by alby13

Python File Lock: A 4-Layer Encryption, Quantum-resistant Computer Program

Executive Summary

Python File Lock is an advanced, open-source file encryption program designed to provide exceptional security for sensitive data. Utilizing a unique 4-layer encryption approach coupled with a complex derived key and randomly generated keys, this program offers robust resistance against traditional attacks, including brute-force, and encryption breaking from both current and future, supercomputers and quantum computers. This white paper presents an in-depth analysis of the program's architecture, security features, and potential applications.

1. Introduction

In an era of increasing digital threats and advancing computational power, the need for robust encryption methods has never been more critical. Python File Lock addresses this need by implementing a multi-layered encryption approach that combines several strong cryptographic algorithms. This approach not only provides enhanced security but also aims to future-proof encrypted data against potential advancements in computing technology, including quantum computing.

2. Technical Overview

2.1 Overview of Encryption Layers

Python File Lock employs four distinct layers of encryption:

1. **Fernet**: Utilizes AES-128 in CBC mode with PKCS7 padding, coupled with HMAC using SHA256 for authentication.
2. **AES-256-CBC**: A robust symmetric encryption algorithm widely used for securing sensitive information.
3. **ChaCha20-Poly1305**: A modern stream cipher with built-in authentication, known for its speed and security.
4. **XOR**: A simple yet effective operation that, when used with a truly random key of equal length to the data, provides theoretical perfect secrecy.

[SPACE INTENTIONALLY LEFT BLANK]

2.2 Key Derivation and Management

The program uses a single master key derived from the user's password using PBKDF2 (Password-Based Key Derivation Function). This key derivation process includes:

- **PBKDF2HMAC with SHA256**: Ensures the key is securely derived.
- **100,000 iterations**: Provides increased resistance to brute-force attacks.
- **16-byte salt**: Randomly generated to ensure uniqueness and further strengthen the derived key.

The derived 32-byte (256-bit) key is used for the first three encryption layers, while the XOR layer generates a separate random key for each encryption operation.

2.3 Implementation Details

- **Written in Python**: Leverages the language's cryptography libraries.
- **Random elements**: Incorporates IVs, nonces, and other random elements in each layer to enhance security.
- **Key management and secure random number generation**: Ensures that keys are handled properly and generated securely.

3. Security Analysis

3.1 Encryption Strength

The combination of four distinct encryption layers provides an exceptionally high level of security:

- **Individual layer security**: Each layer alone is considered highly secure against current cryptographic attacks.
- **Multi-layer approach**: Offers defense in depth, requiring an attacker to break through all four layers.
- **Longevity of security**: Conservative estimates suggest that breaking this encryption would take much longer than hundreds of years, even with the advent of quantum computers.

[SPACE INTENTIONALLY LEFT BLANK]

3.2 Resistance to Quantum Computing

While quantum computers pose a theoretical threat to many current encryption methods, Python File Lock's multi-layered approach provides significant protection:

- **Symmetric encryption algorithms**: (AES, ChaCha20) are less vulnerable to quantum attacks compared to asymmetric algorithms.
- **Complexity increase**: The combination of multiple layers exponentially increases the complexity for quantum algorithms.
- **XOR layer**: With a truly random key offers information-theoretic security, immune to computational power increases.

3.3 Summary of Encryption

The file encryption is extremely strong. Each layer adds significant security, making breaking all four layers a monumental task even for the most advanced adversaries.

- **Password dependency**: Security ultimately depends on the password. The use of PBKDF2 with 100,000 iterations significantly slows down brute-force attempts.

- **Breaking the encryption**: Assuming a strong password, breaking this encryption through brute force or cryptanalysis would be infeasible with current and foreseeable future technology, including quantum computers.

- **Conservative estimate**: Even with hypothetical quantum computers, it would likely take billions of years to break this encryption. With classical computers, it would take many times the age of the universe.

- **Practical security**: Attackers are far more likely to attempt to obtain the password through other means (phishing, keylogging, social engineering) rather than trying to break the encryption itself.

In conclusion, the encryption itself is extremely secure. The focus of any security measures should be on protecting the password. As long as the password remains secure and has sufficient entropy (is long and complex enough), the encrypted data can be considered safe from cryptographic attacks for the foreseeable future.

3.4 Potential Vulnerabilities

The primary security considerations for Python File Lock are:

- **Password strength**: The overall security depends on the user's chosen password.
- **Implementation security**: Proper deployment and use of the program are crucial.
- **Side-channel attacks**: If the encryption/decryption process is observable, timing or power analysis attacks could potentially be employed.

4. Best Practices and Recommendations

To maximize the security provided by Python File Lock:

1. **Use strong, high-entropy passwords**: Ensure that passwords are long and complex.
2. **Protect the password**: Keep the password safe from unauthorized access or interception.
3. **Secure the system**: Ensure the security of the system running the encryption/decryption processes.

5. Conclusion

Python File Lock represents a significant advancement in file encryption technology. By leveraging a unique 4-layer encryption approach, it offers a level of security that is highly resistant to both current and anticipated future attacks, including those from quantum computers and considering the advancement of computer technology according to Moore's Law. While the program provides robust file protection, users must still adhere to best practices in password management and system security to fully benefit from its capabilities. This open-source solution demonstrates the potential for innovative approaches in cryptography to address evolving security challenges in the digital age.

This open-source solution demonstrates the potential for innovative approaches in cryptography to address evolving security challenges in the digital age.

6. About the Author

This white paper was compiled by an encryption user and Python programmer, based on a comprehensive analysis of the Python File Lock program. The author has work experience in cybersecurity and computer technology and relies on existing encryption standards and sources rather than a strong personal academic background in cryptography.

[SPACE INTENTIONALLY LEFT BLANK]

7. References

Encryption standards and key management techniques:

1. **Fernet**: Fernet is a recipe that provides symmetric encryption and authentication to data. It is a part of the cryptography library for Python, which is developed by the Python Cryptographic Authority (PYCA).

   - Comparitech – What is Fernet? Secure data encryption in Python:
   https://www.comparitech.com/blog/information-security/what-is-fernet/

   - Cryptography – Fernet (symmetric encryption):
   https://cryptography.io/en/latest/fernet/

2. **AES-128 in CBC mode with PKCS7 padding**: AES-128 in CBC mode with PKCS7 padding is used for encrypting data. The Advanced Encryption Standard (AES) block cipher in Cipher Block Chaining (CBC) mode encrypts the data, along with a random Initialization Vector (IV) and Key.

   - Springer Link – Enhanced image encryption using AES algorithm with CBC mode: a secure and efficient approach:
   https://link.springer.com/article/10.1007/s42044-024-00191-y

3. **HMAC using SHA256 for authentication**: HMAC-SHA256 is a message authentication code that uses a shared secret between the two communicating parties. This signature is generated with the SHA256 algorithm and is sent in the Authorization header by using the HMAC-SHA256 scheme.

   - Wikipedia – HMAC (keyed-hash message authentication code):
   https://en.wikipedia.org/wiki/HMAC
   - Microsoft – How to sign an HTTP request with an HMAC signature:
   https://learn.microsoft.com/en-us/azure/communication-services/tutorials/hmac-header-tutorial
   - JSCAPE by Redwood – What Is HMAC (Hash-based message authentication code), And How Does It Secure File Transfers?:
   https://www.jscape.com/blog/what-is-hmac-and-how-does-it-secure-file-transfers
   - Stack Exchange – When using HMAC-SHA256 do I must save the secret in the database* to verify the sender?:

https://security.stackexchange.com/questions/177553/when-using-hmac-sha256-do-i-must-save-the-secret-in-the-databse-to-verify-the-se

4. **AES-256-CBC**: AES-256-CBC stands for "Advanced Encryption Standard with a 256-bit key in Cipher Block Chaining (CBC) mode." It is a type of symmetric key encryption algorithm that uses a 256-bit key to encode and decode data.
   - Wikipedia – Advanced Encryption Standard (AES) - https://en.wikipedia.org/wiki/Advanced_Encryption_Standard
   - Stack Exchange – Why would I ever use AES-256-CBC if AES-256-GCM is more secure? - https://security.stackexchange.com/questions/184305/why-would-i-ever-use-aes-256-cbc-if-aes-256-gcm-is-more-secure
   - FenixPyre – What is AES-256-CBC?: https://docs.anchormydata.com/docs/what-is-aes-256-cbc

5. **ChaCha20-Poly1305**: ChaCha20-Poly1305 is an authenticated encryption with additional data (AEAD) algorithm, that combines the ChaCha20 stream cipher with the Poly1305 message authentication code. Its usage in IETF protocols is standardized in RFC.
   - Wikipedia – ChaCha20-Poly1305: https://en.wikipedia.org/wiki/ChaCha20-Poly1305
   - RFC Editor – ChaCha20 and Poly1305 for IETF Protocols: https://www.rfc-editor.org/rfc/rfc7539
   - PyCryptodome – ChaCha20-Poly1305 and XChaCha20-Poly1305: https://pycryptodome.readthedocs.io/en/latest/src/cipher/chacha20_poly1305.html
   - Cryptoapp – ChaCha20Poly1305 authenticated encryption scheme combines ChaChaTLS and Poly1305: https://www.cryptopp.com/wiki/ChaCha20Poly1305

6. **XOR**: XOR, or exclusive OR, is a simple bitwise (an operator in a programming language that manipulates the individual bits in a byte or word) operation that allows cryptographers to create strong encryption systems, and consequently is a fundamental building block of practically all modern ciphers.

   - Wikipedia – XOR cipher: https://en.wikipedia.org/wiki/XOR_cipher
   - GeeksforGeeks – XOR Cipher: https://www.geeksforgeeks.org/xor-cipher/
   - Blue Goat Cyber – How XOR is Used in Encryption: https://bluegoatcyber.com/blog/how-is-xor-used-in-encryption/

- Md5 Encrypt & Decrypt – XOR Encrypt and Decrypt:
https://md5decrypt.net/en/Xor/
- Boot.dev – WHY IS EXCLUSIVE OR (XOR) IMPORTANT IN CRYPTOGRAPHY?:
https://blog.boot.dev/cryptography/why-xor-in-cryptography/

7. **Key Derivation and Management**: Key derivation functions (KDFs) are cryptographic algorithms that derive one or more secret keys from a secret value such as a master key, a password, or a passphrase using a pseudorandom function.

   -Wikipedia – Key derivation function:
   https://en.wikipedia.org/wiki/Key_derivation_function
   - Baeldung – What Are Key Derivation Functions?:
   https://www.baeldung.com/cs/kdf-cryptography
   - OWASP – Key Management Cheat Sheet:
   https://cheatsheetseries.owasp.org/cheatsheets/Key_Management_Cheat_Sheet.html
   - Springer Link – Understanding Cryptography > Key Management:
   https://link.springer.com/content/pdf/10.1007/978-3-662-69007-9_14.pdf
   - Wikipedia – Key management:
   https://en.wikipedia.org/wiki/Key_management

8. **PBKDF2HMAC with SHA256**: PBKDF2 applies a pseudorandom function, such as hash-based message authentication code (HMAC), to the input password or passphrase along with a salt value and repeats the process many times to produce a derived key.
   - Wikipedia – PBKDF2:
   https://en.wikipedia.org/wiki/PBKDF2
   - Stack Exchange - What's the difference between PBKDF2 and HMAC-SHA256 in security?:
   https://crypto.stackexchange.com/questions/54539/whats-the-difference-between-pbkdf2-and-hmac-sha256-in-security
   - pbkdf2-hmac-shaPublic on Github – PBKDF2 HMAC SHA256 module in C:
   https://github.com/kurkku/pbkdf2-hmac-sha
   - Code and Life – How to calculate PBKDF2 HMAC SHA256 with Python, example code:
   https://codeandlife.com/2023/01/06/how-to-calculate-pbkdf2-hmac-sha256-with-python,-example-code/

Python Libraries References:

[1] Python Software Foundation. (n.d.). base64 — RFC 3548: Base16, Base32, Base64 Data Encodings.
Retrieved from https://docs.python.org/3/library/base64.html

[2] Python Software Foundation. (n.d.). hashlib — Secure hashes and message digests.
Retrieved from https://docs.python.org/3/library/hashlib.html

[3] Python Software Foundation. (n.d.). getpass — Portable password input.
Retrieved from https://docs.python.org/3/library/getpass.html

[4] The Python Cryptographic Authority. (n.d.). cryptography.
Retrieved from https://cryptography.io/

[5] The Python Cryptographic Authority. (n.d.). Fernet (symmetric encryption).
Retrieved from https://cryptography.io/en/latest/hazmat/primitives/symmetric-encryption.html#fernet

[6] The Python Cryptographic Authority. (n.d.). Hashes.
Retrieved from https://cryptography.io/en/latest/hazmat/primitives/cryptographic-hashes.html

[7] The Python Cryptographic Authority. (n.d.). PBKDF2HMAC.
Retrieved from https://cryptography.io/en/latest/hazmat/primitives/key-derivation.html#pbkdf2hmac

[8] The Python Cryptographic Authority. (n.d.). Ciphers.
Retrieved from https://cryptography.io/en/latest/hazmat/primitives/symmetric-encryption.html#ciphers

[9] The Python Cryptographic Authority. (n.d.). Backends.
Retrieved from https://cryptography.io/en/3.4.2/hazmat/backends/

[10] The Python Cryptographic Authority. (n.d.). ChaCha20Poly1305.
Retrieved from https://cryptography.io/en/latest/hazmat/primitives/aead.html#chacha20-poly1305

Academic papers on multi-layer encryption:

[1] Multi-Layer Encryption for Multi-Level Access Control in Wireless Sensor Networks | SpringerLink
Retrieved from https://link.springer.com/chapter/10.1007/978-0-387-09699-5_49

[2] Data Protection Enhancement in Smart Grid Communication: An Efficient Multi-Layer Encrypting Approach Based on Chaotic Techniques and Steganography by Sajjad Golshannavaz, Osama abd Qasim :: SSRN
Retrieved from https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4872871

[3] Secured Sharing of Data Using Multi-Layer Encryption in Cloud Computing | Harbin Gongcheng Daxue Xuebao/Journal of Harbin Engineering University (harbinengineeringjournal.com)

Retrieved from https://harbinengineeringjournal.com/index.php/journal/article/view/2377

[4] Multi-layer encryption flexible integrating algorithm by Emil M. Oanta of Constanta Maritime University, Iustin Priescu of TUV Hessen, Catalin Apostolescu of Politehnica' University of Bucharest https://www.spiedigitallibrary.org/conference-proceedings-of-spie/12493/2643224/Multi-layer-encryption-flexible-integrating-algorithm/10.1117/12.2643224.short

[SPACE INTENTIONALLY LEFT BLANK]

Additional Reading:

- “Review of the Advanced Encryption Standard” by Nicky Mouha, National Institute of Standards and Technology
Retrieved from https://nvlpubs.nist.gov/nistpubs/ir/2021/NIST.IR.8319.pdf

- “Development of the Advanced Encryption Standard”: This publication discusses the development of the Advanced Encryption Standard (AES), which was the result of a cooperative multiyear effort involving the U.S. government, industry, and the academic community.
Retrieved from https://csrc.nist.gov/pubs/journal/2021/08/development-of-the-advanced-encryption-standard/final

[SPACE INTENTIONALLY LEFT BLANK]