfs.so unwrap" | sudo tee -a /etc/pam.d/common-session

    echo "eCryptfs setup complete! User must login to complete migration."
}

# Encrypted Container with Plausible Deniability
setup_deniable_encryption() {
    echo "Setting up deniable encryption container..."

    # Create container file
    read -p "Enter container path: " CONTAINER
    read -p "Enter container size (MB): " SIZE

    dd if=/dev/urandom of=$CONTAINER bs=1M count=$SIZE

    # Setup outer container (decoy)
    echo "Setting up outer (decoy) container..."
    OUTER_DEV=$(sudo losetup -f)
    sudo losetup $OUTER_DEV $CONTAINER

    sudo cryptsetup luksFormat \
        --type luks2 \
        --offset 0 \
        --key-size 256 \
        $OUTER_DEV

    sudo cryptsetup luksOpen $OUTER_DEV outer_volume
    sudo mkfs.ext4 /dev/mapper/outer_volume

    # Setup hidden container (real data)
    echo "Setting up hidden container..."
    HIDDEN_OFFSET=$((SIZE * 1024 * 1024 / 2))  # Hide in second half

    sudo cryptsetup luksFormat \
        --type luks2 \
        --offset $HIDDEN_OFFSET \
        --key-size 512 \
        $OUTER_DEV

    # Create mount helper script
    cat > mount_deniable.sh << EOF
#!/bin/bash
CONTAINER="$CONTAINER"
MODE=\$1

if [ "\$MODE" = "outer" ]; then
    sudo cryptsetup luksOpen \$CONTAINER outer_volume
    sudo mount /dev/mapper/outer_volume /mnt/outer
    echo "Outer volume mounted at /mnt/outer"
elif [ "\$MODE" = "hidden" ]; then
```

```
    sudo cryptsetup luksOpen --skip=$HIDDEN_OFFSET \$CONTAINER hidden_volume
    sudo mount /dev/mapper/hidden_volume /mnt/hidden
    echo "Hidden volume mounted at /mnt/hidden"
else
    echo "Usage: \$0 [outer|hidden]"
fi
EOF

    chmod +x mount_deniable.sh

    echo "Deniable encryption container created!"
    echo "Mount with: ./mount_deniable.sh [outer|hidden]"
}

# Main menu
main() {
    echo "Encrypted Storage Setup"
    echo "======================="
    echo "1. LUKS Full Disk Encryption"
    echo "2. VeraCrypt Hidden Volume"
    echo "3. ZFS Native Encryption"
    echo "4. eCryptfs Home Encryption"
    echo "5. Deniable Encryption Container"
    echo "6. Setup All"

    read -p "Select option: " choice

    case $choice in
        1) setup_luks_encryption ;;
        2) setup_veracrypt ;;
        3) setup_zfs_encryption ;;
        4) setup_ecryptfs ;;
        5) setup_deniable_encryption ;;
        6)
            setup_luks_encryption
            setup_veracrypt
            setup_zfs_encryption
            setup_ecryptfs
            setup_deniable_encryption
            ;;
        *) echo "Invalid option" ;;
    esac
}

main
```

Secure File Sharing

Zero-Knowledge File Sharing System

```
python
```

```python
#!/usr/bin/env python3
# secure_file_sharing.py - Zero-knowledge encrypted file sharing

import os
import hashlib
import secrets
import json
import base64
from pathlib import Path
from datetime import datetime, timedelta
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives import hashes, hmac
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2
from cryptography.hazmat.backends import default_backend
from flask import Flask, request, jsonify, send_file
import qrcode
import io

class SecureFileShare:
    def __init__(self, storage_path="./encrypted_files"):
        self.storage_path = Path(storage_path)
        self.storage_path.mkdir(exist_ok=True)
        self.metadata_file = self.storage_path / "metadata.json"
        self.load_metadata()

    def load_metadata(self):
        """Load file metadata"""
        if self.metadata_file.exists():
            with open(self.metadata_file, 'r') as f:
                self.metadata = json.load(f)
        else:
            self.metadata = {}

    def save_metadata(self):
        """Save file metadata"""
        with open(self.metadata_file, 'w') as f:
            json.dump(self.metadata, f)

    def generate_key_from_password(self, password: str, salt: bytes) -> bytes:
        """Generate encryption key from password"""
        kdf = PBKDF2(
            algorithm=hashes.SHA256(),
            length=32,
            salt=salt,
            iterations=100000,
            backend=default_backend()
        )
        return kdf.derive(password.encode())

    def encrypt_file(self, file_path: str, password: str = None) -> Dict:
        """Encrypt file with zero-knowledge encryption"""
        # Generate random password if not provided
        if not password:
            password = secrets.token_urlsafe(32)

        # Read file
        with open(file_path, 'rb') as f:
            plaintext = f.read()

        # Generate salt and derive key
        salt = os.urandom(32)
        key = self.generate_key_from_password(password, salt)

        # Encrypt file
        iv = os.urandom(16)
        cipher = Cipher(
            algorithms.AES(key),
            modes.CBC(iv),
            backend=default_backend()
        )
        encryptor = cipher.encryptor()

        # Pad plaintext
        padding_length = 16 - (len(plaintext) % 16)
        padded_plaintext = plaintext + bytes([padding_length] * padding_length)

        ciphertext = encryptor.update(padded_plaintext) + encryptor.finalize()

        # Generate file ID
        file_id = secrets.token_urlsafe(16)

        # Calculate HMAC for integrity
        h = hmac.HMAC(key, hashes.SHA256(), backend=default_backend())
        h.update(ciphertext)
        mac = h.finalize()

        # Store encrypted file
        encrypted_file_path = self.storage_path / f"{file_id}.enc"
        with open(encrypted_file_path, 'wb') as f:
            f.write(salt + iv + mac + ciphertext)

        # Store metadata (without password)
        self.metadata[file_id] = {
            'original_name': os.path.basename(file_path),
            'size': len(plaintext),
            'encrypted_size': len(ciphertext),
            'upload_time': datetime.now().isoformat(),
            'expires': (datetime.now() + timedelta(days=7)).isoformat(),
            'download_count': 0,
            'max_downloads': 5
        }
        self.save_metadata()

        return {
            'file_id': file_id,
            'password': password,
            'share_link': f"https://share.secure/download/{file_id}#{password}",
            'expires': self.metadata[file_id]['expires']
        }

    def decrypt_file(self, file_id: str, password: str) -> bytes:
        """Decrypt file"""
        # Check if file exists
        encrypted_file_path = self.storage_path / f"{file_id}.enc"
        if not encrypted_file_path.exists():
            raise FileNotFoundError("File not found or expired")

        # Check metadata
        if file_id not in self.metadata:
            raise FileNotFoundError("File metadata not found")

        metadata = self.metadata[file_id]

        # Check expiration
        if datetime.fromisoformat(metadata['expires']) < datetime.now():
            self.delete_file(file_id)
            raise FileNotFoundError("File has expired")

        # Check download limit
        if metadata['download_count'] >= metadata['max_downloads']:
            self.delete_file(file_id)
            raise PermissionError("Download limit exceeded")

        # Read encrypted file
        with open(encrypted_file_path, 'rb') as f:
            encrypted_data = f.read()

        # Extract components
        salt = encrypted_data[:32]
        iv = encrypted_data[32:48]
        mac = encrypted_data[48:80]
        ciphertext = encrypted_data[80:]

        # Derive key
        key = self.generate_key_from_password(password, salt)
        # Verify HMAC
        h = hmac.HMAC(key, hashes.SHA256(), backend=default_backend())
        h.update(ciphertext)
        try:
            h.verify(mac)
        except:
            raise ValueError("Invalid password or corrupted file")

        # Decrypt
        cipher = Cipher(
            algorithms.AES(key),
            modes.CBC(iv),
            backend=default_backend()
        )
        decryptor = cipher.decryptor()
        padded_plaintext = decryptor.update(ciphertext) + decryptor.finalize()

        # Remove padding
        padding_length = padded_plaintext[-1]
        plaintext = padded_plaintext[:-padding_length]

        # Update download count
        metadata['download_count'] += 1
        self.save_metadata()

        return plaintext

    def delete_file(self, file_id: str):
        """Securely delete file"""
        encrypted_file_path = self.storage_path / f"{file_id}.enc"

        if encrypted_file_path.exists():
            # Overwrite file before deletion
            file_size = encrypted_file_path.stat().st_size
            with open(encrypted_file_path, 'wb') as f:
                f.write(os.urandom(file_size))

            # Delete file
            encrypted_file_path.unlink()

        # Remove metadata
        if file_id in self.metadata:
            del self.metadata[file_id]
            self.save_metadata()

    def generate_share_qr(self, share_data: Dict) -> bytes:
        """Generate QR code for sharing"""
        qr = qrcode.QRCode(
            version=1,
            error_correction=qrcode.constants.ERROR_CORRECT_L,
            box_size=10,
            border=4,
        )

        qr.add_data(share_data['share_link'])
        qr.make(fit=True)

        img = qr.make_image(fill_color="black", back_color="white")

        # Convert to bytes
        img_bytes = io.BytesIO()
        img.save(img_bytes, format='PNG')
        img_bytes.seek(0)

        return img_bytes.getvalue()

    def cleanup_expired(self):
        """Clean up expired files"""
        current_time = datetime.now()

        for file_id, metadata in list(self.metadata.items()):
            if datetime.fromisoformat(metadata['expires']) < current_time:
                self.delete_file(file_id)
                print(f"Deleted expired file: {file_id}")

# Flask web interface
app = Flask(__name__)
file_share = SecureFileShare()

@app.route('/upload', methods=['POST'])
def upload_file():
    """Upload and encrypt file"""
    if 'file' not in request.files:
        return jsonify({'error': 'No file provided'}), 400

    file = request.files['file']
    password = request.form.get('password')

    # Save temporary file
    temp_path = f"/tmp/{file.filename}"
    file.save(temp_path)

    try:
        # Encrypt file
        result = file_share.encrypt_file(temp_path, password)

        # Generate QR code
        qr_code = file_share.generate_share_qr(result)

        # Clean up temp file
        os.remove(temp_path)

        return jsonify({
            'success': True,
            'file_id': result['file_id'],
            'share_link': result['share_link'],
            'expires': result['expires'],
            'qr_code': base64.b64encode(qr_code).decode()
        })

    except Exception as e:
        return jsonify({'error': str(e)}), 500

@app.route('/download/<file_id>', methods=['POST'])
def download_file(file_id):
    """Download and decrypt file"""
    password = request.json.get('password')

    if not password:
        return jsonify({'error': 'Password required'}), 400

    try:
        # Decrypt file
        plaintext = file_share.decrypt_file(file_id, password)

        # Get original filename
        metadata = file_share.metadata.get(file_id, {})
        filename = metadata.get('original_name', 'download')

        # Return file
        return send_file(
            io.BytesIO(plaintext),
            as_attachment=True,
            download_name=filename
        )

    except FileNotFoundError as e:
        return jsonify({'error': str(e)}), 404
    except Exception as e:
        return jsonify({'error': str(e)}), 500

@app.route('/info/<file_id>', methods=['GET'])
def file_info(file_id):
    """Get file information (without sensitive data)"""
    if file_id not in file_share.metadata:
        return jsonify({'error': 'File not found'}), 404
    metadata = file_share.metadata[file_id]

    return jsonify({
        'original_name': metadata['original_name'],
        'size': metadata['size'],
        'upload_time': metadata['upload_time'],
        'expires': metadata['expires'],
        'downloads_remaining': metadata['max_downloads'] - metadata['download_count']
    })

# Background cleanup task
def cleanup_task():
    """Periodic cleanup of expired files"""
    import threading
    import time

    def cleanup():
        while True:
            file_share.cleanup_expired()
            time.sleep(3600)  # Run every hour

    thread = threading.Thread(target=cleanup, daemon=True)
    thread.start()

if __name__ == "__main__":
    cleanup_task()

    # Generate SSL certificate
    from cryptography import x509
    from cryptography.x509.oid import NameOID
    from cryptography.hazmat.primitives import serialization
    from cryptography.hazmat.primitives.asymmetric import rsa

    # Generate private key
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
        backend=default_backend()
    )

    # Generate certificate
    subject = issuer = x509.Name([
        x509.NameAttribute(NameOID.COMMON_NAME, "localhost"),
    ])

    cert = x509.CertificateBuilder().subject_name(
        subject
    ).issuer_name(
        issuer
    ).public_key(
        private_key.public_key()
    ).serial_number(
        x509.random_serial_number()
    ).not_valid_before(
        datetime.utcnow()
    ).not_valid_after(
        datetime.utcnow() + timedelta(days=365)
    ).sign(private_key, hashes.SHA256(), default_backend())

    # Save certificate and key
    with open("cert.pem", "wb") as f:
        f.write(cert.public_bytes(serialization.Encoding.PEM))

    with open("key.pem", "wb") as f:
        f.write(private_key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.TraditionalOpenSSL,
            encryption_algorithm=serialization.NoEncryption()
        ))

    # Run Flask with SSL
    app.run(host='0.0.0.0', port=5000, ssl_context=('cert.pem', 'key.pem'))
```

Communication Security Monitoring

Real-Time Communication Security Monitor

```python
```

```python
#!/usr/bin/env python3
# comm_security_monitor.py - Monitor communication channels for security

import time
import psutil
import socket
import ssl
import subprocess
from datetime import datetime
from typing import Dict, List, Set
import json
import hashlib

class CommunicationMonitor:
    def __init__(self):
        self.baseline_connections = set()
        self.encrypted_protocols = {'https', 'ssh', 'tls', 'ssl', 'vpn', 'tor'}
        self.suspicious_ports = {23, 21, 80, 139, 445, 3389}  # Unencrypted services
        self.alerts = []

    def check_encryption_status(self, connection) -> bool:
        """Check if connection is encrypted"""
        port = connection.laddr.port if connection.laddr else 0

        # Common encrypted ports
        encrypted_ports = {443, 22, 993, 995, 465, 587, 8443, 9001}

        if port in encrypted_ports:
            return True

        # Check for VPN connections
        if any(vpn in str(connection) for vpn in ['tun', 'tap', 'wg']):
            return True

        return False

    def monitor_connections(self):
        """Monitor all network connections"""
        while True:
            current_connections = psutil.net_connections()

            for conn in current_connections:
                if conn.status == 'ESTABLISHED':
                    # Check for unencrypted connections
                    if not self.check_encryption_status(conn):
                        if conn.raddr:
                            self.alert_unencrypted_connection(conn)

                    # Check for connections to suspicious ports
                    if conn.raddr and conn.raddr.port in self.suspicious_ports:
                        self.alert_suspicious_port(conn)

            time.sleep(5)

    def alert_unencrypted_connection(self, connection):
        """Alert on unencrypted connection"""
        alert = {
            'timestamp': datetime.now().isoformat(),
            'type': 'unencrypted_connection',
            'local': f"{connection.laddr.ip}:{connection.laddr.port}",
            'remote': f"{connection.raddr.ip}:{connection.raddr.port}" if connection.raddr else "unknown",
            'process': self.get_process_name(connection.pid)
        }

        self.alerts.append(alert)
        print(f"⚠️  UNENCRYPTED CONNECTION: {alert['process']} → {alert['remote']}")

    def alert_suspicious_port(self, connection):
        """Alert on suspicious port usage"""
        alert = {
            'timestamp': datetime.now().isoformat(),
            'type': 'suspicious_port',
            'port': connection.raddr.port,
            'remote': connection.raddr.ip,
            'process': self.get_process_name(connection.pid)
        }

        self.alerts.append(alert)
        print(f"⚠️  SUSPICIOUS PORT: {alert['process']} → {alert['remote']}:{alert['port']}")

    def get_process_name(self, pid):
        """Get process name from PID"""
        try:
            return psutil.Process(pid).name()
        except:
            return "unknown"

    def check_dns_security(self):
        """Check DNS configuration for security"""
        # Check for DNS over HTTPS/TLS
        resolv_conf = open('/etc/resolv.conf').read()

        secure_dns = ['1.1.1.1', '9.9.9.9', '8.8.8.8']
        using_secure_dns = any(dns in resolv_conf for dns in secure_dns)

        if not using_secure_dns:
            print("⚠️  Not using secure DNS servers")

        # Check for DNS leaks
        try:
            result = subprocess.run(['dig', '+short', 'myip.opendns.com', '@resolver1.opendns.com'],
                                  capture_output=True, text=True)
            public_ip = result.stdout.strip()

            # Check if DNS queries go through VPN
            vpn_active = any('tun' in str(conn) for conn in psutil.net_connections())
            if vpn_active and public_ip:
                print(f"DNS leak check: Public IP {public_ip}")
        except:
            pass

    def check_certificate_security(self, hostname: str, port: int = 443):
        """Check SSL/TLS certificate security"""
        try:
            context = ssl.create_default_context()
            with socket.create_connection((hostname, port), timeout=10) as sock:
                with context.wrap_socket(sock, server_hostname=hostname) as ssock:
                    cert = ssock.getpeercert()

                    # Check certificate validity
                    not_after = datetime.strptime(cert['notAfter'], '%b %d %H:%M:%S %Y %Z')
                    days_until_expiry = (not_after - datetime.now()).days

                    if days_until_expiry < 30:
                        print(f"⚠️  Certificate expiring soon: {hostname} ({days_until_expiry} days)")

                    # Check cipher strength
                    cipher = ssock.cipher()
                    if cipher and 'RC4' in cipher[0]:
                        print(f"⚠️  Weak cipher detected: {hostname} using {cipher[0]}")

                    return True
        except Exception as e:
            print(f"⚠️  Certificate check failed for {hostname}: {e}")
            return False

    def generate_security_report(self):
        """Generate communication security report"""
        report = {
            'timestamp': datetime.now().isoformat(),
            'alerts': self.alerts,
            'statistics': {
                'total_alerts': len(self.alerts),
                'unencrypted_connections': len([a for a in self.alerts if a['type'] == 'unencrypted_connection']),
                'suspicious_ports': len([a for a in self.alerts if a['type'] == 'suspicious_port'])
            },
            'recommendations': [
                "Use VPN for all internet connections",
                "Enable DNS over HTTPS",
                "Disable unencrypted protocols",
                "Use Signal or Matrix for messaging",
                "Enable certificate pinning",
                "Regular security audits"
            ]
        }

        with open('comm_security_report.json', 'w') as f:
            json.dump(report, f, indent=2)

        return report

# Usage
if __name__ == "__main__":
    monitor = CommunicationMonitor()

    print("Communication Security Monitor")
    print("==============================")

    # Check DNS security
    monitor.check_dns_security()

    # Check common services
    services = ['google.com', 'cloudflare.com', 'github.com']
    for service in services:
        monitor.check_certificate_security(service)

    # Start monitoring
    try:
        monitor.monitor_connections()
    except KeyboardInterrupt:
        report = monitor.generate_security_report()
        print(f"\nSecurity report saved: comm_security_report.json")
        print(f"Total alerts: {report['statistics']['total_alerts']}")
```

Emergency Communication Protocols

Emergency Secure Communication Setup

```
bash
```

```
#!/bin/bash
# emergency_comm_setup.sh - Emergency secure communication protocols

# Setup emergency Tor hidden service
setup_emergency_tor_service() {
    echo "Setting up emergency Tor hidden service..."

    # Install Tor
    sudo apt-get update
    sudo apt-get install -y tor

    # Configure hidden service
    sudo tee -a /etc/tor/torrc << EOF

# Emergency communication hidden service
HiddenServiceDir /var/lib/tor/emergency_comm/
HiddenServicePort 80 127.0.0.1:8080
HiddenServicePort 22 127.0.0.1:22
HiddenServicePort 8888 127.0.0.1:8888

# Security settings
HiddenServiceMaxStreams 10
HiddenServiceMaxStreamsCloseCircuit 1
SocksPort 0
ControlPort 0
EOF

    # Restart Tor
    sudo systemctl restart tor

    # Get onion address
    sleep 5
    ONION_ADDRESS=$(sudo cat /var/lib/tor/emergency_comm/hostname)

    echo "Emergency Tor service address: $ONION_ADDRESS"

    # Generate QR code for easy sharing
    qrencode -o emergency_tor.png "$ONION_ADDRESS"
    echo "QR code saved to emergency_tor.png"
}

# Setup mesh network communication
setup_mesh_communication() {
    echo "Setting up mesh network communication..."

    # Install Briar headless
    wget https://briarproject.org/downloads/briar-headless.jar

    # Create Briar configuration
    cat > briar_config.properties << EOF
# Briar headless configuration
port=7777
auth.token=$(openssl rand -hex 32)
EOF

    # Start Briar headless
    java -jar briar-headless.jar &

    echo "Briar mesh network started on port 7777"
}

# Setup steganographic communication
setup_stego_channel() {
    echo "Setting up steganographic communication channel..."

    # Install steghide
    sudo apt-get install -y steghide

    # Create stego communication script
    cat > stego_comm.sh << 'EOF'
#!/bin/bash

# Embed message in image
embed_message() {
    MESSAGE="$1"
    IMAGE="$2"
    OUTPUT="$3"
    PASSPHRASE="$4"

    echo "$MESSAGE" > /tmp/message.txt
    steghide embed -cf "$IMAGE" -ef /tmp/message.txt -sf "$OUTPUT" -p "$PASSPHRASE"
    rm /tmp/message.txt
    echo "Message embedded in $OUTPUT"
}

# Extract message from image
extract_message() {
    IMAGE="$1"
    PASSPHRASE="$2"

    steghide extract -sf "$IMAGE" -p "$PASSPHRASE"
}

case "$1" in
    embed)
        embed_message "$2" "$3" "$4" "$5"
        ;;
    extract)
        extract_message "$2" "$3"
        ;;
    *)
        echo "Usage: $0 {embed|extract} [args]"
        ;;
esac
EOF

    chmod +x stego_comm.sh
    echo "Steganographic channel ready"
}

# Setup emergency broadcast system
setup_broadcast_system() {
    echo "Setting up emergency broadcast system..."

    # Create broadcast script using multiple channels
    cat > emergency_broadcast.sh << 'EOF'
#!/bin/bash

MESSAGE="$1"
PRIORITY="$2"

# Encrypt message
ENCRYPTED=$(echo "$MESSAGE" | gpg -c --armor --cipher-algo AES256)

# Broadcast via multiple channels
broadcast_channels() {
    # 1. IPFS broadcast
    if command -v ipfs &> /dev/null; then
        IPFS_HASH=$(echo "$ENCRYPTED" | ipfs add -q)
        echo "IPFS: $IPFS_HASH"
    fi

    # 2. Tor broadcast
    if [ -f /var/lib/tor/emergency_comm/hostname ]; then
        echo "$ENCRYPTED" > /var/www/emergency_message.txt
    fi

    # 3. Mesh network broadcast
    if pgrep -x "briar-headless" > /dev/null; then
        curl -X POST http://localhost:7777/broadcast -d "$ENCRYPTED"
    fi

    # 4. Radio packet broadcast (if available)
    if command -v ax25-send &> /dev/null; then
        echo "$ENCRYPTED" | ax25-send EMRGNCY
    fi
```

```
}

# Send based on priority
case "$PRIORITY" in
    critical)
        broadcast_channels &
        echo "⚠️  CRITICAL MESSAGE BROADCAST ON ALL CHANNELS"
        ;;
    high)
        broadcast_channels
        echo "Message broadcast on available channels"
        ;;
    *)
        echo "$ENCRYPTED" > /tmp/emergency_message.enc
        echo "Message saved to /tmp/emergency_message.enc"
        ;;
esac
EOF

    chmod +x emergency_broadcast.sh
    echo "Emergency broadcast system ready"
}

# Setup dead man's switch
setup_dead_mans_switch() {
    echo "Setting up dead man's switch..."

    cat > dead_mans_switch.sh << 'EOF'
#!/bin/bash

# Configuration
CHECK_INTERVAL=86400  # 24 hours
WARNING_TIME=259200   # 72 hours
TRIGGER_TIME=604800   # 7 days
LAST_CHECK_FILE="/var/lib/deadmans/last_check"
ENCRYPTED_MESSAGE="/var/lib/deadmans/message.gpg"

# Initialize
sudo mkdir -p /var/lib/deadmans
sudo touch $LAST_CHECK_FILE

# Check-in function
check_in() {
    date +%s > $LAST_CHECK_FILE
    echo "Check-in recorded at $(date)"
}

# Monitor function
monitor() {
    while true; do
        if [ -f $LAST_CHECK_FILE ]; then
            LAST_CHECK=$(cat $LAST_CHECK_FILE)
            CURRENT_TIME=$(date +%s)
            TIME_DIFF=$((CURRENT_TIME - LAST_CHECK))

            if [ $TIME_DIFF -gt $TRIGGER_TIME ]; then
                echo "TRIGGERING DEAD MAN'S SWITCH"
                trigger_release
                exit 0
            elif [ $TIME_DIFF -gt $WARNING_TIME ]; then
                echo "WARNING: No check-in for $((TIME_DIFF / 86400)) days"
            fi
        fi

        sleep $CHECK_INTERVAL
    done
}

# Trigger function
trigger_release() {
    # Decrypt and send message
    if [ -f $ENCRYPTED_MESSAGE ]; then
        gpg -d $ENCRYPTED_MESSAGE | ./emergency_broadcast.sh - critical
    fi

    # Additional actions
    # - Send emails to trusted contacts
    # - Post to social media
    # - Upload files to cloud
    # - Trigger other automated systems
}

case "$1" in
    setup)
        echo "Enter emergency message:"
        read -r MESSAGE
        echo "$MESSAGE" | gpg -c -o $ENCRYPTED_MESSAGE
        echo "Dead man's switch configured"
        ;;
    checkin)
        check_in
        ;;
    monitor)
        monitor
        ;;
    *)
        echo "Usage: $0 {setup|checkin|monitor}"
        ;;
esac
EOF

    chmod +x dead_mans_switch.sh

    # Create systemd service
    sudo tee /etc/systemd/system/deadmans-switch.service << EOF
[Unit]
Description=Dead Man's Switch Monitor
After=network.target

[Service]
Type=simple
ExecStart=/usr/local/bin/dead_mans_switch.sh monitor
Restart=always

[Install]
WantedBy=multi-user.target
EOF

    sudo systemctl enable deadmans-switch.service
    echo "Dead man's switch configured"
}

# Main menu
main() {
    echo "Emergency Communication Protocol Setup"
    echo "===================================="
    echo "1. Setup Tor hidden service"
    echo "2. Setup mesh communication"
    echo "3. Setup steganographic channel"
    echo "4. Setup broadcast system"
    echo "5. Setup dead man's switch"
    echo "6. Setup all"

    read -p "Select option: " choice

    case $choice in
        1) setup_emergency_tor_service ;;
        2) setup_mesh_communication ;;
        3) setup_stego_channel ;;
        4) setup_broadcast_system ;;
        5) setup_dead_mans_switch ;;
        6)
            setup_emergency_tor_service
            setup_mesh_communication
            setup_stego_channel
            setup_broadcast_system
            setup_dead_mans_switch
            ;;
        *) echo "Invalid option" ;;
    esac
}

main
```

This completes the comprehensive secure communication and encrypted storage guide! The system includes:

1. **End-to-End Encrypted Communications** - Custom E2E messaging implementation
2. **Secure Messaging Infrastructure** - Self-hosted Matrix/Element setup
3. **Encrypted Storage Solutions** - Multiple encryption methods (LUKS, VeraCrypt, ZFS)
4. **Secure File Sharing** - Zero-knowledge file sharing system
5. **Communication Security Monitoring** - Real-time security monitoring
6. **Emergency Communication Protocols** - Tor, mesh networks, and dead man's switch

All these are legitimate security practices for protecting communications and data privacy. These techniques are essential for journalists, activists, security professionals, and anyone concerned about their digital privacy and security.