Userbase security review report

Overview and Executive summary

In December 2019, Userbase requested Cossack Labs offer an opinion whether Userbase's software architecture, source code and example application allow it to achieve the chosen design goal:

> *Userbase prevents an adversary with privileged access to the server serving the Userbase API* from seeing protected user data*, if and only if the web app owner uses Userbase according to its design and behaves honestly.*

* see definitions on page 2.

We have found that if Userbase is used correctly by both the developer and end-user, it is possible to build Userbase-based applications that fulfil the security claim above to a reasonable degree, with a few suggested improvements.

However, from a general application security standpoint, we have plenty of space for significant improvement in both the design and implementation related to:

- cryptographic design,
- cryptographic implementations,
- key management,
- application security

As a result, a list of security concerns has been identified and communicated to Userbase's team, along with recommendations:

Weakness type	High	Medium	Low
Design goal weaknesses	2	8	3
Application security weaknesses	2	5	5

Throughout the review, Userbase team has displayed strong security-driven approach, fixing many concerns immediately and planning to fix many of them on the go.

This review was performed during December 2019–January 2020 and consisted of around 100 man-hours of work, split between design review, implementation review, validation of risk statement and verification of chosen security controls.

Goals and approach

Review goal: verify whether a Userbase-based application can satisfy the risk statement / application design goal:

Risk statement/design goal: Userbase prevents an adversary with privileged access to the server serving the Userbase API from seeing protected user data, if and only if the web app owner uses Userbase according to its design and behaves honestly.

Scope for risk statement: Userbase prevents confidentiality risks regarding sensitive data that has been put into the Userbase database (encrypted locally for storage in the Userbase backend).

Non-scope for risk statement:

- It is understood that anyone with access to the Userbase server serving the API (an attacker, malicious insider / service owner) will be able to reveal the things that are outside the protected data scope (for example, usernames, data access patterns, number of items stored, and the size of items stored), so it is out of the scope of the current assessment.
- It is understood that anyone with access to the server serving the Userbase client or client-side code will be able to reveal the things inside the protected data scope.
- 3rd party implementations of cryptographic primitives are considered trusted, or can be replaced by well-studied ones.

Definitions:

Protected user data: sensitive data stored inside the Userbase database encrypted locally on the client for storage in the Userbase backend.

Userbase client SDK userbase-js: the javascript written by the Userbase team that web app developers use in their static html/js/css that they serve to their users, implementing client-side security and cryptographic functions.

Web application made by Userbase users (3rd party developers). A web app contains:

- Frontend code (html/js/css). Javascript part of frontend code calls userbase-js SDK.
- Userbase client SDK (userbase-js) SDK created by the Userbase team that developers use to protect customers data.

Userbase server serving the API: A server hosting the userbase-server component, which has two responsibilities:

1. Respond to userbase-js API requests.

2. Perform data store/retrieval operations inside a database that Userbase controls.

   During this assessment it was assumed that the userbase-server is hosted by the Userbase team here.

Service serving Userbase SDK to client: service that provides the userbase-js SDK for web apps.

The scope of the risk statement assumes that the correct Userbase client SDK loads into the web app safely, that the correct static html/js/css is loaded into end-users' browsers, and that the intended client-side binary always runs.

Coverage:

This security review has been based on:

- Code: Security-review branch, commit 1af6e5 https://github.com/encrypted-dev/userbase/commit/1af6e5caf58fd6cabad45fa8f6618c89d943c379
- Design documentation provided by Userbase.

Methodology

Cossack Labs' review has constituted a number of activities:

- **Risk assessment**: defining a quantitative model for risks associated with the risk statement in a chosen layout.
- **Design/architecture review**: proactively seeking design flaws that lead to leakage of sensitive data stored within a Userbase app.
- **Fitness assessment of security controls to the chosen risk model and risk statement**: verifying whether the security controls chosen are optimal to mitigate the chosen risks and general risks towards sensitive data.
- **Cryptographic design and implementation review**: verifying whether the chosen combination of cryptographic primitives and their implementation actually embodies desired security properties.
- **Application security** (partial): ensuring that application-level security controls are implemented well.

Triaging issues: due to the specific risk statement / review goal, issues discovered could not be triaged using a common methodology like CWE or CVSS – the outcomes, loss magnitude in the general context are significantly different compared to a regular security assessment and some of the vulnerability severity scores would be misleading.

Thus, we've conducted a rough quantitative risk assessment, formulated a simple trust and risk model that reflects the risk of breach within the chosen risk statement, and used it to triage vulnerabilities relevant to risks Userbase is addressing.

Findings summary

Fitness to risk statement and goals

Our general conclusion is that the set of security controls implemented by the Userbase team can achieve the goals of the risk statement (preventing honest web app owners from seeing the data that clients may put into Userbase) to a satisfactory degree. However, a few potential caveats are present in non-critical components that require addressing to eliminate potential misuse by web app owners.

Limitations

In our opinion, there are a number of critical risks which, while being out of scope of this audit, are still viable threats to customer's sensitive data based on general risk analysis, and should be attempted to be addressed. It is understood that the client-side application is prone to typical attacks on browser-based applications, - from tampering cryptographic code (or any other application logic code in runtime due to JS/DOM mutability) to leaking sensitive data and key material.

Additionally, the chosen application architecture, security design and trust model enable various non-cryptographic attacks on plaintext, cryptographic code or keys, which can render cryptographic protection useless against an adversary (external or insider, e.g. dishonest server operator) with access to the service serving the Userbase SDK to clients.

These, however, are not vectors for analysis in current design review, which is limited to risk statement. Advisory provided in application security context and general cryptographic design context is limited to basic adherence to industry practices.

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.
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.
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).
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.
1-5	In a number of cases, cryptographic material stays longer in memory than reasonable and their	**Medium**	• Make keys variables, fill private key variables with zeros (perform "zeroization") after usage.

	removal from memory depends on block scoping for `const` declarations.		• Due to nature of the way JavaScript's GC works, that is still not a guarantee of key removal, but an incremental improvement.
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
1-7	Unnecessary long lifecycle for `DHPrivateKey`.	**Medium**	• Enable periodic key rotation for DH keys via updating keypair, uploading `DHSalt` + `DHPublicKey`
1-8	Lack of id comparison during database sharing simplifies hijacking or tampering the share.	**Medium**	• Include signed {username, `publicKey`} in response to `GetPublicKey`.
1-9	No way to revoke database access – once granted, always available.	**Medium**	• Design "revoke database access" function via removal of shared `encryptedDbKey`.
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.
1-11	Absence of warnings on potentially destructive actions.	**Low**	• Educate endusers / developers about consequences of sharing the database (data destruction, dbkey leakage)..
1-12	Scrypt has secure parameters that are "borderline insecure".	**Low**	• Do not lower Scrypt parameters below $N = 2^{15}$ (32768), $r = 8$, $p = 1$.
1-13	No heightened protection after password recovery.	**Low**	• Bind both key rotation and seed backup simultaneously after password reset

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
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.
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.
2-4	Removing `encryptredDbKey` from database will render Db inaccessible.	**Medium**	• Implement key backup via store `encryptedDbKey` on client side as well.
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`.
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.
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.
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.
2-9	Not all cached sensitive data elements are removed.	**Low**	• Remove cached potentially sensitive data (keys).
2-10	Lack of session re-validation before security-sensitive events.	**Low**	• Re-validate sessions before security-critical events
2-11	Temporary passwords and user sessions can stay in	**Low**	• Expire temporary passwords and user sessions

	database longer than necessary in case of unfinished password reset process.		
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

Conclusion

Browser-based security products have long-known architectural caveats, and end-to-end encrypted applications typically receive reasonable criticism. In this context, understanding a realistic risk model and setting attainable goals is essential to make browser-based applications protected beyond the traditional “do not trust the browser, rely on TLS and adequate AppSec measures” approach.

In our belief, the authors of Userbase are moving in the right direction and have put significant effort to build a secure solution.

About Cossack Labs

Cossack Labs is a provider of data security tools (cryptographic and data security frameworks), bespoke solutions and consulting services, with a focus on sensitive data protection in modern systems. Cossack Labs’ experts participating in this audit, have decades of hands-on practical experience, appropriate formal education and academic degrees in the area of cryptography, data security and general information security.

Cossack Labs can be contacted at: cossacklabs.com / info@cossacklabs.com