SHIELD Security Design

Om Santosh Patil
patilom001@gmail.com

Key Security Features

SHIELD offers a number of security features, including

True end-to-end encryption

Encryption keys are derived from your master password dynamically by the client on your device, and all encryption and decryption is processed locally on your device. We go beyond relying on the standard TLS encryption by always encrypting your information, securing you from a wide variety of attacks.

Zero knowledge architecture

We have designed our systems with the user privacy as the most important priority. All your data is encrypted with the most advanced encryption algorithms so that only you have access to your data.

Privacy by design, trust the math

A typical web service stores your passwords hash and encryption keys. If captured, they can be used in password cracking attempts. They also transfer your passwords in plain text relying on TLS encryption. We use a password key derivation function algorithm with an asymmetric hashing algorithm to derive your encryption keys on the client side, this key is never stored anywhere. We then double hash this key to verify user authenticity.

Traditional Encryption and Authentication

In traditional encryption and authentication models, systems often rely on sending plain-text passwords and storing encryption keys or hashes directly on the server. While this approach may seem simple, it opens up several critical vulnerabilities:

Plain-Text Password Transmission

In many older or less secure systems, passwords are transmitted in plain text over the network. Even if these transmissions occur over encrypted channels (e.g., TLS/SSL), there is still a risk. If the encrypted channel is compromised, such as through a man-in-the-middle attack or if an attacker has access to the TLS keys, the plain-text password can be intercepted, granting attackers full access to the system.

Storing Plain Hashes

A common practice in traditional systems is to hash passwords (using algorithms like MD5, SHA-1, or even SHA-256) and store the hashes in the database. However, this model is vulnerable in multiple ways:

- **Hash Cracking**: Password hashes are vulnerable to brute-force attacks and precompiled attacks like rainbow tables. If an attacker can access the hashed password, they can reverse the hash through brute-force methods, especially if users have weak or commonly used passwords.
- **Lack of Salt**: If no salt (a unique random value) is used when hashing passwords, attackers can compare stolen hashes with precompiled tables or hashes from other users to crack passwords even faster.

Static Encryption Keys

In traditional encryption schemes, encryption keys (used for data encryption) are often stored on the server. This poses a significant risk because:

- **Key Theft**: If an attacker gains access to the server, they can steal the encryption keys. Once they have the keys, they can decrypt all the stored data, rendering encryption efforts useless.
- **Insider Threats**: If the server stores both the encryption key and the data, even an internal employee with access to the server could easily retrieve both and decrypt sensitive information.

No End-to-End Encryption

Traditional systems often encrypt data only during transmission (e.g., using TLS/SSL) but not at rest. This means that once the data reaches the server, it is stored in plaintext or with minimal encryption. If an attacker compromises the server, they can access all stored data.

Conclusion

Traditional encryption and authentication systems face several security flaws due to the practice of sending plain-text passwords and directly storing encryption keys or hashes on servers. These vulnerabilities can be exploited through man-in-the-middle attacks, brute-force cracking of password hashes, and theft of encryption keys, leading to massive data breaches and unauthorized access. More modern, secure models like zero-knowledge architectures and end-to-end encryption are necessary to mitigate these risks.

Our Security Design

Signup Process

1. **User Submits Password (P1)**
   User provides their password (P1) and clicks submit.
2. **Generate AES-GCM Key (K1)**
   On the client side, 250000 iterations of PBKDF2 with SHA-256 is used to derive an AES-GCM encryption key (K1) from P1, using a randomly generated salt (S1).
3. **Generate Hash (H1)**
   A hash (H1) is created by hashing P1 again using K1 as the salt with SHA-256.
4. **Send Data to Backend**
   The client sends S1, H1, and a randomly generated initialization vector (IV) to the backend for storage.
5. **K1 for Encryption**
   K1 is used to encrypt all user data with AES-GCM-256. The JWT token contains an encrypted package with S1, IV, and H1 along with user credentials.

Login Process

1. **User Submits Password (P2)**
   User enters password (P2) and clicks login.
2. **Derive AES-GCM Key (K2)**
   K2 is derived from P2 using 250000 iterations of PBKDF2 and SHA-256, with S1 as the salt. This ensures that K2 is the same as K1 if the correct password (P2 = P1) is used.
3. **Generate Hash for Authentication (H2)**
   Hash P2 again with K2 as salt with SHA-256 to generate H2.
4. **Authenticate User**
   Compare H1 (stored in the database) and H2 (calculated from the login password). If they match, the user is authenticated.
5. **Encrypt Data with K2**
   Once authenticated, K2=K1 is used to encrypt and decrypt all user data with AES-GCM-256.

Key Points

AES-256 GCM is used in the password manager to encrypt user data securely. AES-256 provides robust encryption with a 256-bit key, ensuring that sensitive information remains confidential. The GCM mode adds an authentication layer, which verifies both the integrity and authenticity of the encrypted data, protecting against tampering and unauthorized modifications.

PBKDF2 is employed to derive a strong encryption key from the user's password. By applying a pseudorandom function multiple times (250,000 iterations), PBKDF2 makes it computationally intensive for attackers to derive the encryption key through brute-force attacks.

SHA-256 used twice on the user password generates a unique, irreversible hash , which is stored and compared for authentication. This prevents passwords from being stored or transmitted in plain text, enhancing security by ensuring that the original password cannot be easily recovered or exploited.

Even though H1 is stored in the database, it is impossible to reverse-engineer K1 or the original password due to SHA-256's asymmetric nature. K1 is the AES-GCM key, and because it is derived from the user’s password, even if the backend is compromised, decryption is impossible without the user password which is used to derive the K1.

The security design ensures that even if the database is compromised, password cracking and data decryption require immense time and compute resources, making attacks impractical, even for the most advanced computers.

Security Design Diagram

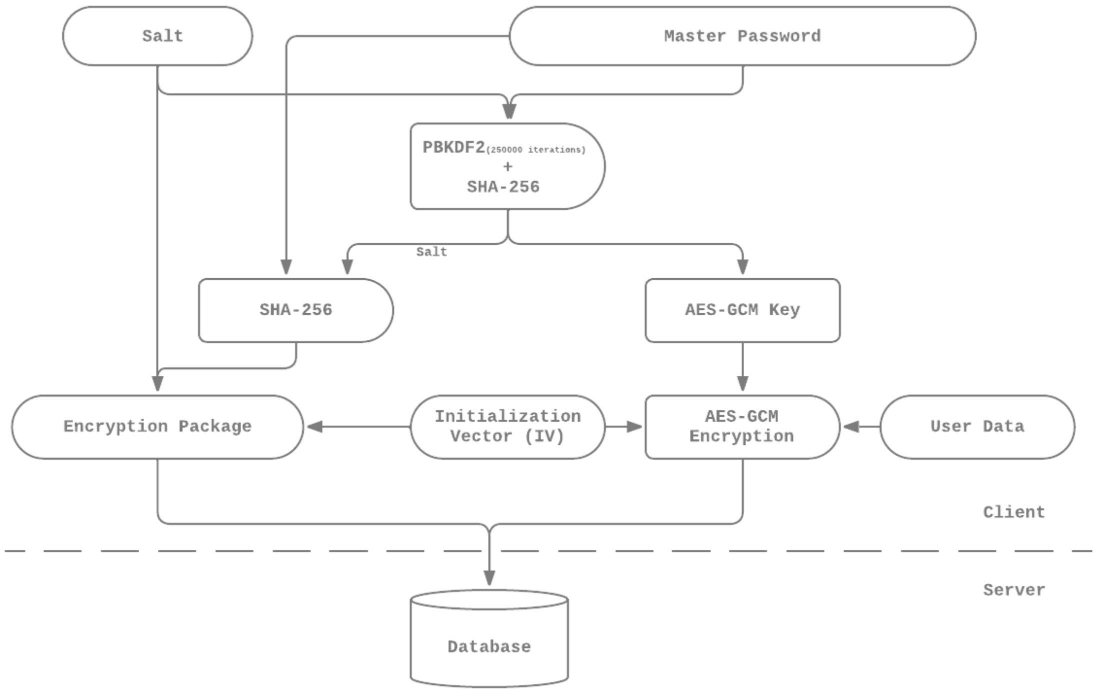

Diagram 1: SHIELD Security Design