Layered Encryption Application for Computer Security

Computer Security

1. Introduction

This report documents the design and implementation of a layered encryption/decryption application using three cryptographic algorithms: Caesar Cipher, Autokey Cipher, and RSA Encryption. The goal is to demonstrate how layered security mechanisms can enhance confidentiality by combining classical and modern cryptographic techniques. The application is built in Python using tkinter for the GUI and pycryptodome for RSA operations.

2. Implementation Overview

Layered Workflow

The encryption/decryption process follows a strict order:

Encryption: Caesar → Autokey → RSA (with Base64 encoding).

Decryption: RSA (Base64 decoding) → Autokey → Caesar.

Key Components:

Caesar Cipher

Function: Shifts characters by a user-defined integer key (e.g., "A" → "D" with shift=3).

Strengths: Simplicity and speed.

Weaknesses: Vulnerable to brute-force attacks (26 possible shifts).

Autokey Cipher

Function: Uses a keyword to initialize a keystream, which is extended using the plaintext itself.

Example: Keyword "KEY" encrypts "HELLO" as "RIJVS".

Strengths: Longer key resistance compared to Vigenère.

Weaknesses: Vulnerable to known-plaintext attacks.

RSA Encryption

Function: Asymmetric encryption using dynamically generated 2048-bit keys.

Steps:

Generate RSA key pair (public/private) for each encryption.

Encrypt the Autokey output with the public key (e, n).

Use Base64 to encode binary RSA ciphertext for safe text representation.

GUI Design

Input fields for text, Caesar shift, Autokey keyword, and RSA parameters (e, d, n).

Buttons to trigger encryption/decryption.

Scrolled text areas for input/output.

3. Security Analysis

Strengths of Layered Design

Defense-in-Depth:

Even if one layer is compromised (e.g., Caesar's weak key space), subsequent layers (Autokey, RSA) add protection.

Hybrid Encryption:

Combines symmetric (Caesar, Autokey) and asymmetric (RSA) cryptography. RSA secures the symmetric ciphertext.

Key Management:

RSA keys are regenerated for each session, reducing long-term key exposure.

Weaknesses and Mitigations

Caesar Cipher:

Risk: Easily breakable.

Mitigation: Used only as the first layer; subsequent layers obscure its output.

Autokey Cipher:

Risk: Vulnerable if the keyword is short or reused.

Mitigation: User-provided keyword adds entropy, but stronger keywords are advised.

RSA Limitations:

Risk: Slow for large data; requires padding (e.g., PKCS1_OAEP) to prevent attacks.

Mitigation: Limited to encrypting small texts (Autokey output).

4. Practical Considerations

Performance:

RSA key generation (2048-bit) adds latency but is acceptable for small texts.

Base64 encoding increases ciphertext size by ~33% but ensures compatibility.

Key Storage:

The RSA private key (d) is stored temporarily in memory during the session. For long-term use, secure storage (e.g., encrypted files) is recommended.

Error Handling:

Input validation (e.g., integer checks for Caesar shift) prevents crashes.

Base64 decoding errors are caught during decryption.

5. Lessons Learned

Layered Security: Combining ciphers can mitigate individual weaknesses but requires careful ordering (weakest → strongest).

Real-World Constraints:

RSA cannot encrypt large texts directly (size limit ≈ 245 bytes for 2048-bit keys).

Character encoding (UTF-8) and padding (Base64) are critical for interoperability.

User Experience:

A minimal GUI improves usability but must enforce secure practices (e.g., strong keywords).

6. Future Improvements

Replace Caesar with a stronger cipher (e.g., AES).

Add hashing (SHA-256) for integrity verification.

Implement secure key storage/retrieval for RSA private keys.

7. Conclusion

This project demonstrates the practical application of layered encryption, emphasizing the importance of combining classical and modern cryptographic techniques. While the individual ciphers have

weaknesses, their layered implementation provides a robust defense against casual attacks. The GUI serves as an educational tool to visualize encryption workflows and key management challenges.

Appendix:

```
import tkinter as tk
from tkinter import ttk, scrolledtext
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
import base64

def caesar_encrypt(plaintext: str, shift: int) -> str:
    shift = shift % 26
    ciphertext = []
    for c in plaintext:
        if c.isalpha():
            offset = ord('A') if c.isupper() else ord('a')
            shifted = (ord(c) - offset + shift) % 26
            ciphertext.append(chr(shifted + offset))
        else:
            ciphertext.append(c)
    return ''.join(ciphertext)

def caesar_decrypt(ciphertext: str, shift: int) -> str:
    return caesar_encrypt(ciphertext, -shift)

def autokey_encrypt(plaintext: str, keyword: str) -> str:
    plaintext_upper = plaintext.upper()
    keyword_upper = ''.join([c.upper() for c in keyword if c.isalpha()])
```

```
    letters = []
    indices = []
    for i, c in enumerate(plaintext_upper):
        if c.isalpha():
            letters.append(c)
            indices.append(i)

    keystream = list(keyword_upper)
    for c in letters:
        if len(keystream) >= len(letters):
            break
        keystream.append(c)
    keystream = keystream[:len(letters)]

    encrypted_letters = []
    for p, k in zip(letters, keystream):
        p_num = ord(p) - ord('A')
        k_num = ord(k) - ord('A')
        c_num = (p_num + k_num) % 26
        encrypted_letters.append(chr(c_num + ord('A')))

    ciphertext = list(plaintext_upper)
    for i, c in zip(indices, encrypted_letters):
        ciphertext[i] = c
    return ''.join(ciphertext)

def autokey_decrypt(ciphertext: str, keyword: str) -> str:
    ciphertext_upper = ciphertext.upper()
    keyword_upper = ''.join([c.upper() for c in keyword if c.isalpha()])
```

```
letters = []
indices = []
for i, c in enumerate(ciphertext_upper):
    if c.isalpha():
        letters.append(c)
        indices.append(i)

if not keyword_upper:
    return ciphertext

keystream = list(keyword_upper)
decrypted_letters = []
for c in letters:
    if not keystream:
        break
    k = keystream.pop(0)
    c_num = ord(c) - ord('A')
    k_num = ord(k) - ord('A')
    p_num = (c_num - k_num) % 26
    p_char = chr(p_num + ord('A'))
    decrypted_letters.append(p_char)
    keystream.append(p_char)

plaintext = list(ciphertext_upper)
for i, p in zip(indices, decrypted_letters):
    plaintext[i] = p
return ''.join(plaintext)
```

```
class CipherApp:
    def __init__(self, master):
        self.master = master
        master.title("Layered Encryption/Decryption")

        self.text_label = ttk.Label(master, text="Input Text:")
        self.text_label.grid(row=0, column=0, sticky=tk.W)
        self.text_input = scrolledtext.ScrolledText(master, width=40, height=4)
        self.text_input.grid(row=0, column=1, columnspan=2, padx=5, pady=5)

        self.caesar_label = ttk.Label(master, text="Caesar Shift Key:")
        self.caesar_label.grid(row=1, column=0, sticky=tk.W)
        self.caesar_key = ttk.Entry(master)
        self.caesar_key.grid(row=1, column=1, sticky=tk.W, padx=5)

        self.autokey_label = ttk.Label(master, text="Autokey Keyword:")
        self.autokey_label.grid(row=2, column=0, sticky=tk.W)
        self.autokey_key = ttk.Entry(master)
        self.autokey_key.grid(row=2, column=1, sticky=tk.W, padx=5)

        self.rsa_e_label = ttk.Label(master, text="RSA e:")
        self.rsa_e_label.grid(row=3, column=0, sticky=tk.W)
        self.rsa_e_entry = ttk.Entry(master)
        self.rsa_e_entry.grid(row=3, column=1, sticky=tk.W, padx=5)

        self.rsa_d_label = ttk.Label(master, text="RSA d:")
        self.rsa_d_label.grid(row=4, column=0, sticky=tk.W)
        self.rsa_d_entry = ttk.Entry(master)
        self.rsa_d_entry.grid(row=4, column=1, sticky=tk.W, padx=5)
```

```
        self.rsa_n_label = ttk.Label(master, text="RSA n:")
        self.rsa_n_label.grid(row=5, column=0, sticky=tk.W)
        self.rsa_n_entry = ttk.Entry(master)
        self.rsa_n_entry.grid(row=5, column=1, sticky=tk.W, padx=5)

        self.encrypt_btn = ttk.Button(master, text="Encrypt", command=self.encrypt)
        self.encrypt_btn.grid(row=6, column=1, padx=5, pady=5)

        self.decrypt_btn = ttk.Button(master, text="Decrypt", command=self.decrypt)
        self.decrypt_btn.grid(row=6, column=2, padx=5, pady=5)

        self.output_label = ttk.Label(master, text="Output:")
        self.output_label.grid(row=7, column=0, sticky=tk.W)
        self.output_area = scrolledtext.ScrolledText(master, width=40, height=4)
        self.output_area.grid(row=7, column=1, columnspan=2, padx=5, pady=5)

        self.private_key = None

    def encrypt(self):
        input_text = self.text_input.get("1.0", tk.END).strip()

        caesar_shift = self.caesar_key.get()
        if not caesar_shift.isdigit():
            self.output_area.delete("1.0", tk.END)
            self.output_area.insert(tk.END, "Caesar shift must be an integer")
            return
        caesar_shift = int(caesar_shift)
```

```
autokey_keyword = self.autokey_key.get()

caesar_ct = caesar_encrypt(input_text, caesar_shift)
autokey_ct = autokey_encrypt(caesar_ct, autokey_keyword)

key = RSA.generate(2048)
self.private_key = key
public_key = key.publickey()

cipher_rsa = PKCS1_OAEP.new(public_key)
try:
    rsa_ct = cipher_rsa.encrypt(autokey_ct.encode('utf-8'))
except ValueError as e:
    self.output_area.delete("1.0", tk.END)
    self.output_area.insert(tk.END, f"RSA encryption error: {e}")
    return

b64_ct = base64.b64encode(rsa_ct).decode('utf-8')

self.output_area.delete("1.0", tk.END)
self.output_area.insert(tk.END, b64_ct)

self.rsa_e_entry.delete(0, tk.END)
self.rsa_e_entry.insert(0, str(public_key.e))
self.rsa_d_entry.delete(0, tk.END)
self.rsa_d_entry.insert(0, str(key.d))
self.rsa_n_entry.delete(0, tk.END)
self.rsa_n_entry.insert(0, str(public_key.n))
```

```
def decrypt(self):
    b64_ct = self.text_input.get("1.0", tk.END).strip()

    caesar_shift = self.caesar_key.get()
    if not caesar_shift.isdigit():
        self.output_area.delete("1.0", tk.END)
        self.output_area.insert(tk.END, "Caesar shift must be an integer")
        return
    caesar_shift = int(caesar_shift)

    autokey_keyword = self.autokey_key.get()

    if not self.private_key:
        self.output_area.delete("1.0", tk.END)
        self.output_area.insert(tk.END, "Encrypt first to generate keys")
        return

    try:
        rsa_ct = base64.b64decode(b64_ct)
    except Exception as e:
        self.output_area.delete("1.0", tk.END)
        self.output_area.insert(tk.END, f"Base64 error: {e}")
        return

    cipher_rsa = PKCS1_OAEP.new(self.private_key)
    try:
        autokey_ct_bytes = cipher_rsa.decrypt(rsa_ct)
    except Exception as e:
        self.output_area.delete("1.0", tk.END)
```

```
            self.output_area.insert(tk.END, f"RSA decryption error: {e}")
            return

        try:
            autokey_ct = autokey_ct_bytes.decode('utf-8')
        except UnicodeDecodeError:
            self.output_area.delete("1.0", tk.END)
            self.output_area.insert(tk.END, "Error decoding Autokey ciphertext")
            return

        caesar_pt = autokey_decrypt(autokey_ct, autokey_keyword)
        plaintext = caesar_decrypt(caesar_pt, caesar_shift)

        self.output_area.delete("1.0", tk.END)
        self.output_area.insert(tk.END, plaintext)

def main():
    root = tk.Tk()
    app = CipherApp(root)
    root.mainloop()

if __name__ == "__main__":
    main()
```

Libraries used: tkinter, pycryptodome, base64.