COMPX518-21A

Assignment 3

Table of Contents

[Application Design 1](#_Toc74859299)

[Creating a Vault 1](#_Toc74859300)

[Saving a Vault 1](#_Toc74859301)

[Opening a Vault 1](#_Toc74859302)

[Usage Flow 3](#_Toc74859303)

[Secure Random Number Generator 3](#_Toc74859304)

[Master Password Authentication 4](#_Toc74859305)

[Password Storage 5](#_Toc74859306)

[Attack Vectors 5](#_Toc74859307)

[Physical Access 5](#_Toc74859308)

[Out of Scope Attack Vectors 5](#_Toc74859309)

[Key Loggers 6](#_Toc74859310)

[Memory Dumping/Scanning 6](#_Toc74859311)

[Notes 6](#_Toc74859312)

[Resources 6](#_Toc74859313)

# Application Design

## Creating a Vault

This flow describes the process of creating a new vault.

## Saving a Vault

This flow describes the process of saving a vault. AES GCM is used to ensure authenticity and integrity of the cipher text. Alternatively, CTR or CBC could have been used alongside a method such as Encrypt-then-MAC.

## Opening a Vault

This flow describes the process of opening a vault.

## Usage Flow

# Secure Random Number Generator

To guide my decision in choosing a random number generator for this application, I read the OWASP Cryptographic Storage Cheat Sheet. This document states that a Cryptographically secure pseudo-random number generator should be used when dealing with cryptography, and that in .NET / C#, I should use the **RNGCryptoServiceProvider** class instead of the standard **Random** class.

**RNGCryptoServiceProvider** is better suited for password generation because it provides a more secure random function that is not as repeatable as **Random** (at the expense of speed).

The graph below shows a distrubution of generated characters (50 million 80 character passwords). The least common character was **k** with *42538130* occurances, while the most common character was **w** with *42572116* occurances (a difference of 33,986).

Chart, bar chart

Description automatically generated

# Master Password Authentication

…

TODO: Diagram.

Salting disallows precomputed tables

When the user enters their master password, it’s hashed using one of the following methods:

* PBKDF2-HMAC-SHA1: 720,000 iterations
* PBKDF2-HMAC-SHA256: 310,000 iterations
* PBKDF2-HMAC-SHA512: 120,000 iterations

# Password Storage

The password vault is stored as a \*.vault file. This binary file contains the encrypted passwords, alongside the password hash salt, nonce, and authentication tag. The C# class “DataPacker” is used to pack and unpack the binary file into usable byte arrays. See the table below for a visual representation of the file.

|  |  |  |  |
| --- | --- | --- | --- |
| **Cipher Text** | **Salt** | **Nonce** | **Tag** |
| Variable Bytes | 32 Bytes | 12 bytes | 16 bytes |

The cipher text is generated using AES GCM (Galois/Counter Mode) as per the OWASP cryptographic storage recommendations. The GCM cipher mode provides both data authenticity (integrity) and confidentially. GCM does not require the use of an external “Encrypt-then-MAC” scheme which is commonly used during CBC and CTR cipher modes (as they don’t provide any guarantees on the authenticity of the encrypted data).

The key used for AES GCM is a 256bit value obtained from a Key Derivation Function (KDF). This KDF is mentioned in more detail in the previous section, “Master Password Authentication”. In summary, the key is derived from the user’s master password going through a PBKDF2-HMAC-SHA512 function with 200,000 iterations.

In order to access the vault, the user must supply the same master password which they used to first create the vault. Without this master password, the contents of the vault cannot be decrypted (the salt, nonce and tag alone cannot decrypt the vault). The users master password, generated hash, or key are never stored to disk.

Once the plain text is obtained, a System.Text.Json Serializer is used to turn the raw bytes into a c# class which contains all the passwords (and vice versa).

# Attack Vectors

## Physical Access

An attacker may gain physical access to the vault file which contains all the encrypted passwords. Even if the attacker knows the file layout (cipher text, salt, nounce and tag), they cannot gain access to the file without brute forcing the master password. A unique salt (randomly generated with a cryptographically secure method) ensures that Rainbow table attacks (containing precomputed hashes) cannot be used against the vault. The KDF (Key Derivation Function – See “Master Password Authentication”) ensures the attacker must spend x amount of time to generate each password, further lessoning the appeal of brute force.

# Out of Scope Attack Vectors

There are two main attack vectors that I considered out of scope for this application and did not design a way to prevent them. Key loggers and memory dumps.

## Key Loggers

There is no way for me to detect if a key logger is on the user’s system, or prevent the users input from being detected when they are entering their master password. If a key logger can gain access to the user’s master password, the users password vault is then compromised (and likely everything on their system regardless).

## Memory Dumping/Scanning

If a malicious user or program is scanning or dumping system memory, an attacker may be able to access to traces of the user’s master password (although not explicitly stored anywhere, it is sent through methods), the key used to encrypt the vault (stored in memory while the vault is open) and all the users encrypted passwords (stored as an object in memory while the vault is open). I’ve considered this out of scope as if someone is scanning / dumping system memory, nothing is safe and the whole system is compromised.

Additionally, due to the C# Garbage collector, after closing a vault, the vault contents and key may still hang out in memory. These variables are set to null within the application, but the garbage collector will decide when to get rid of them.

# Notes

When running the program on macOS, you may get an error like “No usable version of libssl was found”.

See:

<https://docs.microsoft.com/en-us/dotnet/standard/security/cross-platform-cryptography#aes-ccm-and-aes-gcm-on-macos>

Also see: <https://blog.bokan.io/posts/aesccm-and-aesgcm-in-dotnet-core-on-macos-catalina/>

Personally, I already had OpenSSL installed, I added the following environment variable:

* ***export*** [***DYLD\_LIBRARY\_PATH=/usr/local/opt/openssl@1.1/lib***](mailto:DYLD_LIBRARY_PATH=/usr/local/opt/openssl@1.1/lib)

This application has been tested to run on Windows 10 v21H1, and macOS Big Sur, you will need the .NET 5 SDK installed.

# Resources

* <https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html>
* <https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.rngcryptoserviceprovider>
* <https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.aesgcm?view=net-5.0>
* <https://owasp.org/www-community/password-special-characters>
* <https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#pbkdf2>
* <https://1password.com/files/1Password-White-Paper.pdf>
* <https://www.dashlane.com/download/Dashlane_SecurityWhitePaper_March2021.pdf>