10 Point Best Practices Checklist for Your Symmetric (Block-Cipher) Encryption System

By Logan Gore, CASP, Security+

This paper was written in March 2015 as a reflection of current best practices and trends. When reading this paper, consideration should be given to its age and whether this information is still current.

Best Practices checklist at a glance:

- ✓ 128-bit security level
- ✓ AES256 or stronger
- ✓ Cryptographically secure random 256 bit keys that are appropriately managed
- ✓ Cipher Block Chaining (CBC) Mode
- ✓ Random IV at least the length of the encryption key
- ✓ HMAC
- ✓ Salting and stretching
- ✓ Secure memory and storage
- ✓ Application and system security
- ✓ Physical security

1. Security Level: Be diligent in the design of your system and research each part. Apply defense in depth and aim for a 128 bit security level across your entire system. This means that any attack will require 2^{128} attempts. Research current attack methodologies, especially those that can reduce your effective security level and key size.
2. Algorithm: Use a strong, well-researched, and standardized algorithm. At minimum, use *AES256 (Rijndael);* do not use older standards such as *DES* or even *3DES*. Always make sure you research the most secure method of implementing the algorithm in your environment.
3. Key: Your key length should be twice as large as your targeted security level; i.e. 256 bits if your goal is 128 bit security. Keys should contain high entropy and be generated using a *Cryptographically Secure Pseudo-Random Number Generator (CSPRNG)* or secure key generator. The lifecycle and management of your keys should always be implemented as part of your overall cryptographic solution. The main aspects of key management in symmetric key systems include:
    - Generation
    - Distribution
    - Installation/Usage
    - Storage
    - Control/Recovery
    - Change
    - Disposal

4. Block-cipher Mode: Research the various block-cipher modes and select one that best serves your solution. Do not use *Electronic Code Book (ECB)* mode for anything. *Cipher Block Chaining (CBC)* mode is very popular because it's secure and easy to implement. The security provided by CBC mode, however, can deteriorate with a bad initialization vector (IV).
5. Initialization Vector (IV): IVs should be the same size as the key (for our purposes 256 bit) and *always* random. Developers can use a *CSPRNG* to generate a random IV; however, AES and other popular algorithms can usually generate a random IV. A well-known IV exploit occurred in WEP encryption. WEP used 24-bit IVs and as a result the private key could be compromised using downloadable software within minutes on a busy network.
6. Message Authentication: Tampering with your data may be the goal of an attacker rather than exfiltration. Message authentication can prevent tampering if the encryption key has been compromised. A *Hashed Message Authentication Code (HMAC)* is a good way to ensure integrity. *HMACs* need their own secret key. Make sure it's random and different from your encryption key. Choose a secure, robust algorithm for message authentication. For best security, HMAC hash size should be greater than or equal to the size of the block-cipher's key. For example, *HMACSHA256* would be a great algorithm to use with *AES256.*
7. Salting and Stretching: Salting and stretching is one of the best ways to mitigate many current cryptographic attacks and solidify your bit security level by increasing your effective key size. Take advantage of standardized stretching functions like *Password Based Key Derivation Function 2* (*PBKDF2*) if they're implemented in your security library or framework. Most stretching functions allow you to specify the number of iterations that will be performed. Whatever this number is, it should make the function take 200-1000ms to execute [Schneier, Ferguson, Kohno].
8. Securing memory and storage: One of the biggest mistakes people make is storing secrets (usually keys) in plaintext form within their apps or cryptographic code. This is a difficult issue for many developers and one that has led to attackers having much success targeting insecure application and configuration files. Take advantage of secure memory, storage, and config solutions offered by your framework. Confirm there are no remnants of sensitive data. Research how your language or framework handles issues like string immutability, memory allocation, and garbage collection. Virtualized or shared systems should be hardened to mitigate *VMEscape* and compromise of the hypervisor.
9. Application and System Security: Keep patches current. Your applications and systems need to be secured using strong passwords and operate using least privileged accounts. Access Control Lists, application white lists, firewalls (network, host, and web), and intrusion detection/prevention systems should be properly configured, maintained, and monitored. Multifactor authentication is always recommended. Perform regular vulnerability scans and penetration tests. Use transport security when moving sensitive data over the wire. *Application sandboxing* should be used when available. Service Level Agreements (SLAs) should be used to ensure that cloud vendors meet performance and security expectations.
10. Physical Security: Many of the recommendations outlined in this paper will be useless if the attacker has physical access to your system. Keep servers and machines secured that host sensitive data. Mobile devices should have device lock-out, drive encryption, and remote-wipe capabilities.

References

[Schneier, Ferguson, Kohno]

Cryptography Engineering: Design Principles and Practical Applications. Wiley Publishing, Inc. 2010.

About the Author

Logan Gore, CASP, Security+ is a software engineer who has published several Open Source projects for C#/.NET and papers on current industry topics such as enterprise architecture and cryptography. Logan is currently involved in cloud, game, and mobile development.

loganlgore@gmail.com