Findings and recommendations

1. Findings relevant to risk statement

#	Finding	Severity	Recommendation
1-1	Insecure default parameters lead to storing of unprotected cryptographic material: it is possible to configure Userbase client to store key material in unprotected area.	**High**	• Educate developers about risks of storing seed in local storage without encryption in documentation. • Educate developers about risks as separate warning in SDK. • Store seed encrypted with `KDF(userpassword)`, cache KDF for UX concerns. • Do not store decrypted seed in memory longer than needed.

Userbase Response

By default, the signUp and signIn functions exposed in the Userbase SDK store key material in plaintext in browser session storage to allow the user to stay signed in to a web app after they hit refresh, *without requiring additional action* such as re-inputting a password. This is the default expected behavior when using a web app, and as such, is Userbase's default behavior as well. Clients can be configured not to do this.

Both signUp and signIn functions accept a parameter called **rememberMe**, which can be set to 'session', 'local', or 'none'. In the documentation, we state:

> "When **rememberMe** is set to 'session' or 'local', the user's encryption key and session token will be stored in clear in the browser's session or local storage respectively. Someone with access to these values will be able to access a user's account and all their data until the user explicitly signs out, or the user's session expires. If you want to avoid this, you will need to set **rememberMe** to 'none'. When **rememberMe** is 'none', the user will always have to login with the username and password when visiting your web app."

We also state that the default setting is 'session'.

We believe the default setting set to 'session' is especially useful for our target web app developer who is primarily concerned with mishandling data server-side.

For developers who do not want key material stored in plaintext on user devices and are okay with requiring users re-input their password on refresh, the **rememberMe** option set to 'none' tells the Userbase SDK not to store any key material on the user's device.

1-2	These insecure parameters for key agreement open a number of attacks: • 2048-bit MODP group • Primitive (DHRSA) • Scheme: non-ephemeral	**High**	• Improve Diffie-Hellman scheme (either by replacing with EC-based or by providing additional checks): • Use at least 3072-bit MODP Group. • Implement SCA checks from RFC 2631 #2.1.5 • Switch to ECDH(E) + ECDSA.

Userbase Response

We replaced Diffie-Hellman with ephemeral ECDH + ECDSA in userbase-js version 2 when we released a feature allowing users to share Userbase databases. Any new users created with userbase-js version 2 and up do not use Diffie-Hellman. When users created with userbase-js version 1 sign in from a client using userbase-js version 2 for the first time, the client first proves access to the Diffie-Hellman private key, and then uses the updated ephemeral ECDH + ECDSA scheme from then on. We plan to deprecate all usage of Diffie-Hellman in the future.

The attack vector prior to implementing this is not as dastardly as it may appear, nor does it violate our risk statement. Diffie-Hellman plays a minor role in userbase-js version 1. This attack vector was more significant in the code reviewed for the security review because we used Diffie-Hellman to exchange private keys in that code. However, no released version of userbase-js uses Diffie-Hellman to exchange private keys.

When a user signs up in userbase-js version 1, the user's client uses HKDF to derive a Diffie-Hellman key pair from a randomly generated seed; the server stores the public Diffie-Hellman key. This key is only used in a single step of the authentication process. When the user attempts to sign in, the server computes a shared secret with the user using the user's public key that was stored at sign up, then the server encrypts a randomly generated message that is unique to the single session. The user's client must decrypt this randomly generated message in order to prove to the server the user has access to their Diffie-Hellman private key.

Note that the shared secret the user's client and the server compute each session stays constant. Thus, if an attacker were to get access to this shared secret (or even if the attacker knew the user's Diffie-Hellman private key), **and also had access to the user's password token or session token**, the attacker could pass the authentication process to the server. Even still, after passing the authentication process, the attacker would not be able to decrypt user data.

Clarifying, in userbase-js version 1, **even if an attacker knows the user's Diffie-Hellman private key and has complete access to the server serving the Userbase API and Userbase database, the attacker would not be able to decrypt user data**.

1-3	Potential nonce reuse/forbidden attack due to AES-GCM in transaction bundling process: server can trigger bundling the same data infinite times, thus increasing the chance for nonce reuse.	**Medium**	• Ensure that server can't force client to bundle the same data more than several times. • Consider switching to envelope encryption via **RFC3394** (AES-KW, available in WebCrypto subtle API) or AEAD-based construction like **NIST 800-38F** (AES-GCM based).

Userbase Response

Starting in userbase-js version 2.1.1, clients only attempt to bundle the same set of data once, even if the server requests the client re-bundle data. This puts a much lower practical limit on the number of times the server can realistically get the client to bundle transactions. Keep in mind we use a 96-bit randomly generated IV as recommended by NIST, which means if truly random, a client must generate 40 trillion IV's and send all of them to the server before there's a 1% chance it will reuse an IV (the birthday problem).

We use envelope encryption for the newly released file storage feature in userbase-js version 2.1.1. Every 512 KB file chunk is encrypted with a randomly generated encryption key, which itself is encrypted with the user's encryption key. We plan to switch to envelope encryption in the general case in the future.

1-4	Trust depends on user's ability to compare cryptographic keys and hashes visually – which, under current design, is easy to manipulate. Failure to detect key/hash forgery leads to a number of impersonation attacks, data manipulation and data leakage by both dishonest server and external attackers.	**Medium**	• Provide human-friendly key representation for key comparison by securely translating long key into emoji or a N-digit hash. • Ensure reasonable limits of verification events per second to prevent server / MiTM exhausting verifier attention.

Userbase Response

Before releasing the feature that allows users to share databases with each other, we modified our approach entirely to avoid the issue highlighted here. The verification process we settled on does not rely on users to visually compare long hashes, as it does in the code reviewed in the security review. Users must actually provide each other with their public keys in order to verify each other.

In the code reviewed, when User A wanted to share a database with User B, we expected User A to manually ensure User B's fingerprint is accurate by communicating with User B either in person or through a secure communication channel, asking for User B's fingerprint, and then visually comparing it to the fingerprint displayed to User A inside the app using userbase-js. If the fingerprints matched, User A would know they are sharing the database with User B.

In the code we actually ended up releasing, User B must send their fingerprint to User A through a secure communication channel, and then User A must provide that exact fingerprint to their client in order to verify User B. Only then can User A share a database with User B.

The requirement for User A to verify User B before sharing a database with User B can be turned off, but it is turned on by default. It is the **requireVerified** Boolean parameter passed to the shareDatabase function exposed in the Userbase SDK.

1-5	In a number of cases, cryptographic material stays longer in memory than reasonable and their removal from memory depends on block scoping for `const` declarations.	**Medium**	• Make keys variables, fill private key variables with zeros (perform "zeroization") after usage. • Due to nature of the way JavaScript's GC works, that is still not a guarantee of key removal, but an incremental improvement.

Userbase Response

We plan to address this in the future.

1-6	A number of primitives have custom JavaScript implementations that are better replaced with well-audited primitives that perform computation outside webpage context.	**Medium**	• Consider limiting list of primitives used to those available in WebCrypto API

Userbase Response

We currently use 2 crypto packages: diffie-hellman and scrypt-js.

Diffie-Hellman - We plan to deprecate usage of Diffie-Hellman entirely and it is no longer used for new users in userbase-js version 2.

Scrypt - The only alternative to Scrypt in the Web Crypto API is PBKDF2. We chose Scrypt because it is memory-hard and had a lightweight browser Javascript implementation. We plan to stick with Scrypt.

1-7	Unnecessary long lifecycle for `DHPrivateKey`.	**Medium**	• Enable periodic key rotation for DH keys via updating keypair, uploading `DHSalt` + `DHPublicKey`

Userbase Response

We plan to address this in the future. Note that in userbase-js version 2 there is now a private ECDSA key with a long lifecycle instead of a Diffie-Hellman key.

1-8	Lack of id comparison during database sharing simplifies hijacking or tampering the share.	**Medium**	• Include signed {username, `publicKey`} in response to `GetPublicKey`.

Userbase Response

The risk highlighted here is that a user can be fooled into sharing a database with a malicious man-in-the-middle who has access to the server. We assume that if someone malicious has access to the server, they can tamper with this result and fool clients with invalid signed responses anyway.

We expect developers and users who want protection from man-in-the-middle attacks during the database sharing process to rely on the verification process explained in our documentation, and touched on in our response to finding 1-4. The verification message generated through the verification process ties a username to a public key.

1-9	No way to revoke database access – once granted, always available.	**Medium**	• Design “revoke database access” function via removal of shared `encryptedDbKey`.

Userbase Response

The ability to revoke access to a database was implemented in the official release of database sharing in userbase-js version 2.

1-10	Long-lived security credentials stored on server (even encrypted) lowers trust to credential	**Medium**	• Have seed rotation policy and ability to migrate seed if suspected to be compromised by dishonest server or adversary. • Provide a way for user to report that their credentials were compromised.

Userbase Response

We plan to address this in the future.

1-11	Absence of warnings on potentially destructive actions.	**Low**	• Educate end-users / developers about consequences of sharing the database (data destruction, dbkey leakage).

Userbase Response

We explain in our documentation how sharing a database can result in a man-in-the-middle attack, and how our verification process can prevent it.

We also explain how deleting a user deletes all the databases they own, including ones that have been shared with other users.

1-12	Scrypt has secure parameters that are “borderline insecure”.	**Low**	• Do not lower Scrypt parameters below $N=215$ (32768), $r=8$, $p=1$.

Userbase Response

The Userbase SDK hashes passwords with Scrypt using the following parameters:

N = 16384 (or 16mb)
r = 8
p = 1
Hash length = 32 bytes
Salt = 16 bytes

From the Scrypt paper: "100ms is a reasonable upper bound on the delay which should be cryptographically imposed on interactive logins" [1]

With an optimized Scrypt algorithm running on a 3.1 GHz Intel Core i5, N = 32768 is the highest work factor that takes <100ms for the algorithm to run. Thus, it's the latest recommended work factor. [2]

However, we are not running an optimized version of the algorithm on a single machine. Users are running a pure js version written for the browser. Safari, for example, takes >6 seconds to run when N = 32768 on a 2.5 GHz Intel Core i5. A higher end CPU can only shave around 1 second off that time. Further, it takes over 1s to run in Firefox, and over 500ms to run in Chrome. This is an exceptionally slow interactive login delay to impose on users.

Thus, we went with N = 16384 to ensure interactive logins are closer to the reasonable delay the function will impose on users, while still maintaining a reasonable level of safety.

[1] - https://www.tarsnap.com/scrypt/scrypt.pdf (pg. 13)
[2] - https://blog.filippo.io/the-scrypt-parameters/

1-13	No heightened protection after password recovery.	**Low**	• Bind both key rotation and seed backup simultaneously after password reset

Userbase Response

We plan to address this in the future.

2. Findings relevant to general security

#	Finding	Severity	Recommendation
2-1	Lack of XSS, CSRF, CORS prevention measures.	**High**	• Harden web application against typical attacks

Userbase Response

We leave XSS and CSRF protection to the web app developers using userbase-js.

We plan to implement an origin whitelist to protect against CORS-related attacks.

2-2	Weak ciphersuites allowed in SDK <> Server communication, lack of HSTS.	**High**	• Harden transport/TLS: HSTS, ciphersuite, • Enforce correct TLS usage within hosted solution, recommend in documentation for on-prem version.

Userbase Response

The Userbase server sets the Strict-Transport-Security header on all responses and redirects all http requests to https.

Of note, userbase-js comes shipped with the default server endpoint hardcoded to https://v1.userbase.com. Therefore by default, all clients should use TLS when communicating with the Userbase server.

2-3	Admin endpoint is exposed from the same code and process as main user-facing API.	**Medium**	• Separate `/admin` endpoint in code and in access logic.

Userbase Response

We plan to address this in the future.

2-4	Removing `encryptredDbKey` from database will render Db inaccessible.	**Medium**	• Implement key backup via store `encryptedDbKey` on client side as well.

Userbase Response

We plan to make it easier for both admins and users to back up their data regularly in the future.

2-5	Absence of user removal flow leads to inactive users with valid credentials that can be misused.	**Medium**	• Implement key cleanup for users that are removed from the system – seed, seed backup, client's `pubkey`, `encryptedDbKey`.

Userbase Response

We implemented a purge process that purges deleted users' data from the server.

2-6	Server's public key, which is used in any operations, is not signed / verifiable.	**Medium**	• Enable client to validate server's public key from a third party.

Userbase Response

The client does not rely on the particular public key referenced here to be correct. It is only used so that users can prove themselves to the Userbase server. If a client uses an incorrect public key, the client's user would not be able to prove they are in fact the correct user to the Userbase server, and would not be able to sign in. Further, users created in userbase-js version 2 do not use this public key. We recommend all developers use the latest version.

2-7	Use package verification for Userbase SDK	**Medium**	• Implement integrity checks for library via NPM TBV. • Encourage developers to verify library signatures against public ones.

Userbase Response

We plan to address this in the future.

We also plan to make all versions of userbase-js available at direct links, making it simple for developers to access the userbase-js script using subresource integrity.

2-8	Duplicate implementations for the same cryptographic primitives and security controls.	**Low**	• Remove duplicate implementations, each security control should have one implementation used across codebase where needed.

Userbase Response

We plan to address this in the future.

2-9	Not all cached sensitive data elements are removed.	**Low**	• Remove cached potentially sensitive data (keys).

Userbase Response

We plan to address this in the future.

2-10	Lack of session re-validation before security-sensitive events.	**Low**	• Re-validate sessions before security-critical events

Userbase Response

Users must re-input their password to change their password in all officially released versions of userbase-js.

2-11	Temporary passwords and user sessions can stay in database longer than necessary in case of unfinished password reset process.	**Low**	• Expire temporary passwords and user sessions

Userbase Response

Both are purged from the database after 24 hours.

2-12	Insufficient logging for some security events (sharing, revoking, sharing seed, password reset, restoring seed backup)	**Low**	• Log access events in more depth on server

Userbase Response

We plan to address this in the future.