CMPS426 - Security of Computer Systems and Networks

Cryptography Course Project

Submitted by:

Abdelrahman Ahmed Mohamed	1210248
Ahmed Essam Elshahat	1210346
Karim Magdy Mounir	1210070
Mariam Essam	1180027

Submitted to:

Eng. Aymen Reda

Eng. Mahinour

Date of Submission:

05/03/2024

1. Introduction

1.1 Background and Motivation

Ransomware represents a critical cybersecurity threat worldwide. Understanding its mechanisms is essential for network defenders, developers, and security educators. *FileSecurityDemo* simulates core ransomware behaviors—file encryption, key storage and retrieval, user interaction—within a safe, controlled environment. It serves as both a learning aid and a testbed for defensive strategies.

1.2 Scope and Objectives

- Implement robust file encryption using AES-256 in CBC mode.
- Obfuscate and securely store encryption keys and integrity hashes within large random-padding files.
- Provide a user interface (Tkinter) for ransom notification and decryption workflow.
- Package the tool into a standalone executable (PyInstaller) for easy distribution.
- Measure performance: encryption/decryption throughput, resource usage, and integrity verification accuracy.

2. Architecture and Design

2.1 System Overview

FileSecurityDemo follows a modular design:

1. **Encryption Module:** Processes files in 1MB chunks, applies PKCS#7 padding, and streams ciphertext to new .encblob files.

2. **Key Management:** Generates 32-byte AES keys and 16-byte IVs. Obfuscates them via rotating XOR and stores in a 10MB dummy file at fixed offset.

3. **Integrity Verification:** Computes a SHA-256 hash of all target files before encryption, stores the hash similarly, and verifies post-decryption.

4. **GUI Interaction:** Uses Tkinter for two windows: a ransom note during decryption attempts and a post-encryption warning GUI.

5. Packaging: Bundles Python script and assets into a single executable via PyInstaller.

2.2 Key Components

- **Encryption Module:** encrypt_file / decrypt_file methods with chunked streaming.
- **Key Obfuscator:** _xor_obfuscate rotates through byte masks.
- **Hash Computation:** compute_folder_hash and verify_folder_integrity for tamper detection.
- **GUI Classes:** show_ransom_gui and show_post_encrypt_gui deliver interactive dialogs.
- **Command Handler:** handle_encryption orchestrates encryption/decryption logic based on target path state.

3. Implementation Details

3.1 Encryption Algorithm (AES-CBC)

- Utilizes cryptography library's AES cipher in CBC mode.
- Secure key and IV length: 256-bit key, 128-bit IV.
- Chunk size: 1MB to balance memory usage and throughput.

3.2 Chunked File Processing

- Stream files to avoid loading entire content into memory.

- Pad each chunk via PKCS#7, update padder across chunks, finalize at EOF.

3.3 Key and Hash Obfuscation Technique

Key+IV combined into 48 bytes, XOR-obfuscated with index-based rotating mask.
Stored within a 10MB random-padding file at 1MB offset (.hidden_ransom_key.txt).
Folder hash stored likewise at 2MB offset (.hidden_ransom_hash.txt).

3.4 GUI Design (Tkinter)

- **Ransom Note GUI:** Prompt for manual key entry or “Check Payment” option.
- **Post-Encryption Warning GUI:** Scary message, countdown, and guidance to rerun tool.
- **Assets:** Icon displayed in window, color scheme (black/red) for psychological realism.

3.5 Packaging with PyInstaller

Command:

```
pyinstaller --onefile --name "FileSecurityDemo" --distpath "../dist" --workpath "../build" \
  --add-data "../assets/images.ico;assets" --hidden-import cryptography --noconsole --clean main.py
```

4. Results and Discussion

4.1 Observations

- AES-CBC with chunked streaming delivers consistent performance for large files.
- GUI responsiveness unaffected by background encryption threads.

4.2 Limitations

- Lack of real network-based payment verification.
- No anti-tamper against key file deletion or modification.

4.3 Improvements

- Integrate asynchronous encryption to avoid UI blocking.
- Add secure channel (HTTPS) for simulated payment checks.
- Replace XOR obfuscation with authenticated encryption for key storage.

5. Ransomware Artifact Detection

5.1 Motivation

While our main tool simulates ransomware behavior, it's equally critical to explore detection techniques that identify ransomware-like patterns. One effective heuristic is **entropy analysis**, which estimates randomness in file contents—a characteristic of encrypted or obfuscated files.

5.2 Implementation Overview

We developed a RansomwareScanner module that recursively scans a target directory for ransomware artifacts using three main indicators:

- **Encrypted extension match**: Files ending in .encblob.
- **Hidden key file signature**: Files named .hidden_ransom_key.txt.
- **High entropy values**: Shannon entropy above a threshold (7.5), suggesting encryption or compression.