AI-Augmented Secure Shell Script for File Encryption & Decryption

Solution Overview

I wrote a shell script (code. sh) on Kali Linux 2023.1 which can Encrypt or Decrypt any sensitive text files up to 1MB using OpenSSL with AES-256-CBC Encryption Algorithm. It's an interactive script, asks the user for action, handles keys in a secure way, and tries to follow the best practices of file permissions. It also logs all activities with SHA256 Checksums for the purpose of trace-ability.

My approach:

- Use AI to generate boilerplate script logic, structure, and documentation.
- Focus my own effort on hardening security, refining key verification logic, implementing permission handling, and auditing.

AI Tools Utilized (The 90%)

Tool	Usage
ChatGPT	Used to generate the initial version of the shell script with encryption and decryption logic using OpenSSL. Also used for explaining each command, improving permission handling (e.g., chmod 400/600), and integrating PBKDF2 and iteration options.
Google Gemini	Utilized to cross-check secure shell scripting best practices (e.g., hiding password input with stty -echo) and handling conditional logic.
Perplexity AI	Used to verify AES-256-CBC vs AES-256-GCM for secure scripting in Bash and their compatibility with OpenSSL on Linux.
GitHub Copilot	Helped refine the case structure, automate logging, and catch minor syntax issues.

Non-AI Tools Utilized

Tool	Purpose
Kali Linux 2023.1	Primary OS environment for scripting and testing.

GNU Bash (Shell)	Scripting language used to write the script.
OpenSSL	Command-line tool for AES-256 encryption/decryption.
Nano & Vim	Text editors used to edit and debug the script.
sha256sum	To verify file integrity using checksums.
chmod	To apply strict file access permissions.

Critical Thinking & Personal Input (The Crucial 10%)

1. Challenge: Securing Passphrase Input

The AI-generated script initially took passphrase input in plain text, which is a significant security flaw. I introduced ‘stty -echo’ to securely capture passphrase input without it being visible on screen, and added a confirmation prompt to prevent typos.

My fix:

```
stty -echo
read keyy
stty echo
```

2. Challenge: Preventing Decryption with Wrong Key

AI couldn’t distinguish between decrypting and *verifying* a key. If a wrong passphrase was entered, the script would still create a corrupted decrypted file. I used a clever workaround: run OpenSSL with ‘-out /dev/null’ to test the key first before actual decryption.

```
if ! openssl enc -d -aes-256-cbc -in "$1" -out /dev/null ...; then
    echo "Error: Incorrect Key. Aborting."
    exit 1
fi
```

3. File Permission Hardening

AI suggested using ‘chmod’ but did not apply strict permissions. I set:

- chmod 400 for encrypted files (read-only, owner only)
- chmod 600 for decrypted files (read/write, owner only)

This prevents leakage via accidental exposure, especially when running in shared environments.

4. Audit Logging & Checksums

AI-generated logs were generic. I introduced detailed, timestamped logs and SHA256 hashes before and after encryption, to verify integrity and detect tampering.

```
original_checksum=$(sha256sum "$1" | awk '{print $1}')
encrypted_checksum=$(sha256sum "$1.enc" | awk '{print $1}')
```

5. Key Derivation Improvement

AI used OpenSSL AES encryption but skipped PBKDF2 and iteration count, both crucial for modern key stretching. I implemented:

```
-pbkdf2 -iter 10000
```

This defends against brute-force attacks by making key derivation computationally expensive.

Reflection: AI + Human Synergy

What AI Did Well:

- Including some example boilerplate for encryption/decryption using OpenSSL.
- Helped me bootstrap some logic for case, read and file handling quickly.
- Recommendation of flags such as -salt, -pbkdf2 that was in line with the industry's best practices.

What Required Human Expertise:

- AI didn’t do as well on edge cases, such as detecting the wrong key with no bad output.
- It had not been given real permission hygiene (chmod 400/600 etc).
- Logging was not integrity-aware and was generic.
- Didn't get the user experience right (eg securely hiding the passphrase, confirmation prompts).

Conclusion

These exercises were great practice for real world security scripting activities. It was AI that enabled the rapid pace of development, but only human intervention guaranteed the solution was secure, non-vulnerable and practical. My paranoid mindset have turned this into a rather robust script that can actually be used in pen testing environments or secure file handling in Linux.