ChatGPT

Design Specification: Secure Session History Key Management Extension

Background & Motivation

Chrome's storage APIs do not encrypt data by default, meaning any stored extension data is readable in plain text [1]. Storing sensitive information (like user session history) without protection poses security risks. A known example is the Bitwarden browser extension which stored encrypted vault data **and** the encryption key together in plain storage – a practice described as a "horrible idea" [2]. To address these concerns, we need a robust client-side encryption strategy. The goal is to ensure **all session history data is encrypted at rest**, following a zero-knowledge approach where encryption/decryption happens only on the user's machine [3] [4]. This design will outline how to manage encryption keys, support an optional user passphrase ("passkey") for added security, and handle various scenarios (installation, passkey setup, sync, etc.) in a user-friendly yet privacy-first manner.

Goals & Requirements

- **Automatic Key Generation:** On first install, automatically generate a **random encryption key** (256-bit AES) to encrypt session history data. Store this key in extension storage (Chrome API) so it persists across sessions.
- **Encryption at Rest: Never store session history in plain text** on local storage. All session data must be encrypted using the encryption key before being written to disk. This ensures that even if someone accesses the extension's storage files, they cannot read history without the key [2].
- **Key Encryption with Passkey:** Allow the user to set an optional **passkey** (a master password). If set, the extension will **encrypt the stored encryption key** with a key derived from the user's passkey. This means the encryption key is never persisted in plain form once a passkey is in use.
- **Secure Key Storage:** By default, store the (unlocked or encrypted) encryption key in Chrome's **sync storage** so it syncs across the user's devices for convenience. Provide a setting (toggle) for privacy-conscious users to **disable cloud backup**; if disabled, the key remains only on the local machine (and is removed from sync storage).
- **Seamless Default UX:** Until a user explicitly sets up a passkey, the extension should operate **implicitly** – i.e., no extra prompts. The encryption is transparent (using the auto-generated key) so as not to burden the user with setup steps. This "implicit mode" trades some security (key stored plainly in sync) for ease of use, which is acceptable until the user opts into stronger security.
- **Passkey-Protected Mode:** Once a user sets a passkey, require this passkey on extension startup (or when needed) to **unlock** the encryption key. Without the correct passkey, the session history remains inaccessible (still encrypted). This aligns with password manager behavior where a master password must be entered to access data [5].
- **Local-First Data, Privacy Preserved: No session history data leaves the local machine.** The only thing ever stored in the cloud (Chrome Sync) is the encryption key (initially plaintext, later encrypted if passkey set). This ensures user history isn't uploaded or shared – only the key is synced for recovery. Data encryption and decryption occur **entirely client-side** [3] [4].
- **Reinstallation & Multi-Device Support:** If the extension is reinstalled or installed on a new device (with the same Chrome account), it should be able to **recover the encryption key** from

sync storage. This allows it to decrypt any existing local encrypted history (e.g. if the local data files were not removed) and continue appending to history. However, if the local data files are missing (e.g. extension was uninstalled or we're on a new device with no prior data), the history can't be recovered – which is an expected trade-off for not syncing actual data.

- **Edge Case – Lost Passkey:** If the user forgets their passkey or chooses not to enter it when prompted (on a device where the key is present but encrypted), the extension should allow **starting fresh** with a new key. In other words, the user can opt to generate a new plaintext key (losing access to old encrypted history) so they can continue using the extension. This provides a fallback if passkey access is lost.
- **User-Friendly & Fault-Tolerant:** The design must consider various scenarios and errors to ensure a smooth user experience:
- Minimal prompts in the default flow.
- Clear, non-technical prompts when asking for the passkey.
- Options to reset or recover (if possible) when errors occur – e.g., "Reset extension data" if the user cannot recall the passkey.
- Prevent data loss wherever possible, but prioritize security (if user forgets passkey, the data is effectively unrecoverable by design).

Key Concepts & Components

- **Session History Data:** The sensitive user data (e.g. browsing session info or whatever the extension records as "sessions"). This is stored in local storage (e.g. `chrome.storage.local` or an IndexedDB) on the user's machine. Every record in this history will be encrypted using our encryption key before storage.
- **Encryption Key (Master Key):** A randomly generated symmetric key (suggest use of 256-bit AES) that actually encrypts and decrypts the session history data. This is the **core secret** that must be protected. It is generated on first run and persisted for the lifetime of the extension's data (so that the same key can decrypt old history). Think of this as the "data key" or "vault key" for the session history.
- **Passkey (User Passphrase):** An **optional** user-provided password that adds a layer of security. When a passkey is set, the encryption key will be **encrypted ("wrapped") with a key derived from this passkey**. The passkey itself is *never stored*; it's only entered by the user when needed and used to derive a cryptographic key in-memory (using a KDF, see below).
- **Chrome Storage (Sync vs Local):** We leverage Chrome's `chrome.storage`:
- `chrome.storage.sync` (Cloud Sync Storage): Used to store the encryption key (either in plaintext or encrypted form). Sync allows the key to propagate to the user's other Chrome installations (if logged in) and persists through reinstallation (since sync data is tied to the user account, not the extension installation). **Note:** Chrome sync data is not encrypted by default on Google's servers [6] [7], which is why storing a plaintext key here has some risk – we mitigate that by allowing user to set a passkey or opt out of sync.
- `chrome.storage.local`: Used to store session history records (encrypted). This data remains on the local machine and is cleared if the extension is uninstalled [8]. We prefer local for data due to its 10MB+ capacity and because we *don't* want to sync large history data to cloud.
- `chrome.storage.session`: We may use the `session` storage area for ephemeral sensitive values (e.g. keeping the decrypted encryption key in memory while the extension is running, if needed) [9]. This ensures that if the browser is closed, such sensitive data isn't written to disk.
- **WebCrypto API for Encryption:** The extension will use the Web Cryptography API for all cryptographic operations:
- **AES-GCM** for encrypting session history content with the encryption key. AES-GCM is chosen for its authenticated encryption (integrity check) and efficient in-browser support. Each encrypted record or blob will use a unique Initialization Vector (IV) for security.

- **Key Derivation (PBKDF2 or similar)** for deriving an encryption key from the user's passkey. Using PBKDF2 (with a strong hash like SHA-256, and a high iteration count, e.g. 100k) will convert the user's passphrase into a 256-bit key. We will use a random **salt** (16 bytes) for PBKDF2, which will be stored alongside the encrypted key (salt is not secret, but it ensures that two users with the same passkey get different derived keys) [10] . This derived key will then be used to AES-encrypt our encryption key.
- The **Web Crypto API** can generate random values (for the encryption key and IVs) and perform encryption/decryption securely within the extension context.
- **Encryption Key "Wrapping":** When user sets a passkey, the process of encrypting the encryption key with a key derived from the passkey is often called key wrapping. We will produce an **"Encrypted Key Blob"** – for example, an AES-GCM ciphertext of the encryption key. This blob (along with metadata like the PBKDF2 salt and IV used for wrapping) is what gets stored in `chrome.storage.sync` in place of the plaintext key.
- **State Flags:** We'll maintain some indicator of the state of the key in storage, such as a boolean flag `keyEncrypted=true/false` or storing the key entry in a structured way:
- For instance, in `chrome.storage.sync`, we could have:
    - `encryptionKey` – if using plaintext key, this holds the raw key bytes (e.g. Base64-encoded).
    - `encryptedKey` – if using passkey, this holds the encrypted blob of the key (Base64).
    - `keySalt` – the PBKDF2 salt (if passkey-derived).
    - `keyIV` – the IV used to encrypt the key (if using AES-GCM for wrapping).
    - `usingPasskey` – a flag to denote that the key is encrypted and a passkey is required.
- Alternatively, a single object could be stored with fields indicating whether the key is locked or not. This helps on startup to quickly decide if we must prompt the user.

High-Level Design Overview

At a high level, the extension will operate in two modes: 1. **Implicit Mode (No Passkey Set):** The extension generates and uses an encryption key automatically. The key is stored **in plaintext in Chrome sync storage** (by default) so that it can be retrieved across sessions/devices. All session data is encrypted with this key before storing locally. The user is not prompted for anything in this mode – everything works behind the scenes. 2. **Passkey-Protected Mode:** The user has set a passkey, so the encryption key in storage is **encrypted with the passkey**. On extension startup, the key is not immediately available – the extension enters a "locked" state and prompts the user to enter the passkey. Only after the correct passkey is provided do we decrypt the encryption key in memory and proceed to use it for reading/writing session data. In this mode, the encryption key is never written or transmitted in plain form. Session data encryption remains the same (still using the encryption key), but access to that key is gated by the passkey.

All data encryption/decryption happens on the client side. Even when data (or the wrapped key) is synced or backed up, it remains encrypted until it reaches a client with the proper key/passkey to decrypt [4] . This ensures **zero knowledge** for any external entity – for example, if someone somehow accessed the user's Chrome sync data on Google's servers, they would only see gibberish (either a random key or an encrypted blob) and could not read the user's history.

Below, we detail the workflows for various scenarios and how the system behaves.

Workflow Scenarios & Lifecycle

1. First-Time Installation (No Existing Key)

Upon initial installation (or if no key is found in storage): 1. **Key Generation:** The extension will generate a new **256-bit random encryption key** using `window.crypto.subtle.generateKey` (AES-GCM algorithm). This key will be used to encrypt session history entries. 2. **Store Key (Plaintext):** Since this is a fresh start with no passkey, we store this key directly in `chrome.storage.sync` (e.g. as a Base64 string). We also set a flag (e.g. `usingPasskey = false` or simply do not set an `encryptedKey` field) indicating the key is stored in plaintext form. 3. **Local Data Setup:** Initialize the local storage for session history (e.g. set up an empty data structure in `chrome.storage.local` or an IndexedDB). This will hold encrypted history records going forward. 4. **No User Interaction:** All of this happens behind the scenes. The user is not asked for any input on first install. From their perspective, the extension "just works." The session history feature is ready to use, with data being encrypted transparently using the generated key.

Rationale: We want initial onboarding to be frictionless. Many users may not want to deal with a passphrase initially. By generating a key automatically, we ensure some security (history not stored in plain) and lay the groundwork for users to enable stronger security later. However, we do acknowledge that the plaintext key in sync is a weak point – albeit protected by the user's Google Account security and local OS security. Users concerned about this can proceed to enable a passkey.

2. Extension Startup with Plaintext Key (Returning User, No Passkey)

This scenario covers when the extension loads and finds a plaintext key in storage (meaning the user has been using the extension without a passkey): 1. **Retrieve Key:** The background script (or service worker) will call `chrome.storage.sync.get('encryptionKey')`. The expected result is the Base64-encoded encryption key (since no passkey was set). 2. **Key Found:** If the key is present, the extension decodes it to a CryptoKey object (via `subtle.importKey`) for use. Since it's plaintext, no decryption is needed. The flag `usingPasskey` would be false. 3. **Decrypt Local Data (if needed):** On startup, if the extension needs to load existing session history (e.g. to display it), it will fetch encrypted records from `chrome.storage.local` and decrypt them using the key. Because the key is readily available, this can happen immediately and seamlessly. 4. **Continue Operation:** The extension continues to log new sessions, encrypting each entry with the key and writing to local storage. No prompts are shown to the user. 5. **Offer Security Upgrade (optional):** For better security, the extension's UI might gently prompt or inform the user about setting a passkey. (For instance, a message like "Protect your session history with a passkey" could be shown in the options page or popup.) This is not mandatory, but users who want privacy can choose to enable it. Until they do, everything remains in implicit mode.

3. Extension Startup with Encrypted Key (Passkey-Protected)

This scenario occurs if the user had previously set a passkey. The extension (on this or another device with sync) will find an **encrypted key blob** in storage instead of a plaintext key: 1. **Detect Encrypted Key:** On startup, the extension does `chrome.storage.sync.get(['encryptedKey', 'keySalt', 'keyIV'])` (or a similar structure). It finds that `encryptedKey` exists (and likely a flag indicating `usingPasskey=true`). This tells the extension that the encryption key is locked/encrypted. 2. **Enter "Locked" State:** Since the actual encryption key can't be obtained without the user's passkey, the extension transitions to a locked state. In this state: - The UI should indicate that the session history is locked (for example, the extension's browser action icon could show a "locked" icon, or if the user opens

the extension popup, it shows a passkey prompt rather than the history). - The extension **temporarily suspends recording new sessions** until unlocked (because it has no key to encrypt them with). Alternatively, as a fallback, it could use a temporary new key (see next point), but by default, it's safer to hold off on processing sensitive data until unlock. 3. **Prompt User for Passkey:** A prompt is presented for the user to enter their passkey. This could be a dedicated **login page or popup** of the extension. It should be a secure input field (masked) and possibly offer an option to show the password for convenience. User-friendliness is key: include messaging like "Enter your passkey to unlock your session history." 4. **Passkey Entry & Verification:** When the user enters the passkey: - The extension will retrieve the stored `keySalt` (and knows the PBKDF2 parameters like iterations, hash) and use `window.crypto.subtle.importKey` + `deriveKey` (PBKDF2) to derive the **key-encryption key** from the passkey [10]. - Using the derived key and the stored `keyIV`, the extension calls `subtle.decrypt` (AES-GCM) on the `encryptedKey` blob. - **Correct Passkey:** If the decryption succeeds (authentication tag verifies), we obtain the plaintext encryption key. The extension now has the real encryption key in memory (as a CryptoKey). - **Incorrect Passkey:** If decryption fails (throws an error), the extension knows the passkey was wrong. It should *not* crash or clear data – instead, notify the user that the passkey was incorrect and allow retry. For security, we might lock out after a certain number of attempts or introduce a short delay after multiple failures, but given data is local, the threat is someone manually trying many guesses on the device. We can rely on the strength of PBKDF2 (100k+ iterations) to slow brute force and optionally implement exponential backoff on retries for user experience. 5. **Unlock & Load Data:** Once the correct passkey is provided and the encryption key is decrypted: - The extension transitions to an **unlocked state**. The UI now shows the session history (which can be decrypted with the key) as normal. - The extension decrypts existing session records from local storage using the key (same as in scenario 2). - The extension can resume normal operation: encrypting any new sessions with the key and storing them. - Importantly, the encryption key remains only in memory; we do **not** re-store it plaintext anywhere persistent. It remains available in the background script's runtime. If the extension is disabled or browser closed, memory is lost and the user will need to unlock again next time – this is by design for security, similar to how password managers require re-login each session. 6. **Temporary Key Fallback (Edge Case):** The specification mentioned *"until user enters passkey – generate a new plain stored key in Chrome API."* This could be interpreted as a fallback mode if the user delays entering the passkey. For example, if the user chooses not to unlock (or cancels the prompt), the extension might allow usage by generating a fresh key: - This new key would be treated as a *separate* session (like starting over). It would be stored plaintext in sync (since the user hasn't provided the passphrase, we can't encrypt it with one). - The extension would then use this new key for any *new* session data, effectively not giving access to the old history (which remains encrypted under the old key). - **However, this approach has consequences:** The old history becomes inaccessible until the user eventually enters the passkey (if ever). We would have two parallel histories (old locked data, and new data under the new key). If the user later remembers the passkey and unlocks, we'd have to decide how to merge or handle the two sets of data. - **Recommendation:** *Do not generate a new key automatically unless absolutely necessary.* A better UX is to encourage the user to unlock or explicitly reset. Automatically starting a new history could confuse users or lead to data fragmentation. Instead, we could offer: - A "Reset history" button after X failed attempts or if they indicate they forgot the passkey, which **clears the old encrypted data** and generates a new key to start fresh. - Proper warning that this will discard the old history permanently. - In summary, the extension primarily should stay locked until passkey is entered. Only if the user explicitly opts to abandon the old data (due to lost passkey) do we generate a new key.

4. User Sets a Passkey (Enabling Encryption on an Existing Key)

This scenario is when a user who has been in implicit mode (no passkey) decides to add a passkey for security. This could happen via a settings page or prompt in the extension UI: 1. **Initiate Passkey Setup:** User navigates to a "Set Passkey" option (perhaps in extension options). They are prompted to

choose a strong passkey. Ideally, we ask for it twice to confirm (to avoid typos) and maybe provide an estimate of its strength. 2. **Derive Key from Passkey:** When the user confirms: - Generate a random PBKDF2 salt (if not already existing). - Use `crypto.subtle.importKey` and `deriveKey` with PBKDF2 (100k iterations, SHA-256) on the entered passkey to derive a key-encryption key. 3. **Encrypt the Encryption Key:** - Retrieve the current plaintext encryption key (from memory or from sync storage). Since the extension was running and had the key (the user's data was accessible), we have it available – likely stored as a CryptoKey or we can retrieve it from sync one last time. - Use an AES-GCM encryption with a new random IV to **encrypt the encryption key** with the derived key. This produces an `encryptedKey` blob and an auth tag. 4. **Update Storage:** - Save the `encryptedKey` blob to `chrome.storage.sync`. Also save the PBKDF2 salt and IV (and perhaps iteration count, though that can be constant and known in code). - **Remove Plain Key:** Remove the old plaintext key entry. For example, call `chrome.storage.sync.remove('encryptionKey')`. Ensure no plain copy remains in any storage. From this point on, the sync data will only have the encrypted blob. - Mark a flag like `usingPasskey = true` (if we use such a flag) or otherwise the presence of `encryptedKey` signals that. - Also, in any local caches or variables, update state to indicate the key is now passkey-protected. 5. **Encrypt Existing Session Data (Re-encryption, if needed):** - Since we have not changed the actual encryption key (we only wrapped it), all existing session data remains valid (it's still encrypted with the same key). **Thus, no re-encryption of session history contents is strictly required** – they can already be decrypted with the key we have. - However, consider security: previously, the encryption key was stored in plaintext, so it *might* have been exposed. For maximal security, we could opt to **rotate the encryption key** at this moment: - Generate a *new* random encryption key (Key_2). - Decrypt all session history entries with the old key (Key_1) and immediately re-encrypt them with Key_2. - Use Key_2 going forward for all encryption. - Wrap Key_2 with the user's passkey (replace the blob in sync with an encrypted Key_2, and discard Key_1 entirely). - This ensures that even if Key_1 was ever compromised, it can no longer decrypt any data (because data is now under Key_2). **Drawback:** This operation might be time-consuming (if history is large) and prone to error (if something interrupts the process). It also means all devices must get the new key to read data. - Given the complexity, we can **choose not to rotate keys** on passkey setup. We simply protect the existing key. We will clearly log that from this point on the key is secured. If an attacker had access to the plain key before this moment, the horse has left the barn – but setting the passkey at least prevents future exposures. - The phrase "re-encrypt session history" in requirements seems to imply ensuring everything is encrypted. If up to now we already encrypted data (which we did), there is no action needed. If, hypothetically, the extension had stored some unencrypted data prior, we would encrypt it now. But since *"we never store plain session history"* is a requirement, this step is just a sanity check. 6. **User Feedback:** After successful setup: - Inform the user that their session history is now protected by the passkey. Maybe instruct them that they will need to enter this passkey when they reopen the browser or after a certain period of inactivity (if we implement an auto-lock timer). - Possibly give an option like "Log out now" to test the lock or allow them to continue in the current session unlocked. 7. **Cloud Sync Consideration:** At this point, the key in Chrome sync is the encrypted blob. On other devices: - When they sync, they will replace the old plain key with the encrypted blob. Those other devices' extensions (if running) should detect this change. We need to handle the transition: - If the extension on device B is currently running with the old plain key in memory and receives an update that now there's `encryptedKey` and its `encryptionKey` entry is gone, it should prompt the user to enter the passkey to re-lock. Ideally, we roll this out such that the user manually enables passkey on one device and then enters it on each other device as needed. - Device B could see the `usingPasskey=true` flag and know to prompt for passkey. If the user enters the same passkey there, it will derive the key and decrypt the blob, obtaining the same encryption key, so it can continue working (no data re-encryption needed as both devices share the same data key). - We should note in docs that the user needs to set the *same* passkey on all instances (really, it's automatically the same since the blob is synced – they just have to know it). 8. **Remove Key from Memory:** One more clean-up – if during this process we had the plaintext key in a variable from sync, and especially the user's plaintext passkey, ensure to clear those from memory (overwrite or let

variables go out of scope) after use, to reduce lingering secrets in memory. This is more of a low-level detail (browsers might not easily let us control memory, but we can dereference objects).

After this flow, the extension is now in **Passkey-Protected Mode** (as described in scenario 3 for subsequent startups).

5. Removing/Changing the Passkey

The spec doesn't explicitly cover removing or changing the passkey, but for completeness: - **Changing Passkey:** This would involve the user providing the old passkey (to decrypt the key) and then providing a new passkey: 1. Decrypt current encryption key with old passkey (if not already in memory). 2. Derive key from new passkey, encrypt the encryption key with it (new blob, possibly new salt). 3. Update storage with the new encrypted blob (and salt, etc.). Now the new passkey is required going forward. 4. Communicate to user that passkey changed. Other devices will need the new passkey. - **Removing Passkey (Reverting to implicit):** This would make the key plaintext again – **not recommended** from a security standpoint, but if offered: 1. User enters current passkey to unlock. 2. Retrieve plaintext encryption key. 3. Store the key back to `chrome.storage.sync` as plaintext (`encryptionKey` field), and remove the `encryptedKey` and related fields. 4. Now the extension no longer requires a passkey at startup. 5. This should perhaps warn the user: "Your history will no longer be protected by a passkey. This could make it accessible to anyone who gains access to your Chrome account or filesystem." Given our privacy focus, we might choose **not** to allow removing the passkey once set, to avoid users accidentally downgrading security.

6. Cloud Sync Toggle (Disabling Cloud Backup)

By default, we put the encryption key (plain or encrypted) in Chrome's sync storage (i.e., it's synced to the user's Google account). We provide a **toggle in settings** like "Backup encryption key to cloud (Chrome Sync) [On/Off]". The behavior: - **When user turns *Off* the cloud backup:** - If a key (plain or encrypted) is currently in `storage.sync`, we should **remove it from sync** storage. Use `chrome.storage.sync.remove([...])` for the relevant fields. - Instead, store it in `chrome.storage.local` (which does not sync). However, storing plaintext key in local is similar risk if someone has file access, but at least it's not on Google's servers. If a passkey is set (encrypted key), storing that locally is fine (still encrypted, but an attacker with local file access could also grab the encrypted blob; the difference is just not syncing to other devices). - Mark some flag like `cloudBackup=false` (so we remember user preference). - **Important:** Warn the user that if they disable cloud backup: - If they install the extension on another device or if they reinstall Chrome/ extension, **the key will not be available** to recover data. They would need to manually transfer the key or accept losing history. - Essentially, their session history becomes tied to this device only. If they lose the local data (device crash or extension uninstall), the history is gone permanently (privacy vs backup trade-off). - This might be acceptable for users who prioritize privacy and are okay with local-only data. - **When user turns *On* cloud backup (re-enabling):** - If the key is currently only local, we will push it to `chrome.storage.sync`. - If no passkey: we will sync the plaintext key (basically the same initial step as first install). - If passkey protected: we sync the encrypted blob and related fields. - Essentially we ensure the sync storage reflects whatever the current key state is. - After enabling, that key could propagate to other devices, allowing them to unlock the same history (if passkey provided). - **Default On:** We default this to ON because it provides a safety net (especially in plaintext mode it allows cross-device or reinstall recovery of data). It's also analogous to how Chrome by default syncs extension storage for convenience. Privacy-conscious users can opt out explicitly (and indeed some might if they use Chrome's own passphrase feature or simply don't trust cloud [7]). - **Cleaning Cloud Data:** The requirement says "leave a toggle to disable it and **clean**." Cleaning implies we should delete the key from Google's servers when turned off. So yes, we remove it from sync. (We might also want to call

`chrome.storage.sync.clear()` if we only stored key there to ensure no traces, but removing specific fields is sufficient.) - **Edge Case – Passkey + Cloud Off on new device:** If a user had a passkey set and then disabled cloud backup, then on a **new device** installation, there is *no key available* (since we didn't sync it). The extension will think it's a fresh install (no key) and generate a new key – effectively the user's history from device A cannot be accessed on device B. This is expected because they chose not to sync the key. We should document that if they want multi-device use, keep cloud backup on (or manually transfer the key somehow, which is advanced and not covered here).

7. Extension Reinstallation / Data Recovery Scenarios

- **Reinstall on Same Browser Profile (Cloud Backup On):** If a user removes the extension and later reinstalls it (using the same Google account with sync):
- Chrome *does* clear `storage.local` on uninstall, so the local encrypted session data is gone (unless we had stored it in some external file, which we do not in this design). Therefore, even if we recover the key from sync, there's no data to decrypt. The history is effectively lost due to uninstall (the key doesn't help because the data was removed) [8] .
- However, if the user did not fully uninstall – say they disabled the extension or cleared browser cache – the local storage might still exist. In such a case, on re-enable, the extension can retrieve the key from sync and decrypt the remaining data. The key is crucial to reading whatever data is left on disk.
- Therefore, the statement "if you reinstall extension – it can pick up local encrypted files, decrypt it, and render history" holds true *only if the local encrypted files still exist*. Typically after uninstall they won't, but after a temporary disable or if the user manually backed up the `chrome.storage` files, it could be possible.
- In practice, our focus is on *multi-device continuous usage* rather than uninstall/reinstall, because uninstall is destructive. We can clarify to the user that uninstalling will clear local history (unless they have sync and later re-import the actual data somehow).
- **Installing on a New Device (Cloud Backup On):**
- If the user installs the extension on a second device with the same Chrome account, Chrome Sync will deliver the stored key (after a short time, assuming sync is enabled).
- The extension on the new device will start up, find the key in sync:
    - If it's plaintext (user had no passkey on device A), device B gets the plaintext key. It can immediately use it. But note: device B has no session history data initially (that wasn't synced). So device B effectively starts with an empty history. It will use the same key to encrypt its own new sessions. **The benefit** is if later the user somehow moves the data or if we implement optional data sync via other means, the key is consistent. Also, if the user goes back to device A, both are using the same key so one could theoretically merge data from both devices if combined.
    - If it's encrypted (user had a passkey), device B will find an `encryptedKey` and know it needs the passkey. It should prompt the user for the passkey. Once entered, it derives and decrypts to get the key. Now device B has the same encryption key as device A in memory. Device B again has no prior data, but any new sessions it logs will be encrypted with the key. The user won't actually see device A's history on device B unless they manually exported/imported it (we aren't syncing history). So effectively, each device's history stays separate. This design is *local-first*: data doesn't sync, so multi-device use means each device has its own local history silo, albeit encrypted with the same key.
    - Having the same key on multiple devices is somewhat moot if data isn't shared, but it could allow the user to manually combine data if they wanted. More importantly, it allows continuity if the user *moves* from one device to another (e.g. stops using old device – they could copy over the local data file and since key is the same, decrypt it; but that's advanced usage).

- If the user has **Cloud Backup off**, on a new device it will be treated as a fresh install with no key (scenario 1), so a new key is generated and a new history starts. The histories are totally separate. This is expected with local-only keys.

- **Browser Profile Change:** If a user is logged out or uses a different Chrome profile, sync won't provide the key. The extension would treat it as new. We assume same profile for sync continuity.

- **Data Persistence:** To maximize chances of recovery:

- We might consider storing session history in a more persistent way (e.g., using the Chrome Sync FileSystem or a user-designated file). But since the requirement is local-first and not to upload history, we stick with `chrome.storage.local`. Users who want to preserve history should not uninstall the extension or should export data manually if we provide that feature.

Data Encryption & Storage Details

Session Data Encryption: Each session entry (or the entire history blob) will be encrypted with AES-GCM using the encryption key: - We will generate a fresh 96-bit IV for each encryption operation (e.g. if encrypting each entry separately, each gets its own IV; if encrypting one big blob of all history, we would re-encrypt the whole blob on each update with a new IV – likely less efficient). - The ciphertext along with the IV (and GCM auth tag, which is typically appended to ciphertext) will be stored in `chrome.storage.local`. For example: - If per-entry encryption: store an array of objects, each with `{ iv: "<base64>", data: "<base64-cipher>" }`. - If one blob: store something like `historyBlob = <base64 of IV + cipher + tag>`. - A decision point is whether to encrypt entries individually or as one combined dataset: - **Per-entry encryption:** Easier to append new entries without re-encrypting everything. Each entry can be decrypted independently. This is good if we frequently add data. Downside: searching or slicing the history might require multiple decryptions (but that's fine, typically small overhead). - **All-in-one encryption:** Simpler in terms of storage (just one blob), but updating means decrypting, modifying, and re-encrypting the entire set, which can be slow as history grows. - We lean towards *per-entry encryption for flexibility*. We'll just carefully store and manage the list. - **Integrity:** AES-GCM provides authentication, so if data is tampered, decryption will fail. We should handle a decryption error on a history entry by, say, logging an error and possibly discarding that entry as corrupted (this could happen if storage got corrupted or an attacker flipped bits). - **Metadata:** We might also maintain some metadata like a version number of the encryption scheme or the last rotation of key, etc., in case we ever need to migrate. For now, a simple approach is fine since we have a single scheme.

Key Storage Formats: - Plaintext key in sync: We can store it as Base64 string for ease (since chrome.storage can only store JSON-serializable data, raw ArrayBuffer needs encoding). - Encrypted key blob: Also store as Base64 string (of the cipher bytes + tag). We will have separate fields for salt and IV (Base64 as well). - We must ensure not to accidentally log these values or expose them in any UI.

Local vs Sync Quota: - `chrome.storage.sync` has size limits (~100KB total, ~8KB per item) [11]. Our key and metadata are just a few bytes, so that's fine. - `chrome.storage.local` has ~10MB by default, which should suffice for a lot of text history. If we anticipate more, we might request `unlimitedStorage` permission as noted in Chrome docs [8], so heavy users don't get cut off. - If storing per entry, we should be mindful that too many small writes to sync can throttle. But since we don't write to sync often (only when key changes or initial install), that's fine. Local writes are not as strictly throttled.

Memory Handling: - When unlocked, the plaintext encryption key will reside in memory (the extension's background script context). We should avoid overly exposing it. It will be used whenever we encrypt/decrypt. - After use, we could call `crypto.subtle.exportKey('raw')` and then zero out the ArrayBuffer if we want to explicitly clear it. JavaScript doesn't give us manual control of memory for a CryptoKey object, but if security is critical, we might do cryptographic operations in a lower-level way to ensure no lingering plaintext. However, given this is an in-memory concern and assuming the user's machine isn't compromised, it's an acceptable risk (most password managers also hold keys in memory while unlocked [12]). - The user's passkey is only used transiently to derive keys. We should not store it. If we have a settings page that asks for current passkey (for verifying identity when changing settings like removing passkey), we should take care to not keep it around.

User Experience Considerations

To make the feature **maximally user-friendly** while preserving security: - **Transparent Default:** Users who do not care to set up a passkey should not feel any difference in using the extension. They install it and it just works (their data is quietly encrypted in the background). Only an advanced user checking Chrome's internal storage would even know a key exists. - **Education and Encouragement:** Provide non-intrusive prompts or info sections about the benefits of setting a passkey. For example, an "🔒 Secure your data with a passkey" banner in the UI, which can be dismissed. Emphasize that without a passkey, someone with access to their Google account or filesystem could potentially read their session data (since the key is stored plainly) [2]. With a passkey, only they can unlock it. - **Setting Passkey Flow:** When the user opts to set a passkey, ensure the flow is clear: - Show warnings about remembering the passkey ("If you forget this passkey, you will lose access to your session history. There is no 'forgot password' recovery."). - Possibly allow the user to optionally **hint** or remind themselves (e.g., a hint field stored plaintext, but user-provided). Or suggest them to use a strong password manager to store it. - Use quality feedback: e.g., a strength meter, or rules (minimum length, etc.) to encourage a strong passkey. - Confirm passkey by double-entry to avoid mistakes. - Upon success, maybe show a message: "Passkey set! You will be asked to enter this when you reopen the browser or use the extension on other devices." - If user cancels mid-process, make sure not to leave things half-changed. - **Unlock Prompt:** When the extension is locked (passkey mode on startup): - The extension's browser action icon might display a badge or lock symbol to indicate locked status. - Clicking it should immediately show the passkey prompt (so user can unlock). Possibly allow pressing Enter to submit, etc. - If an incorrect password is entered, display an error message in the prompt "Incorrect passkey, please try again." Don't reveal how many characters were correct or any specific clues. - If multiple failures, you might slightly shake the input field or clear it for retyping. - Ensure the user cannot bypass the prompt to see data – e.g., disable any context menu or secondary features that might reveal something until unlocked. - **Locking Behavior:** Consider if the extension should **auto-lock** after a period of time for added security. Some password extensions lock after the browser is idle or after a set timeout. For simplicity, this spec doesn't enforce auto-lock beyond the natural lock on browser restart. But we could add an option "Lock after X minutes of inactivity" as a later feature. - **Reset Option:** In case the user can't unlock (forgot passkey): - Provide a "Reset data" button on the lock screen with a clear warning. If clicked, confirm ("Are you sure? This will delete your saved session history since you cannot unlock it."). - If confirmed, proceed to wipe `chrome.storage.local` (history) and `chrome.storage.sync` (the encrypted key blob), effectively factory resetting the extension data. Then generate a new key as in first-time flow. - This ensures the user can continue using the extension (albeit with lost history) rather than being permanently locked out. It's critical to communicate the consequence. - **Visual Indicators:** In settings, show the status of the key: - e.g., "Encryption: **Enabled (Passkey protected)**" or "Encryption: **Enabled (key stored in cloud)**" or "Encryption: **Enabled (local-only)**" depending on their configuration. This transparency builds trust that encryption is working. - If local-only, maybe also indicate "(No cloud backup – data tied to this device)". - **Testing & Fault Tolerance:** We must test scenarios such as: - Passkey setup and immediate browser crash – ensure next startup can still prompt and accept the

passkey (the key blob was written properly). - Sync conflicts: If two devices independently generate a new key (could happen if both were offline and user sets passkey on one while the other wasn't updated?). To mitigate, when in passkey mode, the key should only be changed by explicit user action. We could use `chrome.storage.onChanged` to detect if a key change happens from sync and handle it. - If a user quickly toggles cloud backup or rapidly sets/clears passkey, ensure our code doesn't get confused. Typically these actions will be rare. - Backwards compatibility or version updates: If we release this feature as an update, existing users might already have unencrypted history stored (contradicting "never store plain"). In that case, the extension update should detect if any plaintext history exists and migrate it (encrypt it with newly generated key). This is more an implementation detail if relevant.

Security Considerations

- This design ensures that **at rest, all sensitive data is encrypted**. Even though Chrome storage is not encrypted by Chrome [1], our manual encryption protects the data.
- Initially, the encryption key is stored plainly for usability. This means the user's data confidentiality relies on the security of their environment (Google account, Chrome's cloud storage, and local file system). For many users, this is acceptable (Chrome sync data is somewhat protected by account credentials). For high-security use, the user should set a passkey.
- Once a passkey is set, the extension achieves a higher level of security:
- The data key is encrypted with user's passphrase which only they know. So even if an attacker dumps the extension storage (cloud or local), they cannot decrypt the history without brute-forcing the passphrase (which, if strong and PBKDF2-hardened, is computationally infeasible).
- This essentially becomes a **zero-knowledge encryption model**: only the user with the correct passkey can read the data [13]. Not even Google (who holds the sync blob) or the extension developer can access it.
- The passkey itself is never stored or transmitted – only a salted hash via PBKDF2 in memory is used to check the key. This aligns with best practices (similar to how master passwords derive keys in password managers [14]).
- We should guard against common threats:
- **Shoulder surfing**: If someone is watching when the user types passkey, that's beyond our scope except maybe providing an eyeball icon to toggle hide/show the password if needed.
- **Memory dump attacks**: If malware is on the user's machine with the ability to dump extension memory, it could theoretically steal the key when unlocked (or capture keystrokes of passkey). This is true for any in-memory encryption – at that point, the system is compromised. Our focus is on protecting data at rest and from less-privileged threats.
- **Browser vulnerabilities**: If a malicious site could access extension storage (normally not possible unless XSS in extension or a compromised content script), we ensure even if they read `chrome.storage.local` or `sync`, they get encrypted gibberish [2].
- **Brute force**: Using a strong KDF (100k iterations PBKDF2) and urging a strong passkey mitigates brute force attempts on the encrypted key. We can also intentionally not provide any easy way to programmatically brute force via the extension UI (no API to test passkey except user input).
- **Privacy:** No session data leaves the device. The only thing that does (the key) is either innocuous (random string) or encrypted with user's secret. This complies with a local-first, privacy-respecting philosophy. Even if compelled to turn over data, without the user's passkey the data is not intelligible.

Edge Cases & Error Handling

- **No Key Found on Startup:** (Should only happen on first install or if storage was manually wiped) – handle as fresh install (generate new key). If for some reason `chrome.storage.sync` read fails (temporary issue), we might retry or fall back to generating a new key and marking it (could

result in multiple keys if sync later returns old one – to avoid confusion, maybe check for sync conflict: e.g., if we generate a key but then sync returns an old encrypted key, prefer the encrypted key and discard new data).

- **Multiple Keys in Sync (Conflict):** Unlikely, but if, say, the user used extension without passkey on two devices offline (each generated a different key) and then both synced – perhaps last write wins. We should treat whichever is present as authoritative and possibly alert user if data inconsistency is detected. But given sync merges values per key name, having two separate keys might not both survive. We can mitigate by using a single sync entry for key; the second device would override the first or vice versa. Not much we can do except maybe notice a discrepancy in history decryption and warn user of conflict.
- **Corrupted Encrypted Key Blob:** If for any reason the encrypted key in sync is corrupted (wrong size, tampered):
- On passkey entry, decryption will fail even with correct passkey. We should detect if the blob is fundamentally broken (e.g., wrong length) and possibly inform the user that the key data is corrupted.
- Only recovery is if they have another device with it or a backup. If not, they may have to reset. This is rare, but the extension could provide a path: "Encrypted key data is corrupted or unreadable. You may need to reset your extension and create a new key." Possibly offer that option.
- **Forgetting Passkey:** Already discussed – provide a data reset option. No way to recover otherwise (by design).
- **Performance Issues:** Encryption and decryption of many entries or a large blob might freeze the extension briefly. We can mitigate by doing heavy work in small chunks or using Web Workers (if available in extension context) to avoid blocking UI. Typically, AES and PBKDF2 on reasonable inputs are fast enough in JavaScript/WebCrypto, especially since WebCrypto is asynchronous and often hardware-accelerated.
- **Interaction with Chrome Sync Passphrase:** If a user has set up an **overall Chrome Sync custom passphrase** (separate from our extension's passkey) – Chrome will encrypt all synced data (bookmarks, etc.) with that. It likely also encrypts extension sync storage data [15]. In that case, our plaintext key in sync might actually be end-to-end encrypted by Chrome as well. That's great for security but we cannot assume it. We should operate under the assumption that sync data *could* be read by Google or others unless we encrypt it ourselves. Our design is independent of Chrome's sync passphrase.
- **Updates to Extension:** If in future we change encryption schemes or want to increase PBKDF2 iterations, we'd need a migration plan. Possibly store a version number in sync with the key and handle accordingly.

Conclusion

In this design, we implement a secure encryption mechanism for session history that balances **security, privacy, and usability**: - **Security:** All session data is encrypted locally with a strong key, and an optional user passkey can lock that key to prevent unauthorized access [13]. We eliminate the dangerous scenario of storing data and its key side by side in plaintext [2] once the user opts into passkey protection. - **Privacy (Local-First):** No actual history content is ever uploaded or shared. The only thing leaving the device is the encryption key (to facilitate recovery) and even that can be encrypted by a passkey. This means even if cloud storage is breached, user data remains safe and indecipherable [4]. - **Usability:** By default, the user experience is seamless – encryption works behind the scenes without setup. For those who want more security, enabling it is a guided process, and using the passkey becomes a simple routine (similar to unlocking a password manager). We've added safeguards and options (like key backup toggle, reset function) to handle edge cases and user mistakes as gracefully as possible. - **Fault Tolerance:** The system handles various scenarios (first install, device sync, lost

credentials) in a predictable manner, prioritizing user control over their data (either secure it or consciously reset it). - **Next Steps:** Implementation should proceed with careful testing of the workflows. Security audits (internal) should verify that no plaintext secrets are inadvertently exposed and that encryption is correctly implemented (e.g., using WebCrypto right, not using weak parameters). UX testing should ensure average users understand what the passkey is and are not confused by the lock/unlock states.

By following this design, our team will deliver a robust feature that significantly enhances user privacy and security for session history, aligning with best practices for client-side encryption and key management in modern web extensions [16] [1] .

---

[1] [8] [11] chrome.storage | Reference | Chrome for Developers
https://developer.chrome.com/docs/extensions/mv2/reference/storage

[2] Bitwarden CLI - Get passwords, username, TOTP and more from Bitwarden - Page 3 - Share your Workflows - Alfred App Community Forum
https://www.alfredforum.com/topic/11705-bitwarden-cli-get-passwords-username-totp-and-more-from-bitwarden/page/3/

[3] [4] [13] [14] Keeper Encryption and Security Model Details | Enterprise Guide | Keeper Documentation
https://docs.keeper.io/en/enterprise-guide/keeper-encryption-model

[5] [9] [16] javascript - Encrypting data to be stored in chrome storage - Stack Overflow
https://stackoverflow.com/questions/27826998/encrypting-data-to-be-stored-in-chrome-storage

[6] [7] [15] Chrome Sync privacy is still very bad | Almost Secure
https://palant.info/2023/08/29/chrome-sync-privacy-is-still-very-bad/

[10] SubtleCrypto: deriveKey() method - Web APIs | MDN
https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/deriveKey

[12] Encrypt passwords in memory of an unlocked browser extension
https://community.bitwarden.com/t/encrypt-passwords-in-memory-of-an-unlocked-browser-extension/58376