The WhatsApp AppState Synchronization Protocol

1. Introduction

The WhatsApp AppState Synchronization Protocol is designed to efficiently and securely synchronize a user's application settings across multiple devices. This state includes data such as chat mutes, pinned chats, archived chats, and other user-configurable settings.

The protocol operates asynchronously, allowing a client to receive incremental updates ("patches") to its state or fetch a full "snapshot" of the current state from the server. It uses a combination of symmetric key encryption, MACs for integrity, and a Linear Time Hash (LTHash) to allow for efficient, order-independent application of state changes. This ensures that even if patches are received out of order, the final state will be consistent across all devices.

2. Overview

2.1. Terminology

- **Client**: A user's WhatsApp application on a specific device.
- **Server**: The WhatsApp server infrastructure that stores and distributes AppState information.
- **State**: A collection of key-value pairs representing application settings. The protocol groups related settings into distinct named states.
- **Mutation**: A single atomic change to the state, either setting a value (SET) or removing a value (REMOVE).
- **Patch**: A collection of mutations that represents an incremental update to a state.
- **Snapshot**: A complete representation of a state at a specific version, used for initial synchronization or recovery.
- **LTHash**: A Linear Time Hash, a type of homomorphic hash that allows for efficient verification of a collection of items. In this protocol, it is used to maintain a hash of the overall application state.

2.2. Cryptographic Primitives

The protocol relies on the following cryptographic functions:

- **HKDF**: HMAC-based Key Derivation Function using SHA-256 (as per RFC 5869).
- **HMAC**: Keyed-Hash Message Authentication Code using SHA-256 and SHA-512 (as per RFC 2104).
- **AES-256-CBC**: Advanced Encryption Standard with a 256-bit key in Cipher Block

Chaining mode for encrypting mutation values. A random 16-byte IV is prepended to the ciphertext.
- **SHA-256**: Secure Hash Algorithm with a 256-bit output.

2.3. State Types

The protocol manages several distinct types of state, referred to as "patch names". Each state is synchronized independently. Known patch names include:

- critical_block
- critical_unblock_low
- regular
- regular_low
- regular_high

3. Key Management

3.1. App State Sync Key

Each client maintains a set of AppStateSyncKey structures, each identified by a keyId. These keys are shared between a user's devices via an end-to-end encrypted protocol message. A new key is generated periodically. Each key consists of 32 bytes of secret key data.

3.2. Key Derivation

When performing cryptographic operations, a base AppStateSyncKey is expanded into a set of five derived keys using HKDF-SHA256 with the info string "WhatsApp Mutation Keys".

ExpandedKeys = HKDF-SHA256(salt=nil, inputKeyMaterial=SyncKeyData, info="WhatsApp Mutation Keys", outputLength=160)

The resulting 160 bytes are split into five 32-byte keys:

1. **Index Key** (ExpandedKeys[0:32]): Used to generate HMACs of mutation indices.
2. **Value Encryption Key** (ExpandedKeys[32:64]): Used as the AES-256 key to encrypt mutation values.
3. **Value MAC Key** (ExpandedKeys[64:96]): Used to generate HMACs of encrypted mutation values.
4. **Snapshot MAC Key** (ExpandedKeys[96:128]): Used to generate HMACs of state snapshots.
5. **Patch MAC Key** (ExpandedKeys[128:160]): Used to generate HMACs of patches.

4. State Representation

4.1. The HashState Object

Each client maintains a HashState for each named state (e.g., critical_block). This object represents the client's local view of the state and consists of two components:

- **Version**: A 64-bit unsigned integer representing the version number of the state. It is incremented with each applied patch.
- **Hash**: A 128-byte LTHash value representing a condensed cryptographic summary of all key-value pairs in the state.

4.2. The LTHash Function

The protocol's core synchronization mechanism relies on an LTHash. This allows the server to verify the client's state version without knowing the state's contents, and it allows the client to efficiently update its state hash.

The 128-byte state hash is treated as an array of 64 little-endian 16-bit unsigned integers. For each mutation, a corresponding 128-byte hash is derived. To update the state, the derived hash for the new mutation is added pointwise to the state hash, and the derived hash for the value being replaced (if any) is subtracted pointwise.

- LTHash.Add(state_hash, mutation_hash): Performs pointwise addition with overflow on each of the 64 16-bit integers.
- LTHash.Subtract(state_hash, mutation_hash): Performs pointwise subtraction with overflow on each of the 64 16-bit integers.

The value derived for each mutation is calculated as:
DerivedHash = HKDF-SHA256(salt=nil, inputKeyMaterial=MutationIndex, info="WhatsApp Patch Integrity", outputLength=128)
where MutationIndex is the raw JSON array of strings identifying the state entry (e.g., ["archive", "1234567890@s.whatsapp.net"]).

5. Data Structures

The protocol uses Protobuf-serialized structures for synchronization.

5.1. SyncActionData

This is the plaintext content of a mutation.

- **index**: A JSON-encoded array of strings that serves as the key for the state entry.
- **value**: A SyncActionValue Protobuf message containing the actual state change (e.g., MuteAction, PinAction).
- **padding**: Random bytes for padding.
- **version**: The version of the SyncActionValue schema.

5.2. SyncdMutation

A single mutation within a patch or snapshot.

- **operation**: The type of mutation, either SET or REMOVE.
- **record**: A SyncdRecord containing the mutation data.

A SyncdRecord contains:

- **index**: A SyncdIndex containing a 32-byte HMAC-SHA256 of the SyncActionData.index (the "Index MAC").
- **value**: A SyncdValue containing the AES-256-CBC encrypted SyncActionData, followed by a 32-byte HMAC-SHA512 of the encrypted data (the "Value MAC").
- **keyId**: The ID of the AppStateSyncKey used for the cryptographic operations.

5.3. SyncdPatch

An incremental update.

- **version**: The version of the state *after* this patch is applied.
- **mutations**: A list of SyncdMutation objects.
- **snapshotMac**: An HMAC-SHA256 of the client's expected HashState *before* applying this patch (the "Snapshot MAC").
- **patchMac**: An HMAC-SHA256 of the entire patch, used for integrity checking (the "Patch MAC").
- **keyId**: The ID of the AppStateSyncKey used.

5.4. SyncdSnapshot

A full state snapshot.

- **version**: The version of the state represented by this snapshot.
- **records**: A list of SyncdRecord objects representing the full state.
- **mac**: An HMAC-SHA256 of the HashState derived from this snapshot (the "Snapshot MAC").
- **keyId**: The ID of the AppStateSyncKey used.

6. Protocol Operations

6.1. Fetching State (Snapshots and Patches)

A client synchronizes a state by sending a request to the server.

- To get a **snapshot**, the client requests the state with return_snapshot=true. This is done for the initial sync.
- To get **patches**, the client sends its current version number. The server returns all patches that have occurred since that version.

6.2. Processing a Snapshot

1. The client receives a SyncdSnapshot.
2. It initializes an empty 128-byte HashState.Hash and sets HashState.Version to the snapshot's version.
3. For each SyncdRecord in the snapshot:
   a. It decrypts the SyncActionData from the record's value blob.
   b. It validates the Index MAC and Value MAC.
   c. It calculates the 128-byte derived hash for the mutation's index.
   d. It adds this derived hash to its local HashState.Hash using the LTHash function.
4. After processing all records, the client calculates the Snapshot MAC on its final HashState and compares it with the mac field in the snapshot. If they match, the snapshot is valid.

6.3. Processing a Patch

1. The client receives a SyncdPatch.
2. It uses its current local HashState to compute a Snapshot MAC and verifies it against the snapshotMac in the patch. If it mismatches, the client has a divergent state and must fetch a new snapshot.
3. The client validates the patchMac to ensure the patch itself has not been tampered with.
4. For each SyncdMutation in the patch:
   a. It decrypts the SyncActionData.
   b. It validates the Index MAC and Value MAC.
   c. If the operation is SET on a key that previously had a value, the client must retrieve the old Value MAC (from a local store) and subtract its corresponding derived hash from the state hash.
   d. If the operation is REMOVE, it does the same as the step above.
   e. If the operation is SET, it adds the derived hash of the new mutation's index to the state hash.
5. After applying all mutations, the client updates its local HashState.Version to the version specified in the patch.

6.4. Creating and Sending a Patch

1. The client determines the mutations it needs to send (e.g., user mutes a chat).
2. For each mutation:
   a. It constructs the SyncActionData Protobuf.
   b. It encrypts the serialized SyncActionData using the Value Encryption Key.
   c. It computes the Value MAC and Index MAC using the appropriate keys.
   d. It constructs the SyncdMutation.
3. It retrieves its current HashState for the relevant patch name.

4. It constructs the SyncdPatch, setting the snapshotMac by computing it from the current HashState.
5. It updates its local HashState by applying the LTHash operations for the new mutations.
6. It increments its local HashState.Version.
7. It computes the final patchMac for the SyncdPatch structure.
8. The patch is sent to the server.

7. Cryptographic Computations

Let keyId be the ID of the AppStateSyncKey in use, and ExpandedKeys be the derived keys.

7.1. Index MAC

IndexMAC = HMAC-SHA256(key=ExpandedKeys.Index, data=SyncActionData.index)

7.2. Value MAC

ValueMAC = HMAC-SHA512(key=ExpandedKeys.ValueMAC, data=op || keyId || encrypted_value || len_bytes)[:32]

- op: A single byte, 0x01 for SET, 0x02 for REMOVE.
- encrypted_value: The AES-256-CBC ciphertext of the SyncActionData.
- len_bytes: The length of keyId || op as a big-endian 64-bit integer.

7.3. Snapshot MAC

SnapshotMAC = HMAC-SHA256(key=ExpandedKeys.SnapshotMAC, data=HashState.Hash || version_bytes || patch_name_bytes)

- version_bytes: HashState.Version as a big-endian 64-bit integer.
- patch_name_bytes: The ASCII bytes of the state name (e.g., "regular").

7.4. Patch MAC

PatchMAC = HMAC-SHA256(key=ExpandedKeys.PatchMAC, data=snapshotMac || valueMac_1 || ... || valueMac_N || version_bytes || patch_name_bytes)

- valueMac_i: The 32-byte Value MAC of the i-th mutation in the patch.
- version_bytes: The new version number from the patch as a big-endian 64-bit integer.

8. Security Considerations

- **Confidentiality**: Mutation values, which can contain sensitive information like contact JIDs, are encrypted.
- **Integrity and Authenticity**: Every component (index, value, patch, snapshot) is

protected by an HMAC, preventing tampering by the server or a man-in-the-middle.

- **Consistency**: The LTHash and versioning system ensure that clients can detect if their state has diverged and can reliably converge to the correct state. An incorrect local state will lead to a Snapshot MAC mismatch, forcing a full resync.
- **Key Separation**: The use of HKDF to derive separate keys for encryption, index MACs, and value MACs ensures that the keys for different cryptographic operations are independent.

9. References

This document is based on analysis of the open-source whatsmeow library, an unofficial Go implementation of the WhatsApp Web API.