Protocol Security Review

Keybase

February 27, 2019 – Version 1.3

Prepared for
Max Krohn

Prepared by
Keegan Ryan
Thomas Pornin
Shawn Fitzgerald

Document Change Log

Version	Date	Change
1.0	2018-10-14	Initial report
1.1	2018-10-18	Updated for initial technical review
1.2	2018-12-13	Updated for additional technical review
1.2	2019-02-11	Updated for final technical and marketing review
1.3	2019-02-23	Public Report Ready for Keybase

Executive Summary

In September 2018, NCC Group performed a review of the Keybase protocol designs and implementations. Keybase offers several features that are designed to provide cryptographic guarantees of security while maintaining a high level of usability. These features include key management, identity proofs, secure chat, ephemeral messaging, encrypted file storage, and key exchange, among others. Keybase supports clients for Android, iOS, desktop (Windows, macOS, and Linux), and web, although the core library is cross-platform and written in Go.

Scope and Limitations

This review aimed to identify any shortcomings in these components that fail to deliver on the security promises of the product. To maximize the benefit of this engagement, NCC Group focused on analyzing the designs to either confirm the claims of security or identify weaknesses. This approach both justifies positive design decisions and strengthens weaker ones. A secondary focus of the review was analyzing the shared core library for exploitable flaws, as the scope of the vulnerabilities would affect multiple platforms and have a greater impact. Most of NCC Group's effort focused on reviewing sigchain verification, local key storage, the Keybase filesystem, key exchange and encrypted chat. The substantial complexity of the Keybase codebase also meant NCC Group could not perform an exhaustive review of all components in the allotted time.

NCC Group devoted nine person-weeks during this security assessment taking place between September 10 and September 28, 2018 and focused the review on a specific Github commit.[1] At Keybase's request, NCC Group retested certain findings that were fixed during the course of the review period and also during the week of February 25th 2019; the results of the retest are reflected in this report and the findings marked as fixed where appropriate.

Key Findings

Keybase aims to offer services to its users that are both easy to use and cryptographically secure, reducing the reliance on a central authority. Keybase's architecture protects user data, limits the need to trust servers, and links identities via cryptographic signatures. In its investigation, NCC Group found that while Keybase's design protects against many types of attacks, there were weaknesses in the Keybase implementation; these were quickly fixed.

One of Keybase's design features is hosting public information in a way that allows third parties to verify that the data has not been tampered with. For this to succeed, independent implementations are needed, which in turn require thorough documentation on how to consume and verify the data. While newer features are documented, some features, especially legacy ones, are lacking in documented implementation details. In addition, some of the documents do not perfectly represent the current state of Keybase's code. This increases the difficulty of auditing the code, as it may not be clear that some features even exist. This increases the difficulty of verifying both the data and the client code. Some documentation shortcomings are recorded in Appendix B on page 36.

Another common theme was the presence of legacy code. In order for clients to verify the public record, they must understand data generated by the earliest iterations of the protocols, which have evolved significantly since their creation. Verifiability requires backwards compatibility, and so the client increases in complexity over time, making it difficult to remove legacy code.

This does not necessarily imply that legacy code is insecure, but complexity and security are intertwined–every new piece of code may contain a security vulnerability, and more code correlates with more risk. New code also means more ways different components may interact: legacy payloads and legacy signatures may be secure, but security may not extend to modern payloads with a legacy signature. The need to support legacy data contributed in part to both finding NCC-KB2018-001 on page 22 and finding NCC-KB2018-004 on page 28.

Strategic Recommendations

NCC Group recommends a long-term approach of addressing the themes identified above. This report suggests design principles that would further strengthen the overall security posture of the Keybase ecosystem. Improving documentation requires writing up not only the concepts, but also the fine details of the Keybase protocol, so that independent but compatible implementations may be written. Such documentation would greatly improve the auditability of the system and increase user trust.

NCC Group also recommends taking steps to address the current complexity of the protocols. While an

[1] https://github.com/keybase/client/commit/049d28bddb3c71cdd986f52d0f9fd9344d17a713

effort to phase out legacy data would be considerable and may pose security risks, clients could convert old data into newer formats in the background, and then eventually only support the new formats. Moving forward, the security risks of a complex code base should be weighed when introducing new features. Keybase has considered the risk and reward of phasing out legacy mechanisms multiple times but has decided that the benefits are not worth the cost at this time.

Simplicity should be adopted as a design goal, and features should be introduced with some notion of future-proofing: changes should be possible without having to redesign security-critical components. Addressing these two points would go a long way in improving Keybase's overall security posture while achieving the goal of user-friendly cryptographic guarantees.

Dashboard

Target Metadata

Name	Keybase Protocol
Type	Multiplatform Client
Platforms	Golang
Environment	v2.7.0

Engagement Data

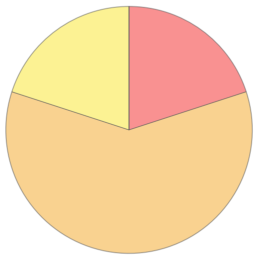
Type	Design and Implementation Review
Method	Code-assisted, Design Review
Dates	September 10, 2018 to September 28, 2018
Consultants	3
Level of effort	9 person-weeks

Targets

Documentation	https://keybase.io/docs
Client Code	https://github.com/keybase/client/commit/049d28bddb3c71cdd986f52d0f9fd9344d17a713
Encoding and Decoding Library	https://github.com/keybase/go-codec/tree/1cf5262595f30a38e4c594385d275b83653a6a89

Finding Breakdown

Critical Risk issues	0
High Risk issues	1
Medium Risk issues	3
Low Risk issues	1
Informational issues	0
Total issues	**5**



Category Breakdown

Cryptography	2	🟧 🟨
Data Validation	2	🟥 🟧
Denial of Service	1	🟧

Key

Critical 🟥 High 🟥 Medium 🟧 Low 🟨 Informational ⬜

Sigchain Verification

NCC Group reviewed the verification process of a user's *sigchain*,[2] which is a per-user list of statements regarding the user's identity. As a Keybase user performs certain actions, such as provisioning a new device, following another user, or connecting an identity to a third-party website, a new entry is appended to the sigchain. The sigchain for every user is public and designed to be immutable, since the ability to retroactively modify a user's sigchain could undermine the confidence in earlier statements about their identity. Keybase periodically adds a hash derived from all users' sigchains to the Bitcoin blockchain[3] to allow users to verify the sigchain offered up by the server.

There are two primary adversaries the user sigchain verification process must protect against. The first is a malicious server that is capable of returning arbitrary data in response to sigchain API queries. As sigchain contents are obtained by the client from the server, this means the malicious server could return unmodified data, a slightly changed version, a totally manufactured version, or no information at all. In the cases where the server returns inauthentic statements about a user, the client should be able to detect this, and verification should fail. Another adversary is a malicious client in possession of their own private keys. This adversary can create signatures over arbitrary payloads and upload arbitrary statements to the server in an attempt to attack either how the server or the legitimate clients handle malformed sigchain data. Well-behaved servers and clients must therefore perform full validation of sigchain data, even if it is properly signed. It is also possible that a malicious client and malicious server work together to target a legitimate user.

The sigchain design must protect against these adversaries while simultaneously offering high-assurance statements about each user's identity. It should allow users to append to their own sigchains for common identity management operations, prevent unauthorized modifications to other users' data, and minimize the amount of trust placed in the operator of the server.

Sigchain Design

At a high level, a user's sigchain is composed of a series of statements known as *chain links*. These chain links contain signed information detailing changes to the user's sigchain, such as the addition or revocation of a new device, an identity proof, or a follower statement. As part of the sigchain verification process, the signature within each chain link needs to be verified to ensure that the chain link actually originated from the user in question. This is done by authenticating the "payload_json" field against the "sig" field using the client's understanding of the signing keys active to the user's account at the time the chain link was added to the sigchain. By using a strong signature algorithm and signing a value derived from the entirety of the "payload_json" field, it is infeasible for an attacker who does not possess the signing key to modify the "payload_json" field and still provide a valid signature over the data. In other words, an attacker would be unable to modify the "payload_json" contents. This gives client performing verification confidence that those contents within the chain link are legitimate and trustworthy.

The client is also responsible for verifying the order of chain links within a sigchain. Maintaining order is important, otherwise an adversary could perform attacks such as delaying the revocation of a key or dropping certain identity assertions. Every chain link in a sigchain has a unique sequence number. These sequence numbers begin at one for every user, and increment by one for every additional chain link added to the sigchain. To cryptographically prevent modification of a sigchain, every newly added chain link includes a hash of the previous chain link's payload, which includes the previous sequence number as well. This hash is included in the signed payload of the current link. Since the SHA-256 hash used in this process is secure against preimage and collision attacks, an attacker cannot change the payload or sequence number of a previous chain link without changing the hash and invalidating the current signature. Therefore, if the client trusts the payload hash and signature of the most recent chain link in a sigchain, they similarly trust all prior chain links in the sigchain.

After creation of a new user or resetting an account, a new subchain is created within the sigchain. This subchain simply represents a particular starting point within the sigchain where any prior statements are ignored and a new "eldest" signing key for the sigchain is declared. This key can be used to provision additional keys or sign future chain links.

[2] https://keybase.io/docs/sigchain
[3] https://keybase.io/docs/server_security/merkle_root_in_bitcoin_blockchain

To give the client greater confidence in the payload hash and signature of the most recent chain link, these data are encoded in the Bitcoin blockchain, which provides an immutable and public record of the data. The cost of any attacks on these properties of the Bitcoin blockchain is going to be great, and an adversary capable of performing such attacks will likely derive greater benefit from other attacks, such as double-spends. For this reason, Keybase users can be confident that all other Keybase users are seeing the same Keybase data in the Bitcoin blockchain.

All users' most recent chain links are incorporated into a single value via the use of a data structure called a Merkle tree.[4] In a Merkle tree, all data (represented as leaf nodes in a tree) are combined in a series of cryptographic hashes to produce a single output value. The cryptographic properties of the hash prevent an attacker from modifying any of the leaf data without changing the output value. Keybase periodically writes this output value to the Bitcoin blockchain, allowing users who wish to verify a chain link to efficiently conclude that the specific chain link was in fact one of the inputs in the Merkle tree. By the immutable and public nature of the Bitcoin blockchain, the user is confident that all other users see the same chain link as the most recent for any given Keybase user.

Combining all of these steps, this design enables a Keybase client to verify that a sigchain is consistent, all chain links were signed in the order presented starting from the declared eldest keys, and that all Keybase users have a consistent view of the sigchain at that point in time. Before discussing how the Keybase implementation compares to the design, it is important to consider which attacks are and are not practical against the design, and the implications that these have.

Potential Attacks on the Sigchain Design

As described above, the design leaves the possibility for certain types of attack and excludes the possibility of others. A malicious server cannot modify the contents of any payloads without knowing the signing keys, but the server can modify other fields. Clients should not depend on any of these modifiable fields for information about a user. A malicious server cannot remove chain links from the beginning of a sigchain without detection, since the first chain link must have sequence number 1. Additionally, the attacker cannot insert, remove, or rearrange any later chain links because the last chain link is verified and each chain link specifies a unique previous chain link in its signed payload. An attacker cannot append to a sigchain without knowledge of any of the unrevoked signing keys either. An attacker is also unable to present inconsistent views of the same sigchain to two separate users due to the use of the public and immutable Bitcoin blockchain.

However, the attacker is capable of refusing to update a sigchain or rolling back a user's sigchain to a previous state by truncating later chain links. The sigchain will verify exactly as it did before up to the point of truncation, and the server can falsely attest that this chain link in the middle of the sigchain is actually the end, publishing this statement to the Bitcoin blockchain. Since all Keybase users see the same truncated sigchain, this does not violate the consistency guarantees of the blockchain. Clients could catch this behavior by continuously verifying that the server advertises a latest chain link that is at the end or past the locally cached sigchain verified by the client. This also presents the possibility of a Keybase user provisioning a key, revoking the key, and disclosing the key, and the server rolling back the user's sigchain to before the revocation and appending to the sigchain using the disclosed key.

As a final attack to discuss here, the Keybase server could at any time reset or replace a sigchain with a similar sequence of chain links where all keys are replaced with keys known to the attacker. If the user has posted any proofs to third-party sites, this can be detected, but if not, detection requires either some sort of out of band verification of keys or a user frequently verifying their own sigchain against the one Keybase has committed to the Bitcoin blockchain.

Both caching by clients and follower statements are an important part of preventing these sorts of attack. By frequently having other Keybase users attest to the view they have of a user's sigchain and comparing the locally cached version with the server-provided value, the act of modifying the sigchain becomes more easily detected. More information is found in Keybase's documentation,[5] which discusses the benefits of having a user attest another's sigchain by following it as early as possible. However, there are benefits to both these so-called older follower statements and newer follower statements. Older follower statements have had more time for the follower to detect potential fraud or compromise;

[4] https://en.wikipedia.org/wiki/Merkle_tree
[5] https://keybase.io/docs/server_security/following

in this respect they give greater confidence that the sigchain is trustworthy *up to the point in time when the follower statement was created*. Newer follower statements are more likely to cover new additions to the sigchain, preventing such attacks as rolling back a revocation. Both older and newer follower statements give greater confidence in the claimed identity of a user, and they are therefore a large contributor to the trustworthiness of the Keybase system.

According to Keybase, the client currently does perform the stringent verification needed to prevent these attacks, although NCC Group did not verify these claims during the time-limited engagement. Resistance to these attacks relies on a multi-part defense-in-depth approach, including client caching and verification and an auditable public history. By caching information locally, clients can recognize if the server makes unexpected changes, like truncating sigchains or sigchain replacement. To ensure that the current state of a sigchain has been incorporated into the immutable record of the Bitcoin blockchain, the server improperly modifies a sigchain and incorporates the new sigchain into the public record. A third party auditing the publicly available Keybase history would not be able to verify the integrity of the sigchain. This is the major benefit of having a publicly auditable record, but there are important caveats to this approach that deserve mention.

First is a question of how users communicate potential discrepancies in server state; if one client fails to verify information from the server, how does that user communicate their concerns to the rest of the Keybase user base? What channels exist to amplify the detection of a misbehaving server? The other caveat is the requirement of a third party auditing the public record. The potential for external verification does not provide as strong of a security benefit if no one is performing external verification. In order to maximize the security benefit of auditability, an unrelated third party or parties should mirror all public data and verify the historical record using an independently developed verification implementation. It is unclear who has this responsibility, and it is also unclear if this is even a question Keybase themselves can address, given the assumption in these hypothetical scenarios of Keybase being compromised. Nevertheless, having answers to these questions is an important part in building public trust in the Keybase system.

Implementation Details

During the course of the engagement, NCC Group compared several critical properties of the design against the Keybase implementation. The large amount of complex code and limited time means that not all security-relevant properties could be checked.

NCC Group reimplemented the code that verifies each chain link contains a cryptographic hash of the previous chain link and found that the several tested sigchains were compatible with the documentation. Source code review of the Keybase implementation determined that the client properly checks hashes and sequence numbers. Furthermore, NCC Group implemented signature verification within a chain link for some of the signature types.

Rather than use one signature type, the Keybase client supports multiple. There are variations in key type, payload value, and payload encoding, and multiple combinations of these parameters can be used for valid signatures. Currently Keybase includes legacy support for PGP keys[6] and support for more modern NaCl keys.[7] Signatures within chain links may be of either format. For NaCl keys, there is also flexibility in how the payload bytes are signed. In some cases, a prefix and null byte may be prepended to the payload bytes, and this concatenated message is signed. In other cases, the payload itself is signed. The final distinction is between version 1 (V1) and version 2 (V2)[8] payloads. In V1 payloads, the payload matches the JSON object in the chain link's "payload_json" field. In V2 payloads, the payload is a msgpack buffer comprised of a version number, a sequence number, the previous hash, a hash of "payload_json," signature type, sequence number type, and a boolean flag. V2 signatures enable the server to skip providing the full "payload_json" value when only the hash is needed by the client.

This variety of supported signature types requires that a client can unambiguously and securely determine the signature format; otherwise a client may interpret a payload in a different way than the signer intended. In the code tested by NCC Group, this was found not to be the case, as is documented in finding NCC-KB2018-004 on page 28. This

[6]https://keybase.io/docs/command_line
[7]https://keybase.io/blog/keybase-new-key-model
[8]https://keybase.io/docs/teams/sigchain_v2

finding exploits the fact that some unsigned data is used to influence how signatures and payloads are interpreted.

A similar vulnerability was found in the verification process of individual chain links. Due to past server bugs, current clients must recognize which chain links contain invalid data. The outcome of the check for invalid links can be arbitrarily influenced by a malicious server, causing victim chain links to be skipped during verification without proper processing. This is detailed in finding NCC-KB2018-001 on page 22.

In conjunction with source code review, NCC Group also performed limited dynamic testing of the sigchain verification process. This was accomplished by using a HTTPS proxy to intercept API requests and dynamically change the server responses. This made it possible to observe how the client behaved in response to different inputs and to quickly develop test cases and proof of concept exploits of the client application.

Assessment Summary

- The sigchain and chain link have many desirable properties that prevent several types of attacks.
- Untrusted data is used in multiple locations to influence interpretation of trusted data in an insecure way.
- This review did not exhaustively cover all security properties of sigchain verification, and further review may uncover additional bugs.

Local Key Security

Keybase clients add additional protections to private keys on the user's devices with Local Key Security[9] (LKS). In environments where native key-protection facilities (such as Keychain on macOS) are not available, LKS uses the user's Keybase password to encrypt the local contents. This way, if the key files are compromised on the device, the private keys are still protected by the password. While there are standard ways to protect data with a password, Keybase has additional requirements which necessitate a more complex LKS design.

Design and Security Model

The LKS must fulfill the following characteristics:

- Physical possession of a user device, and knowledge of the user password, should be sufficient to unlock the stored keys (possibly with help of the server).
- Secured secret keys should not be stored or exposed anywhere else than on the relevant client device. In particular, these keys are never sent to the server in plaintext or encrypted form.
- If a user changes their password on one device, the password change should be propagated to all the user's other devices, even those that are currently offline. In other words, the new password should be usable on all devices to unlock the keys stored on these devices.

Note that the user password is used for authentication against the server as well as local key security. The former operation uses the following process: the user password is stretched into a 32-byte secret value with scrypt, which is then interpreted as a private key on elliptic curve edwards25519 and used to sign a challenge value sent by the server. The server then stores the corresponding public key. In particular, the server can already perform an offline dictionary attack on the user password. Therefore, the password-dependent elements of the LKS may also be exposed to the server without changing the existing security model.

Encryption and Password Security

The LKS operates in the following way:

- Each device generates a random symmetric key k_d. This key is used for encrypting the other device keys for local storage. k_d is *not* used for anything else.
- The user password is stretched (with scrypt) into a bit string π_u of the same length as k_d.
- The value $m_{u,d} = k_d \oplus \pi_u$ ("mask") is stored on the server.
- To decrypt local secrets, $m_{u,d}$ is retrieved from the server, and XORed with π_u to obtain k_d.
- When the user changes their password on one of their devices, that device recomputes the stretched versions of the user's old password (π_u) and their new password (π'_u); the XOR of these two strings $\delta_u = \pi_u \oplus \pi'_u$ is then sent to the server. The server updates all the stored masks $m_{u,d}$ for all the user's devices by XORing them with δ_u.

The server only sends the mask values $m_{u,d}$ to the authenticated user.

This mechanism ensures the required security properties, thanks to the following points:

- Secret keys are ultimately stored on the device, and only on the device. They *cannot* be extracted by attackers other than through compromise of the user device even assuming a malicious server.
- An attacker who compromised the user's device cannot perform an offline dictionary attack on the password: the attacker does not have the mask value $m_{u,d}$, and cannot obtain it from the server without demonstrating knowledge of the user password. The local keys are themselves encrypted with k_d, a truly random high-entropy key, therefore assumed to be immune to brute force attacks.
- On the other hand, if the attacker compromised both the user device and the server, then they can run an offline dictionary attack on the user password. This is unavoidable, given the requirement that physical possession of the device, knowledge of the password, and server cooperation are enough to unlock local keys: given snapshots of the contents of the device and the server, the attacker can leverage server knowledge of the mask value to try candidate passwords on his own computers without any limitation save the total computing power they can muster.
- The server can use δ_u (on password change) to infer the user password, assuming it has low entropy; the method

[9] https://keybase.io/docs/crypto/local-key-security

is described below. However, the server can already do that with the password-derived public key that the server holds to perform user authentication.

User-created passwords tend to have low entropy. If, during a password change, both the old password and the new password are part of a small given set of N possible passwords (e.g. $N = 2^{30}$ for 30-bit entropy passwords), then, given δ_u, the old and new password can be reconstructed with the following process:

1. For all possible passwords p_i in the list ($1 \leq i \leq N$), recompute the corresponding stretched key $\pi_u(p_i)$.
2. Sort the N values $\pi_u(p_i)$ lexicographically; this has cost $O(N \log N)$.
3. For each password p_i, suppose that it is the value of the old password. In that case, the stretched version of the new password p'_i is such that: $\pi_u(p'_i) = \pi_u(p_i) \oplus \delta_u$. Find the value in the list via binary search (this has cost $O(\log N)$); if it is present, then, with overwhelming probability, the hypothesis on the old password p_i is correct, and the looked-up value corresponds to the new password p'_i.

The sorting and binary search can be optimized with bucket sorting and hashtables, but the cost of the N stretching operations will dominate in a practical attack while all N values will fit in the RAM of a common PC.

Assessment Summary

The Local Key Security design provides the security characteristics described above, subject to two important caveats:

- All the security analysis relies on the fact that the server *already* has enough information to perform an offline dictionary attack on the user password; in that sense, the password-update δ_u value is not a security risk because it does not make the situation worse than it already was. However, if the user password for local key security and the user password for authentication with the server differed, then the password update mechanism would represent a change to the existing security model.
- During the analysis of the LKS, NCC Group noticed that the user passwords have weak entropy constraints: only a minimum size of six characters is enforced. This has been detailed in finding NCC-KB2018-003 on page 26.

Keybase Filesystem

NCC Group reviewed the Keybase Filesystem (KBFS) cryptographic design.[10] This part of the protocol offers hierarchical storage of arbitrary data, with several security features:

- Each hierarchy of files has a top-level folder (TLF). Access rights are defined on the TLF and applied on all the hierarchy below the TLF.
- A TLF may be public (in which case data is not encrypted), or not. A non-public TLF has a defined set of *writers* (user devices that can read and write data) and a set of *readers* (devices that can read but not write). A *private* TLF is a special case of a non-public TLF with a single writer and no reader.
- The server stores the data and can, by construction, prevent access or destroy data; however, it must not be able to read non-public data, or perform any alteration that would not be reliably detected by clients.

Keys

Each device maintains its own specific asymmetric keys, one key pair suitable for asymmetric encryption (in NaCl's `box` format, using the Curve25519 elliptic curve) and one key pair suitable for signatures (with the Edwards25519 curve); in older Keybase versions, PGP key pairs could also be used. The public keys are pushed onto the user's signature chain: the signature key becomes a "sibling key", and the encryption key is a subkey of that signature key. The encryption key pair is used to transmit to the device the secret values needed to read and write non-public file contents, and the signature key pair is used to maintain file data and folder integrity.

Keys are identified by their *identifiers*. In recent Keybase versions, the identifier is the public key itself, with an extra header; in older versions, PGP keys were identified by the hash of the public key. An important feature of such key identifiers is that they cannot be faked: No two distinct key pairs may have the same identifier.[11]

Each TLF has its own symmetric secret key. That secret key is encrypted with the public encryption keys of all the users who need access to the TLF (both readers and writers); similarly to what is done with Local Key Security (see Local Key Security on page 10), what is sent to these users is actually a *masked key*, i.e. the XOR of the key with a random mask, the latter being stored on the server. The mask is used to enable the server to "delete" the information proactively if a user's access is to be revoked. Of course, nothing prevents any user to remember any information it has acquired at least once; thus, the deletion of the mask does not guarantee eviction of any former reader or writer. However, the feature can still be useful as a first-level, immediate mitigation, prior to a more thorough rekeying process in which a new TLF key is selected and shared only with remaining users.

Access Rights

All readers and writers of the TLF are duly identified as such, both to each other and to the server. Who can read and/or write to a TLF is not public information, for privacy reasons[12]; only allowed readers and writers should be able to obtain such knowledge. The server is not a reader or writer, but it can necessarily infer who performs every read or write operation, by simply observing the traffic. Moreover, the server must prevent alteration or deletion of data by readers who do not have write access; it cannot perform that service without clear and precise knowledge of access rights.[13] For these reasons, access rights information is made available to the server.

The creator of a TLF is the "owner" and initial writer. New readers and writers are created through signed messages; each writer can create new writers and new readers, and each reader can create new readers. By obtaining these signed messages and verifying them, both readers and writers can know the public keys and access rights of all other members of the TLF access group.[14] Since modifications are indirectly signed by the user who performs them, all readers can cryptographically decide whether a write access is valid or not. This characteristic "keeps the server honest": while the server maintains integrity of the data storage, it cannot itself corrupt the data without being detected.

[10]https://keybase.io/docs/crypto/kbfs
[11]In the case of PGP keys, this is due to the hash function being resistant to collisions.
[12]In a public TLF, everybody is a reader, but the exact set of allowed writers is still a private information.
[13]It would be conceptually feasible to have the server simply accumulate write requests without destroying any data element, and let the clients sort out the valid and invalid writes. This would, however, impact performance, require more storage space, and prevent application of storage quota by the server.
[14]A consequence is that newly added readers and writers can learn about the past members of the reader and writer sets.

Integrity Checks

Every file is stored as one or several blocks; for non-public TLFs, blocks are encrypted. Each block is identified by its hash (with SHA-256). Folders are also represented as blocks that contain the hashes of the blocks of the files and subfolders contained in that folder. The process is recursive up to the block for the top-level folder. Each write operation implies a new TLF block hash value. That root hash value is signed by the user who performs the write (specifically, by the device-specific key through which the write request is made). Other readers verify the signature to make sure that the write request is authorized (i.e. was made by one of the TLF writers): as described in the previous section, the server is supposed to prevent unauthorized writes, but the server is not *trusted*, and clients must validate signatures to make sure that the modification was indeed valid.

While the mechanisms above prevent the server from making unauthorized modifications, the server could still rollback the complete TLF to a previous version or fork the storage, maintaining two or more diverging versions for separate sets of users. KBFS includes several features to prevent such actions:

- Aggressive caching is used by clients, making most rollback actions highly detectable: a malicious server cannot reliably rollback modifications that have been already shown to clients.
- Global Merkle trees are maintained, through which the server commits to a specific version of the whole world at regular intervals (every hour). The successive roots of these trees are published and can be retrieved by any third-party auditor. The illusion of a fork can be maintained for a user only if that user never compares what they see with the tree roots observed by auditors that use a different version.

In order to maintain some level of privacy against traffic analysis, the TLF root values, for purposes of global Merkle tree building, are furthermore encrypted asymmetrically with randomized encryption (ECDH key exchange with an ephemeral key pair, combined with symmetric encryption). External observers cannot know whether or not each newly encrypted root is identical to the previous version. TLF readers and writers have access to the decryption key, that allows them to read and verify the true TLF root value.

Block Encryption

Each block in a non-public TLF is encrypted with a block-specific random key. The encryption format is NaCl's `secretbox`, which includes an integrity check. The block specific key is the XOR of the TLF symmetric key and a newly generated random mask, which is stored on the server. This mask is meant to support some level of enforced deletion; by erasing the mask, the server can prevent decryption of blocks, even if these blocks are stored on a third-party CDN where it cannot guarantee deletion.

Block identifiers (SHA-256 hash values), which are part of the tree up to the TLF root, are not computed on the plaintext for two reasons:

- A hash value of plaintext data can allow exhaustive search attacks on the corresponding data. Real-world sensitive data may have low entropy (e.g. passwords) and be vulnerable to such attacks.
- The server does not have access to the plaintext for non-public TLF, and therefore could not itself verify any hash value computed over the plaintext.

Instead, the hash value is computed over the *ciphertext* (including the random nonce). Unfortunately, the SHA-256 hash is not cryptographically bound to the per-block key. As described in finding NCC-KB2018-005 on page 31, it is possible to compute encrypted blocks that can be decrypted successfully with different keys, but to different resulting plaintexts. The Poly1305 MAC included in the `secretbox` format does not protect against such attacks. This in turn allows an attack scenario in which a hostile client with write access to a TLF may collude with the server to make the other readers of the TLF observe diverging versions of the data contents. This is a relatively low risk issue because the attack setup is quite convoluted, and all but one of the diverging versions will be filled with random bytes.

Assessment Summary

The KBFS protocol applies appropriate cryptographic algorithms to ensure its security properties. In particular, the protocol gives the server no power over the data confidentiality and integrity that the server did not already possess (the server can always block access or delete data, but not without detection by clients). The local caches and global Merkle trees furthermore allow reliable detection of unauthorized alteration, making it infeasible for the server to enact any lasting rollback or fork.

A minor issue on the integrity check was detected, because the SHA-256 hash value is computed over the ciphertext and nonce of a block, but is not bound to the decryption key, which is specific to the block and thus cannot be validated. Some suggestions for a fix are described in finding NCC-KB2018-005 on page 31.

Key Exchange Protocol

NCC Group reviewed the Key Exchange (KEX) protocol,[15] which is used to provision a new device for a user. The safety of this component is critical, as a failure in this section could allow an attacker to add a key to a victim's account and subvert any proof of identity that the user previously relied upon the sigchain to provide.

The KEX protocol involves a user's two devices, one of which is already linked to that user's account. By the end of the KEX protocol, the new device must generate a new key pair, securely transmit the public key to the old device, and receive a version of the public key signed by the old key pair. While both devices in the KEX protocol are considered trusted, they rely upon the untrusted server to transport key material between the two endpoints. In this attack scenario, the servers are capable of observing, modifying, or deleting data that is sent between the two devices. The attacker has the goal of using these capabilities to compromise the confidentiality or integrity of the information travelling between the provisioner and provisionee, since doing so could result in an attacker provisioning a known key to a victim's account. It is therefore imperative that the devices communicate in a cryptographically secure way that resists such transport-channel attacks.

Key Exchange Design

At a high level, the KEX protocol can be split into two stages. First, a shared secret is established between the two devices, called the provisioner and the provisionee. This happens out-of-band and an attacker should not be able to learn this shared secret. Second, the shared secret is used to establish a secure channel between the provisioner and provisionee.

The out-of-band shared secret is shared directly between the two devices using a string of words or a QR code. This is done so that it is easy for a user to transfer the information from one device to the other by using the keyboard or camera on a mobile device. In the KEX protocol, either the provisioner or provisionee may start the secret sharing process. This device selects eight words at random from a dictionary of 2048 words, providing 88 bits of entropy and a phrase that can be easily typed by a human. This phrase is displayed on the generating device along with a QR-encoded version of the phrase. The user enters the secret phrase into the other device so that both devices share the same 88-bit secret. This secret is used to derive a secret session key and a public session ID. The session ID is then sent to the server.

After learning the shared information, the devices then establish a communications channel via the server. The session key is used to encrypt and authenticate the messages, and the session ID is used by the server to route the protected messages to the correct endpoint. Although the server knows the session ID, it should never learn the session key, and therefore never know the contents of the messages. The remainder of the KEX protocol is built on top of this communications channel, and the exchanged messages include the provisionee's new public keys and the provisioner's signature over these values.

Additional details about the KEX protocol can be found in the documentation.[16]

An attacker could target either of these stages of the KEX protocol. If the attacker can tamper with the initial exchange of the shared secret or target the implementation of the established communications channel, this could result in the provisioner adding the attacker's key pair to the account, completely compromising that user's information.

Attacks on the Shared Secret

The first attack to consider is whether it is possible for the server to learn the shared secret. If so, then the attacker can freely impersonate either the provisioner or the provisionee. During the KEX protocol, the server observes a *session identifier* that is deterministically derived from the shared secret. However, this derivation uses scrypt[17] (specifically, an initial secret seed is processed with scrypt to make the shared secret, which is further transformed with HMAC to computed the session ID). Scrypt is not only considered to be one-way, but also includes configurable CPU and RAM "work factors" to discourage brute force attacks. In KEX, the scrypt parameters are:

[15] https://keybase.io/docs/crypto/key-exchange
[16] Ibid.
[17] http://www.tarsnap.com/scrypt.html

- CPU/Memory cost: $N = 2^{17} = 131072$
  (When the provisioner and/or the provisionee is a phone, a "light" version is used with $N = 2^{10} = 1024$.)
- Block size parameter: $r = 8$
- Parallelization parameter: $p = 1$

With such parameters, evaluating scrypt on a single candidate secret requires using 128 megabytes of temporary storage (only 1 megabyte in the light version), which should be enough to make it cost-prohibitive for attackers to build custom hardware which evaluates scrypt on a large number of candidate secrets in parallel.

The shared secret is generated with 88 bits of entropy, so a brute force attack would require on average about 2^{87} evaluations of scrypt to complete. This is not as strong as the standard recommendation of at least 128 bits of security, but still comfortably outside the range of modern attack capability. However, that cost can be shared between attack instances, because scrypt is used here with a *fixed salt*; thus, an attacker may reuse computing efforts for one attack instance on another. Conceptually, the attacker could prepare a large precomputed table of scrypt-derived session IDs for all possible 2^{88} secrets, and then simply look up values in that table whenever a provisioning operation occurs. The salt is a non-secret but variable parameter that is meant precisely to avoid that kind of cost sharing.

Fortunately, a 2^{88} computational effort, for table building, is still a formidable task, beyond the practical reach of today's attackers. Storage of the resulting table, in a format amenable to efficient lookups, would also be very challenging. Moreover, the shared secret only has transient value. In the KEX protocol, it really is used for mutual *authentication* of the provisioner and provisionee, but it is not used for directly encrypting strong secrets. Therefore, breaking that secret is of value to the attacker only if the attack can be conducted while the provisioner is accepting KEX requests from a provisionee. Storage size of a very large precomputed table can be reduced by a factor of t (configurable) using a *rainbow table*,[18] but at an additional $O(t^2)$ computational cost for each lookup; the small attack time frame prevents an attacker from using a large storage reduction factor t.

88 bits are still below the standard recommendation of 128 bits security. NCC Group recommends that a variable salt be added to the scrypt computation for session ID derivation. The salt must be known to both provisioner and provisionee. Ideally, the salt would be randomly generated on one of the two devices and entered in some way by the user on the other device; however, the bandwidth for such an out-of-band value is very limited, and better used as part of the secret. The *user name*, though, could be used: while this is not part of the KEX protocol description, the current client implementations of Keybase require the user to enter the account user name on the provisionee device prior to running the protocol; this is done for usability reasons (e.g. to display a list of the possible provisioner devices). Since two different users, by construction, have different user names, using the user name as salt would nullify the value of precomputed tables for the attacker.

Attacks on the Transport Mechanism

The next attack to consider is if the attacker can successfully modify traffic on the established communications channel. Packets consist of both an unencrypted header and a payload sealed in the NaCl "secretbox" format,[19] using the shared secret as key. Secretbox provides *authenticated encryption*. This means that in addition to the confidentiality provided by encryption, secretbox offers integrity and authenticity guarantees that anyone who receives a message will know its sender and can verify that it has not been altered in transit.

While attackers cannot alter or forge individual packets, they may try to alter the information flow by dropping, duplicating, reordering and rerouting packets. The KEX protocol and implementations apply the following mitigations to prevent such attacks:

- The sealed contents of each packet contain a copy of the ID of the sending device. Each device knows its own ID (which was initially generated randomly when it was first configured). Each device, upon receiving an incoming packet, verifies that the "sender ID" is *not* equal to its own device ID. Since the secret key is known only to the two involved devices, and each device uses its own ID as sender ID, this check is enough to guarantee that the incoming

[18] https://lasec.epfl.ch/~oechslin/publications/crypto03.pdf
[19] https://nacl.cr.yp.to/secretbox.html

packet comes from the other device. This avoids reflection attacks, in which a device is being sent its own packets.

- The sealed contents of each packet contain a sequence number. The first packet sent by a device during the KEX protocol contains the sequence number 1, and that number is incremented for each subsequently sent packet. These properties are checked on the receiving side. This guarantees detection of dropped, duplicated, and out-of-order packets.

These two verifications protect the protocol against alterations of the flow of information. Note that we use here the assumption that the secret key is known only to the two devices. If a secret value is reused for another KEX instance, then messages from the first instance could be replayed in the second instance. This is an improbable event: such a secret collision is expected to happen after 2^{44} provisioning instances, a very large value that is unlikely to be reached in the foreseeable future. If the user name is also used in the scrypt derivation, as recommended in the previous section, then the probability of collision is considerably lower (it would require about 2^{44} provisioning instances *for the same user*).

In the packet format, there are *two* headers, one inside the sealed contents, the other on the outside. The sender ID, session ID, and sequence number are present in both the inner and outer headers; the receiver checks that the inner and outer headers match on each incoming message. The security guarantees offered by the protocol come from the protection offered by the secretbox sealing mechanism on the inner header; the outer header must be thought only as a routing mechanism, not a security feature.

Contrary to other transport protocols such as SSL/TLS, there is no cryptographically protected end-of-transmission packet. An attacker that controls the transport network (in particular the server itself, since it relays all packets) may cut the connection at any moment. However, the protocol does not confer any semantic value to the end-of-transmission. At worst, an attacker who controls the network can prevent the provisioning process from completing (this is unavoidable), or block the last message, thereby making the provisioner assume that the protocol failed, while the provisionee has been properly provisioned. Since both provisioner and provisionee devices are under physical control of the user during the operation, that kind of half-failure is not considered a security concern.

Alternate Design: PAKE

Password Authenticated Key Exchange (PAKE) is a type of key exchange protocol that provides mutual authentication with regards to a possibly low-entropy shared secret. PAKE is noteworthy due to its immunity against offline dictionary attack: eavesdroppers, and even active attackers, do not gain enough information to test potential values for the shared secret without interacting with one of the parties for every secret candidate. The original design is EKE, published in 1992.[20] Patents were issued at that time but have expired since.

With EKE, the provisioner and the provisionee could perform their key exchange without needing scrypt-based stretching; the accumulated computing power of the attacker, and precomputations, would become irrelevant. An EKE protocol based on elliptic curves can be sketched as follows:

- An elliptic curve E is used; it has order cn, where n is a big prime integer, and c is the "cofactor", a usually small integer (for the well-known curve edwards25519, as described in RFC 7748,[21] n is close to 2^{252}, and $c = 8$). A fixed conventional generator point G of order exactly n is defined.
- A hash function H is defined, that takes as input bit strings, and outputs a curve point in the subgroup of order n. An important required characteristic of H is that the discrete logarithm of the outputs be unknown. A suitable definition of H for curve edwards25519 is as follows:
  - Input is a bit string S.
  - Set a counter $k = 0$.
  - Compute t = SHA-256($S \parallel k$), where k is encoded over a single byte.
  - Interpret t into a curve point P using the decoding process outlined in RFC 8032, section 5.1.3.[22]
  - The decoding of t may fail (with probability about 0.5); in that case, increment k and try again. Otherwise, return

[20] *Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks*, https://www.cs.columbia.edu/~smb/papers/neke.pdf
[21] https://tools.ietf.org/html/rfc7748
[22] https://tools.ietf.org/html/rfc8032

$H(S) = cP$ (multiplication of the decoded point P by the cofactor).
On average, only two iterations are needed to find a suitable point; probability of doing more than r iterations is about $2^{-(r+1)}$. Implementations may include an exit condition if k reaches 256, but this condition would not be reachable in practice.

- The shared secret is S.
- The provisioner generates a random non-zero integer a modulo n, and sends $A = aG + H(S)$ to the provisionee.
- The provisionee generates a random non-zero integer b modulo n, and sends $B = bG + H(S)$ to the provisioner.
- The provisioner computes the curve point $a(B - H(S))$. The provisionee computes the curve point $b(A - H(S))$. If the provisioner used the same secret S, then they both compute the same point abG, which they thereafter use as shared high-entropy secret value.

Use of this EKE design in the KEX protocol would involve the following:

- The "session ID" I would not be derived from the shared secret. Instead, it can be made of the combination of the user name and the provisioner device name (or a hash thereof): the provisioner has by definition such information, and the current implementation of the provisionee client also asks for the user name and provisioner device name prior to engaging in that protocol.
- After the initial notification from the provisionee ("`Start`"), but before the "`Hello2`" message, an initial pair of unencrypted messages is sent: The provisioner sends its EKE point A, and the provisionee responds with its EKE point B.
- Subsequent messages (starting with the "`Hello2`" message) are encrypted not with the shared secret S, but with the shared high-entropy secret value obtained from EKE.

With these modifications, the only values that are sent and depend on the secret S are the EKE points A and B, and these do not allow for offline dictionary attacks since they are "masked" by the ephemeral Diffie-Hellman points aG and bG.

Assessment Summary

The KEX protocol, as described, seems to provide the expected guarantees, subject to the following notes:

- As explained above, the derivation of the shared secret, from the initial seed conveyed out-of-band by the user (sequence of random words, typed or sent as a QR code), should use a salt value to prevent cost sharing by attackers. The user name, in particular, could conveniently be used as a salt, since both devices already know it.
- The verification that an incoming message's sender ID in distinct from the device ID of the receiver is crucial for security against reflection attacks. The documentation does not require this check, but the client implements it.
- Use of a PAKE protocol could allow for stronger resistance against brute force attacks on the shared secret S, even if that secret had lower entropy than its current 88 bits. It would also remove the need for key stretching with scrypt, an operation that can be expensive on small portable devices.

NCC Group reviewed the cryptographic aspects of Chat[23] and Exploding Messages,[24] which allow users to communicate over an encrypted channel and set parameters around how long messages are retained. This system must fulfill the following four properties, where the fist two relate to Chat and the second two focus on Exploding Messages:

- The protocol utilizes the same keys as defined for the KBFS mechanisms.[25]
- Messages shall include information on the state of previous messages in order to prevent re-ordering by a malicious server.
- Each user shall have the ability to delete messages by issuing a delete command.
- Users can create messages that have a limited time-to-live; each device shall delete messages whose time-to-live has expired.

Chat Encryption

The chat encryption algorithm uses two formats, however as $MessageBoxedV1$ is no longer written by clients and only used for legacy purposes. Only $MessageBoxedV2$ will be described here. Each message body is encrypted with a shared key as well as a random 24-byte nonce. This is then serialized with $MessagePack$ and a SHA-256 hash is taken of the $version||nonce||ciphertext$. This hash is then added to the message header and serialized. The header, which includes the hash, previous message references, and other metadata, is then $signencrypted$ using another random nonce. Finally, the header plaintext, header ciphertext, encrypted message body, and both nonces are sent to the server. The sign-then-encrypt construct is used to keep the signature private to only involved parties that share the session key as well as ensure that users cannot impersonate each other. The header is sent over as both plaintext and ciphertext for multiple reasons. The plaintext informs the server where to route the message and the type of message. However, this plaintext could be tampered with, so the server can't prove that the chat participants in the header actually constructed the message, offering privacy. The ciphertext allows the recipient to check the information in the plaintext header by decrypting and verifying the signature.

By default, chat messages and the keys protecting them live on both the server and user devices indefinitely. They are only removed if any party to the conversation adds a device, removes a device, or sends the other a message delete request. The message metadata persists even if the message data is deleted with one of the above methods. This data could allow for the post hoc reconstruction of the communication histories of those who have deleted their messages. The default chat protocol does not allow for forward secrecy since the same keys can retain indefinitely on a users device. Additionally, due to the nature of signatures, the protocol does not offer deniable authentication. This means it is possible to construct who is sending messages on the service even if these messages have been deleted.

Exploding Messages

Keybase allows users to create messages that are deleted after a configurable amount of time (currently anywhere from seven days to 30 seconds). To support this sort of ephemeral messaging, each device creates a daily ephemeral encryption key pair that has a validity of seven days. The public half of this keypair is signed with each device's long-term signing key and sent to the Keybase server. The private half is then encrypted with each device's public ephemeral key so that all of a user's active devices know the same ephemeral private key. At the end of the seven-day validity period, the ephemeral keys are deleted and cannot be used to decrypt any stored encrypted messages. However, note that ephemeral keys persist for the full seven days even if they are used to protect exploding messages with a lifetime of less than seven days. This means that the decryption keys for an "exploded" message can stick around much longer than the message, and are at greater risk of compromise. Due to issues with message authentication in large chats, pairwise-MACs are used for exploding messages in teams of 100 members or fewer, and signing is used for teams above that number. This creates a situation where smaller team chats provide deniable authentication because MAC keys are forgettable, whereas larger team chats are signed with the user's long-term signing key, which is neither deniable nor forgettable. Although users can choose to actively delete messages, time-based exploding messages rely on each device's local clock. This could create a situation where each device deletes the message at a different time.

[23] https://keybase.io/docs/crypto/chat
[24] https://keybase.io/docs/crypto/ephemeral
[25] See Keybase Filesystem on page 12 for details.

Assessment Summary

The Chat encryption and exploding message features were found to have the characteristics described in Keybase documentation with the following observations:

- While the default Chat encryption protocol does provide for message confidentiality and integrity, it does not provide for security in the face of device and server compromise, as keys and ciphertext are stored for a potentially indefinite period of time. Exploding messages introduce mechanisms for message deletion and forward secrecy; however, it is not clear to the user that keys and messages could remain on their device beyond the period specified during message creation. It would be reasonable for a user who creates a message that should delete after 30 seconds to assume that its session key would not be retained for another seven days. One way to mitigate this is to create more transparency within the application UI and to allow for more fine-grained modes. An example of this could be a 'paranoid' mode that would enforce a much shorter key lifetime.
- As the deletion of exploding messages on devices is determined by the local device clock, a nefarious device with compliant software could keep messages from being deleted , which would create differing deletion times and contravene the original intent of the sender.[26] This could be somewhat mitigated if the original sender issued a delete command when the timer on the local message expires.

[26] See for example the COMPROMISE #2 scenario of https://keybase.io/blog/keybase-chat

Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. For an explanation of NCC Group's risk rating and finding categorization, see Appendix A on page 34.

Title	Status	ID	Risk
Signature ID Blacklist Bypasses Verification	Fixed	001	High
Maliciously Crafted Packets Can Crash Clients and Servers	Fixed	002	Medium
Weak Password Policy	Fixed	003	Medium
Ambiguity in Signature Payload Interpretation	Fixed	004	Medium
KBFS Blocks Can Be Forked	Fixed	005	Low

Finding	**Signature ID Blacklist Bypasses Verification**
Risk	High Impact: High, Exploitability: Medium
Identifier	NCC-KB2018-001
Status	Fixed
Category	Data Validation
Component	Client
Location	`go/libkb/sig_chain.go:591`

Impact A compromised server can selectively exclude chain links from a victim client's sigchain verification. If the targeted sigchain includes revocation of a compromised key, the server can leverage this vulnerability to prevent clients from recognizing the revocation and to add additional chain links using the compromised key.

Description As clients verify the sigchain, they begin at the oldest link in the subchain and replay all links since that point. Each chain link includes declarations about the delegation or revocation of new keys, follower statements, cryptocurrency addresses, and so on. As the subchain is replayed, the client applies these declarations to its internal model of the user, verifying that each statement was signed by a delegated and non-revoked key at that point in time.

However, some subchains contain chain links with invalid data. These were accepted by the server due to various server bugs between March 2015 and September 2018[27] and incorporated into the immutable record. Once the bugs were identified, the invalid chain links could not be removed, and clients needed to be modified to prevent acceptance of the invalid links. This is done by hardcoding the signature ID in a blacklist and skipping these chain links during the verification process. This can be seen in the following snippet from `sig_chain.go`:

```
for linkIndex, link := range links {
    if isBad, reason := link.IsBad(); isBad {
        m.CDebugf("Ignoring bad chain link with sig ID %s: %s", link.GetSigID(),
➜  reason)
        continue
    }
```

However, `IsBad` only uses the chain link's signature ID to determine if the chain link is valid; this value is sent by the server and not verified by the client. This means that when a client requests the sigchain of a Keybase user, the server can arbitrarily replace one of the signature IDs of the legitimate chain links with one of the blacklisted values, even if the chain link is unrelated to the prior server bugs. The victim's client will recognize the modified signature ID as blacklisted and skip whatever declarations were made by that chain link. Since the signature ID is not used when computing the sigchain hashes, the sequence of payload hashes and signatures still pass sigchain validation; the only difference is that the contents of one of the chain links is skipped during verification.

Although the server cannot use this directly to make false statements about a Keybase user, it can do so indirectly. Assume the targeted user has added a new sibling key to their sigchain, later revoked the key, and then the private key is somehow leaked. This may happen for a

[27] https://github.com/keybase/client/blob/049d28bddb3c71cdd986f52d0f9fd9344d17a713/go/libkb/chain_link.go#L127-L149

variety of reasons, such as a user realizing one of their devices is compromised or likely to be compromised and wanting to minimize the impact. Since the key has been revoked, the Keybase user expects that knowledge of the private key can no longer be used to modify information about the user.

In this scenario, the server could use the leaked sibling key to forge and append identity statements to the user's sigchain, but because the key was revoked before the forged statements were appended, a client is able to recognize the added statements as invalid. However, if the server uses this signature ID vulnerability against a different Keybase user in the process of verifying the targeted user, the server causes the victim to ignore the chain link that revokes the leaked key, and the victim accepts the forged statements as valid. Note that while the attack would succeed in the moment, if the user verifies the forged statement is included in the immutable public record, then the attack could theoretically be detected after the fact by a third party auditing the immutable record.

This behavior was verified by intercepting and modifying the sigchain data requested by the Keybase command line client and observing that the client completes verification successfully but does not properly revoke the proper key when processing the modified data. For more details, refer to the reproduction steps in Appendix C on page 38.

It is currently believed that only a compromised server (or equivalently a network-based attacker who has compromised the TLS connection between the victim client and server) is capable of exploiting this issue. It does not currently seem possible for one Keybase user to use this issue to target another.

Reproduction Steps Detailed reproduction steps can be found in Appendix C on page 38.

Recommendation Clients should perform more strict checks on signature ID. If the signature ID is a value cryptographically derived from the signature, such as a hash, the value should be checked when verifying the chain links are in the correct order and belong to the correct user. If the signature ID is a random identifier that is not tied to the chain link contents, further verification steps are necessary, such as checking the username, user ID, and sequence number.

Note that the signature ID is compared against a hardcoded list in other areas as well, such as the `hardcodedResets` array. Any mitigation for this issue should similarly be applied to these other cases to prevent the possibility of a similar issue elsewhere.

Retest Results Keybase has patched[28] the issue by performing more rigorous checks for the bad links. The links now are only excluded if they match the user ID, sequence number, and link ID, which a malicious server is unable to spoof.

[28] https://github.com/keybase/client/commit/3dd51645fd1878884eb11892217e362556e3b5ee

Finding	**Maliciously Crafted Packets Can Crash Clients and Servers**
Risk	Medium Impact: Medium, Exploitability: High
Identifier	NCC-KB2018-002
Status	Fixed
Category	Denial of Service
Component	Serialization Library
Location	`keybase/go-codec` library
Impact	An unauthenticated packet can crash the whole receiving process on either the client or server. This denial-of-service attack prevents legitimate users from interacting with the service.

Description

Keybase messages use two serialization formats for messages; depending on context, some use MessagePack,[29] while others rely on JSON.[30] Both formats support nested structures to arbitrary depths. For MessagePack messages, Keybase uses a generic library,[31] which is itself an import from a well-known third-party library.[32] The library works by decoding streamed data and mapping the resulting objects onto the fields of a caller-provided typed object. Mapping is done by name where explicit names provided in the encoded structure are compared with the field names (or appropriate type annotations) of the destination in-memory object. By design, unknown elements in the incoming structure are ignored and skipped silently; this is meant to support future protocol extension in a backward-compatible way.

In order to properly locate the start and end of a structure element, the decoding library must maintain some transient state. In practice, the decoding function is recursive; a structure of depth d will involve recursive calls of depth d within the decoding library, and thus use of $O(d)$ bytes of stack space. By sending a deeply nested extra element in an otherwise normal message, an attacker can force the recipient to exhaust its stack space, and trigger a stack overflow condition. In practice, this means adding the following bytes in a message, between two normal, valid fields: the bytes `0xA1 0x7A`, followed by d bytes of value `0x91`, then one final byte of value `0x90`. These $d+3$ extra bytes encode a structure field of name `"z"`, whose value is a nested array of depth d (an array that contains an array that contains an array, and so on; at the deepest level, there is an array of length zero).

On a 64-bit x86 system (running Ubuntu 18.04), the default stack space in Go is one gigabyte. That space is exhausted with a call depth of about 3.73 million. This means that if $d \geq 3730000$ in the constructions above (requiring a 3.73 MB MessagePack object), then a stack overflow condition is triggered:

```
runtime: goroutine stack exceeds 1000000000-byte limit
fatal error: stack overflow
```

Most importantly, this is a "fatal error", not a "panic": It cannot be caught and recovered from using deferred functions. When such a condition occurs, the whole process terminates (not only the offending goroutine).

[29] https://msgpack.org/
[30] https://www.json.org/
[31] https://github.com/keybase/go-codec
[32] https://github.com/ugorji/go

On a 32-bit x86 system (again running Ubuntu 18.04), stack usage per recursive call is lower, thanks to the lower size of native pointers; however, the stack space is also reduced, down to about 250 megabytes, allowing the attack to work with a lower depth of about 1.98 million. On such a target system, a 1.98 MB MessagePack message is sufficient to trigger a stack overflow.

Additional Considerations:

- Even if hard limits are enforced on message size, below the sizes required by the attack above, nested messages can still imply substantial transient memory usage. For instance, on a 64-bit x86 system, a one-megabyte message can still force temporary usage of about 268 megabytes of RAM; this is a large amplification factor, that can be leveraged to bring down a server by RAM exhaustion with relatively few medium-sized requests.
- Other deserialization layers may suffer from the same issue. For JSON objects, Keybase currently uses, depending on context, the JSON decoder integrated into Go's standard library (`encoding/json`), and `buger/jsonparser`.[33] The latter seems by design immune to such attacks: it performs recursive exploration only when explicitly instructed to do so with path, and only up to the depth specified by that path. However, `encoding/json` may reach a stack overflow condition (at depth about 2.69 million, for a 5.38 MB malicious JSON input) when decoding into an open-ended map (`interface {}`, with no schema). The Keybase code base should also be checked for such open-ended JSON decoding.

Recommendation

Generally, it is not possible to make the decoding library immune to such attacks, as long as arbitrary depth is supported. In particular, the MessagePack format encodes arrays and maps with an explicit length in the header; a decoder that can process streamed data must then necessarily store $O(d)$ information for a nested depth of d.

NCC Group recommends that an explicit maximum depth be set in the decoding library. The library does not currently support such a feature. The amount of work needed to enforce a maximum nesting depth in the library is not known at this point.

Retest Results

Keybase implemented in `keybase/go-codec` explicit checks on current decoding depth, with a default maximum depth of 100; any incoming message that exceeds that depth triggers an exception (a "panic" in Go terminology) which is converted into an explicit error message. Additional safeguards were implemented in order to avoid abnormally large transient allocations on malicious input. This fixes the issue reported here. Relevant commits are:

- d7adf74c7df69e09ef331d136ea80c35a79b5aa9
- 2270557d28a6bf5ec3f748eedccd9bc8e318180a
- a7011fe42ddcc32e5b55a8a09dd03e3431a533e5
- 1643975621235e11516b5722dd9475e342465adf

[33] https://github.com/buger/jsonparser

Finding	**Weak Password Policy**
Risk	Medium Impact: Medium, Exploitability: Medium
Identifier	NCC-KB2018-003
Status	Fixed
Category	Data Validation
Component	GUI Client
Location	• client/shared/settings/passphrase/index.js:53 • client/go/libkb/checkers.go:46
Impact	Users may choose very weak passwords, allowing attackers to access and reset their web account. Weak passwords also mean poor protection for local keys.

Description

When a user wants to change their password in the client application or the phone app, the interface applies a six-character minimum size for password. This is the only rule applied by the interface, as shown in the `UpdatePassphrase._canSave()` function:

```
  _canSave(passphrase: string, passphraseConfirm: string): boolean {
    const downloadedPGPState = this.props.hasPGPKeyOnServer !== null
    return downloadedPGPState && passphrase === passphraseConfirm && this.stat
➜   e.passphrase.length >= 6
  }
```

A similar check is done in the command-line client:

```
var CheckPassphraseNew = Checker{
        F: func(s string) bool {
                r := []rune(s)
                if len(r) > 0 && unicode.IsSpace(r[0]) {
                        return false
                }
                return len(s) >= MinPassphraseLength
        },
        Hint:           fmt.Sprintf("passphrase must be %d or more characters", M
➜  inPassphraseLength),
        PreserveSpace: true,
}
```

The `MinPassphraseLength` value is defined in `constants.go` with a value of 6.

Six-character passwords are very weak, especially since, when given the possibility, most users tend to use common words as passwords. There are about 15000 six-character words in English, including some uncommon ones such as “quippu” or “hexose.” Even with the addition of names of places and people, and “simple” sequences such as “zxcvbn” (consecutive letters on a common keyboard), it can be estimated that most users will choose their passwords among a list of less than 30,000 possible values. These users will also avoid adding case variations, because an all-lowercase password is much easier to type, especially on mobile devices with a “visual keyboard.”

An attacker guessing a user’s password can access the web account and reset it, thereby breaking the functionality for the user, and banning the user’s account name forever. The user’s password is also used for protecting local keys: If a user device is stolen, then the user's password is the only remaining protection against key compromise or even full account hijacking (If the user cannot revoke the stolen device before the attacker guesses the user's password, then the attacker can then revoke all other devices, and essentially lock out the user from their own account.)

Recommendation The user interface should enforce a stronger password policy, e.g. a minimum password length of 10 characters.[34] Additionally, the same user interface should offer a password generator that creates strong memorizable passwords (e.g. passwords consisting of four words from the BIP0039 dictionary of common words). Ultimately, the user is responsible for choosing and remembering a high-entropy password, but the interface should provide guidance and tools.

Retest Results Keybase has increased the minimum passphrase length from 6 characters to 8. These changes are found in the Keybase client pull request 13916, with the following commits:

- a8c4222e835742e33fb87df37dd5917fe08c3c6f
- b11f5059e2f21668f737e6afdd3a899629bea6ae
- effe7f3223d30de0d43aaafe2407417207982d4e
- 86df80c84c6a90f5220afba61801d3d4140b72fb

[34]Some best practices on password creation and management are detailed in NIST Special Publication 800-63-B.

Finding	**Ambiguity in Signature Payload Interpretation**
Risk	Medium Impact: Medium, Exploitability: Medium
Identifier	NCC-KB2018-004
Status	Fixed
Category	Cryptography
Component	Client
Location	• `(NaclSigInfo).Verify` in `libkb/naclwrap.go` • `(*ChainLink).VerifyLink` in `libkb/chain_link.go`
Impact	A malicious client and server can collude together to make valid sigchains with multiple valid interpretations, giving two victim users inconsistent views under the same Merkle tree. This sort of forking violates a design goal of the Merkle tree, and it enables an attacker to deceive a trusted user into attesting an innocuous interpretation of the attacker's sigchain while claiming the attestation was over the malicious interpretation.

Description

Chain link signatures currently cover one of two types of payloads. Version one (V1) payloads are JSON objects and version two (V2) payloads are msgpack objects, and a hash of the payload is used to build the sigchain. However, some extra information must be used to determine whether to interpret the payload as V1 or V2. This information cannot come from any information outside of the payload, because this is unsigned and unused in the chaining hashes and therefore can be modified by the server. This means that the information must come from the payload itself, which requires knowledge of how to interpret the payload.

Therefore, it is theoretically possible a sequence of bytes exists that can be validly interpreted as both a JSON object and msgpack object. The JSON-decoding of these bytes could report that the payload is V1, and the msgpack-decoding could report that the payload is V2, so the client cannot rely on the content of the payload to verify that the payload was correctly decoded. This leads to potential ambiguity in signatures that verify correctly but have multiple interpretations based on unverified version information in the sigchain. A misbehaving attacker colluding with the server could craft such a polyglot, request a trusted user to follow the attacker's sigchain based on one interpretation, then present the second interpretation to a third user, who trusts the follower statements.

Additionally, there is a related issue in `naclwrap.go:535–543` where a signature may be over a server-controlled prefix concatenated with a null byte and the chain link payload.

```
switch s.Version {
case 0, 1:
    if !key.Verify(s.Payload, &s.Sig) {
        return nil, VerificationError{}
    }
case 2:
    if !key.Verify(s.Prefix.Prefix(s.Payload), &s.Sig) {
        return nil, VerificationError{}
    }
```

This is problematic because multiple combinations of prefix and payload can both be validly signed by a single signature. For example, the server could present a chain link with prefix `aa`, payload `bb00cc`, and a signature over `aa00bb00cc` and the signature would verify. The server could also present a chain link with prefix `aa00bb`, payload `cc`, and the same signature over aa00bb00cc. It is impossible to distinguish from the signature alone whether the intended payload was bb00cc or cc, and these payloads may have different meanings.

This second, prefix-related issue will be difficult to exploit in practice, because it requires two different payload sequences, which means the payload hash will be different and can be caught by the Merkle tree. It is pointed out because there should be zero uncertainty in what the signer intended to cover in the signature.

On the other hand, the first issue related to a msgpack/JSON polyglot is practical to exploit. It is possible to construct a sequence of bytes (as detailed in Appendix D on page 39) that is validly interpreted by both the Keybase msgpack and JSON decoders. For the most part, these structures can contain arbitrary data. There is a small limitation regarding which bytes are allowed to appear in the msgpack hash fields, but initial testing shows that brute-forcing hash values that satisfy this requirement would be very easy to do, and is in line with the assumption of a malicious client.

While polyglots are easy to construct and this issue can effectively be used to fork the interpretation of a sigchain, this issue is made less severe in two ways. First, it requires a compromised server to present different views to victim clients, and second, the attacker can only cause an inconsistent view of the attacker's own sigchain, not the sigchains of others.

Recommendation Signatures should be completely unambiguous in how they are interpreted. To allow for flexibility in format, one option is to have one field in the payload that indicates the version and format, and the rest of interpretation depends on this value, but all signatures should contain this field.

Since there is no practical way to revoke existing signatures with this potentially ambiguous format, one possible remediation is to verify that the payload being verified cannot be interpreted under another payload format. For example, if the signature version provided by the server indicates a V2 payload, confirm that the payload can be validly decoded as a V2 payload but cannot be validly decoded as a V1 payload. This will catch a malicious server attempting to provide such a polyglot to the client.

To help mitigate the prefix issue, the verifier should compare the received prefix against a whitelist of values, since this would considerably increase the complexity of an attack on the syntactic ambiguity of signatures.

Retest Results Keybase has patched the prefix issue by comparing the received prefix against a whitelist of 6 values, with a seventh value for testing in development builds. The polyglot issue has been patched by making polyglots impossible. `getSigPayload()` now calls `assertCorrectSigVersion`, which compares the expected signature type against the first payload byte. JSON objects are rejected if they do not start with `0x7B`, and msgpack objects are rejected if they do not start with `0x90` through `0x9f`, `0xdc`, or `0xdd`. These are mutually exclusive events, so a signed payload cannot be successfully interpreted in both ways.

These changes are found in the Keybase client pull requests 14320 and 14006, with the following commits:

- b31c274a80d383710e369a9d07a9ddc58da1249c
- 6aa1fa8b9aad81639d1d49d885afaa7f6923190f
- 31103ac92b8c4024c139fbafbc720c039dfdd6be
- 81842779a86f4974c6d41d2eb64486e41c40aac3

Client Response Keybase initially responded that prefix whitelisting was already in place, preventing the prefix-modification attack. This prompted further testing against version 2.8.0 of the client, where the whitelist in `kbcrypto/signature_prefix.go` was found to be unused during sigchain verification, and prefix-modification was demonstrated using the man-in-the-middle techniques of Appendix C on page 38. NCC Group used the man-in-the-middle proxy to add `version` and `prefix` fields to the `NaclSigInfo` structure, which was Base64- and msgpack-encoded in the `sig` field of the first chain link. NCC Group also updated the `hash` field in the surrounding `keybasePacket` structure to account for the modified signature. The modified and re-encoded value was returned to the client, which continued processing it as if the response had come from the server. The modified payload successfully passed initial validation, reaching the `Verify` method in `kbcrypto/nacl_signing_key.go`, where it was confirmed that the arbitrary prefix and null byte was successfully injected into the message awaiting verification. This demonstrated how a malicious server could return modified data and perform the prefix-modification attack above.

Finding	**KBFS Blocks Can Be Forked**
Risk	Low Impact: Undetermined, Exploitability: Low
Identifier	NCC-KB2018-005
Status	Fixed
Category	Cryptography
Component	KBFS
Location	https://keybase.io/docs/crypto/kbfs, section 4.1.2
Impact	A hostile client with write access to a private folder can collude with the server to present different versions of the file contents to other writers and readers of that folder, which contradicts the consistency requirements of KBFS.

Description

Each file block in KBFS is encrypted with a block-specific key. The writer who creates the block generates a random symmetric key k (32 bytes) and nonce n, and applies NaCl's `secretbox` on the block plaintext. The server stores a *mask value* equal to $k \oplus k_f$, where k_f is the key associated with the top-level shared folder and known to all readers and writers of that folder. When a reader needs to access the data, it obtains the mask from the server, and recombines it (XOR) with k_f to recompute the key k; with the key k, the secretbox can be opened. Opening the secretbox entails both decrypting the value, and verifying the Poly1305 MAC, which was computed when the secretbox was sealed.

Integrity of the data (both the block data itself, and the overall folder hierarchy structure) is ensured in the following way: Each block is identified by its ID, which is the SHA-256 hash of a structure that contains the nonce n, and the secretbox data (ciphertext and Poly1305 tag). The block ID is then referenced in the block that describes the containing directory; this is done recursively up to the top-level folder, whose block (called "root") is signed by the writer. This structure is meant to "keep the server honest", as the documentation states it.

However, it is possible to create a pair of keys (k, k') along with a nonce n and secretbox data, such that the secretbox can be successfully opened with both keys (the Poly1305 tag will match in both cases), but the resulting plaintexts will differ. A hostile client with write access can create such a block, and collude with the server to serve other readers and writers the mask value for either k or k'. Readers who obtain the mask for k will decrypt the block data into a plaintext distinct from the plaintext obtained by readers to whom the mask for k' was sent. Since the same nonce and secretbox data are used, the change of key does not impact the block ID; all cryptographic validations will pass. In effect, the hostile client and the server can together maintain a *fork* of the file contents, with two distinct versions that other clients accept as valid.

The mathematical details of the attack are as follows. NaCl's `secretbox` works in the following way:

- From the key k and nonce n, a pseudorandom stream of bytes is generated, of length $n+32$ bytes, where n is the length of the plaintext (in bytes).
- The last n bytes of the stream are XORed with the plaintext, yielding the ciphertext of length n bytes.
- The first 32 bytes of the pseudorandom stream are split into two 16-byte keys r and s (a few specific bits of r are cleared for historical implementation reasons). (r, s) are the Poly1305 keys.

- Poly1305 is applied on the ciphertext: the ciphertext is split into 16-byte blocks, each block being interpreted as an integer c_i in the 2^{128} to 2^{129} range. The Poly1305 output (called the "tag") is $t = (\sum_i c_i r^{(n/16)-i} \bmod 2^{130} - 5) + s \bmod 2^{128}$.
- The secretbox data is the concatenation of the tag (16 bytes) and the ciphertext (n bytes).

The attacker's goal is to obtain a ciphertext c such that the tags with keys (r, s) (derived from k) and (r', s') (derived from k') are identical; in that case, the block consisting of the nonce n, common tag t, and ciphertext c, can be submitted to the server. This means finding a ciphertext c such that:

$$(\sum_i c_i r^{(n/16)-i} \bmod 2^{130} - 5) + s \bmod 2^{128} = (\sum_i c_i r'^{(n/16)-i} \bmod 2^{130} - 5) + s' \bmod 2^{128}$$

If we single out a specific ciphertext block c_j, then this equation becomes:

$$(r^{(n/16)-j} - r'^{(n/16)-j})c_j = \sum_{i \neq j}(r'^{(n/16)-i} - r^{(n/16)-j})c_i + \delta$$

where all computations are performed modulo $p = 2^{130} - 5$ (which is prime), and δ is one of the values modulo p that, when taken modulo 2^{128}, are equal to $s' - s$. Since p is very close to 2^{130}, it is expected that there will be about four possible δ values on average. For each matching value δ, the equation above yields a corresponding ciphertext block c_j. With probability about $1/4$ each time, that value c_j will fall in the appropriate range for a ciphertext block (i.e. between 2^{128} and 2^{129}). If the c_j computed above falls in the proper range, then it suffices to replace the corresponding block in the ciphertext to obtain a solution.

The attack scenario is the following:

- The attacker chooses a "meaningful" plaintext m, in a format such that one 16-byte block can be modified at will (this requires that the data format includes some unstructured bytes or an "ignored" area; some text-based formats, for instance, can have "comments").
- The attacker selects random keys k and k', and a random nonce.
- The equations above are applied to obtain the altered ciphertext. When decrypted with k, this yields m back, save for the modified block, which will contain random bytes. When decrypted with k', this yields a message m', whose contents are essentially random.
- If the equations fail to find a ciphertext block c_j in the proper range, or if the alternate plaintext m' is not meaningful enough for the application that will interpret it, the attacker tries again with a new random key k'.

Exactly how many iterations the attacker will have to go through depends on the notion of "meaningful" for the consumer application. Attack difficulty can range from "trivial" to "infeasible."

Recommendation The conceptual problem in the protocol can be summarized as follows: The integrity check aims at guaranteeing the plaintext contents; however, the block ID is not computed over the plaintext (which is inaccessible to the server) but on a subset of several elements from which the plaintext can be recovered (nonce, ciphertext, ...). Unfortunately, one of these elements (the decryption key itself) is not used in the block ID computation, and can therefore be changed without detection. The Poly1305 MAC computation that is part of `secretbox` is not designed to offer any protection against collisions by an attacker who knows the involved secret keys: The attack described here shows how to obtain tag collisions efficiently.[35]

The solution is to cryptographically bind the decryption key within the block ID computation. The server must not be shown the value of the per-block key; however, that key is random and large, and therefore immune to brute force attacks. It is then safe to include in the block ID a value derived from the key with a one-way function.

A proposal that emerged from discussions between NCC Group and Keybase is the following: From the per-block key k, compute SHA-512(k); the first 32 bytes of that value will be used as secret key for `secretbox` (instead of k itself), while the next 24 bytes will be the random nonce. The nonce is already part of the input to the block ID computation, so the on-disk format is unchanged. Readers must then recompute the nonce from k and verify that the value in the block header matches the recomputed nonce.

The problem of how to manage the transition to the new format, in particular preventing clients from creating old-version blocks but without losing access to old data, remains under investigation.

Retest Results

Keybase implemented the recommended fix (use of SHA-512 over the key to generate both the encryption key and the nonce, and verification of the nonce computation upon decryption) in a series of commits grouped in two pull requests:

- PR 1836: https://github.com/keybase/kbfs/pull/1836
- PR 1837: https://github.com/keybase/kbfs/pull/1837

A new block format version (V2) was defined for this new method, and is enforced through the block ID: a V2 block ID cannot be used with a V1 block format.

[35] If, instead of using Poly1305, the encryption format used HMAC, then there would be no efficient method to compute tag collisions, and the attack would not work. The HMAC output would have to be at least 256 bits, to achieve the usual 2^{128} security level. Collision resistance against attackers who know the key is not part of the definition of a MAC algorithm; however, HMAC offers that extra property.

Appendix A: Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.
Medium	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.

Appendix B: Documentation Issues

This section lists discrepancies between documentation and implementation, and omissions from the documentation. These discrepancies are not security issues—the implementation is robust, even where it does not match the documentation. Nevertheless, the documents should be updated to reflect the current state of the implementation.

Keybase Key Exchange (KEX) Protocol

Document URL: https://keybase.io/docs/crypto/key-exchange

- The initial secret W is defined as a list of eight random words, each word being randomly selected in the BIP0039 dictionary (a ninth word of value "`four`" is appended if either of the involved devices is a phone). The documentation does not specify how a list of eight words becomes a single sequence of bytes; the implementation concatenates the words with ASCII spaces (`0x20` character) as separators (a single space is added between any two consecutive words, but there is no leading or trailing space), then encodes the whole string in ASCII.
- The session ID is obtained from the shared secret S by using HMAC/SHA-256, with S as key, over a conventional fixed string. The documentation states that the string is "`kex2-session-identifier`". However, the implementation uses the string "`Kex v2 Session ID`".
- The documentation states that the QR code Q encodes the derived secret S. However, the implementation encodes the 9-word sequence W in the QR code, not the derived secret S.
- In the transport protocol, each message has an outer header (unencrypted) that contains the sender ID (ID of the sending device), the session ID, the message sequence number, and a random nonce. The encrypted payload also has an inner header that contains a copy of the sender ID, session ID, and sequence number. The documentation states that the receiver must verify that the sender ID, session ID and sequence number values from the inner header are equal to the corresponding values in the outer header. However, there are other checks that are important for security:
  - The sequence number of the first received message must be 1, and for all subsequent received messages, the sequence number must increment monotonically by 1.
  - The session ID must match the session ID value derived from the shared secret S.
  - The sender ID in a received message must not be equal to the device ID of the receiving device (this protects against reflected messages).

  These checks are implemented in the code but not specified in the documentation.

Per-User Keys

Document URL: https://keybase.io/docs/teams/puk

- A number of mathematical elements are typeset with extra underscores, and look like "_E_" instead of "E". This is probably a consequence of a confusion between two Markdown-derived syntaxes.
- The user private keys (EdDSA signing key e, Curve25519 DH decryption key d, and symmetric secret key c) are documented as derived from the random seed s using HMAC/SHA-512 (s is the HMAC key, the data is a conventional fixed string that identifies the key type). However, the implementation uses HMAC/SHA-256 (function `DeriveFromSecret()` in `client/go/libkb/naclwrap.go`).
- The "reason" strings (key type identifiers) differ between documentation and implementation:

Key type	In documentation	In implementation
Signature key	`Keybase-Derived-User-NaCl-EdDSA-1`	`Derived-User-NaCl-EdDSA-1`
Encryption key	`Keybase-Dervived-User-NaCl-DH-1`	`Derived-User-NaCl-DH-1`
Symmetric key	`Keybase-Derived-User-NaCl-SecretBox-1`	`Derived-User-NaCl-SecretBox-1`

- The documentation states that the value B is encrypted with the public key of the new device when a new device is added. B is not defined in the documentation; in the implementation, the per-user seed (denoted s) is encrypted.

Sigchain and Merkle Tree Verification

Document URL: https://keybase.io/docs/server_security/merkle_root_in_bitcoin_blockchain

- Version 2 signatures are not properly handled by this example implementation. If the last chain link has a version 2 signature, the last code block incorrectly computes the payload hash and verification apparently fails.

Document URL: https://keybase.io/docs/teams/sigchain_v2

- The documented msgpack encoding for outer links in V2 links omits the `ignore_if_unsupported` boolean field, which is present in the actual client code.

Document URL: https://keybase.io/docs/api/1.0/call/sig/post

- The document states that signature packets always have `body.version` set to `1`. Although not checked in the source code, this is unexpected if the client is uploading a V2 signature.

Appendix C: Signature ID Blacklist Bypass Reproduction Steps

To confirm finding NCC-KB2018-001 on page 22, NCC Group simulated a malicious server modifying the contents of a sigchain. The following Keybase CLI command was run twice to demonstrate the issue:

```
HTTPS_PROXY=https://127.0.0.1:8888 ~/gopath/bin/keybase --standalone --debug --db=/home/ncc/tmp id max
➜  2>log
```

The various tokens have the following meaning:

- `HTTPS_PROXY` is used to redirect API calls to a local HTTPS proxy server. NCC Group used mitmproxy,[36] but Burp Suite[37] or similar would also work. To prevent certificate errors, the CA certificate of the proxy tool was added to the local Keybase configuration with `keybase config set proxy_ca_certs "/home/ncc/.mitmproxy/mitmproxy-ca-cert.pem"`. This enables the modification of API responses, simulating a compromised server.
- `~/gopath/bin/keybase` is the version of the Keybase CLI being tested.
- `--standalone` instructs the client to run as a standalone process. This makes API calls use the `HTTPS_PROXY` environment variable.
- `--debug` prints extra information about the verification process, giving insight into individual verification steps.
- `--db=/home/ncc/tmp` sets a custom cache directory that is wiped clean between executions. This is so the Keybase CLI starts with a fresh cache and performs all verification steps every time.
- `id max` instructs the client to verify the sigchain of user `max`.
- `2>log` redirects command output to a file for later comparison.

The command is run once, allowing API responses to pass through the proxy unmodified. The log is saved, the cache directory is cleared, and the command is run a second time. During the second run, the response to the call to `get.json` replaces signature ID `4a60aa2068fa7b1c99957731c72ff32df941155fa2bef6a33ffd7245127c4f300f` with `b175aaafbab6faf5740334039bb547a626c3b47b3ef4e55032b6aeaf6ce690520f`. The former signature ID is a revocation of the “tired purchase” backup key[38] and the latter is one of the hardcoded invalid chain links.

The output logs of the two executions are compared. Observe that the non-debug statements are identical, indicating the client does not indicate anything out of place and verification succeeded in the second case with the modified sigchain. Next, observe that the log of the unmodified run has the following sequence:

```
[DEBU keybase keyfamily.go:1056] 286 + UpdateDevice
[DEBU keybase keyfamily.go:1061] 287 | Device ID=4af8f8d8a6244f2f7c204af72a7adf18; KID=
[DEBU keybase keyfamily.go:1065] 288 | merge with existing
[DEBU keybase sig_chain.go:690] 289 - UpdateDevice -> ok
```

This sequence corresponds to the action of revoking the “tired purchase” key from the key family. Observe that the second, modified sequence does not have the same sequence. Instead, the modified run has the following log line, which indicates the chain link was skipped.

```
[DEBU keybase sig_chain.go:592] 286 Ignoring bad chain link with sig ID b175aaafbab6faf5740334039bb547
➜  a626c3b47b3ef4e55032b6aeaf6ce690520f: Link 98 of ajar's sigchain allowed a PGP update with a broken
➜  PGP key [tags:ID2=cb3WSSkEKI2J,ENG=BZYXkOxGD4FF,LU=HQO7CkKyGq68]
```

Since sigchain verification succeeded without revoking the backup key, the revoked backup key could still be used in this scenario to make further changes to the user’s sigchain.

[36] https://mitmproxy.org/
[37] https://portswigger.net/burp
[38] https://keybase.io/max/sigchain#4a60aa2068fa7b1c99957731c72ff32df941155fa2bef6a33ffd7245127c4f300f

To provide a higher confidence in the exploitability of finding NCC-KB2018-004 on page 28, NCC Group has written two scripts to help validate the issue. The first is a Python script that can be used to create an arbitrary JSON-msgpack polyglot, which it saves to `payload.bin`. The second is a Go program that loads `payload.bin` and uses the same decoding steps as the Keybase client to show that both decoders succeed when attempting to decode the binary blob. The following output is expected:

```
$ python payload.py && go run polyglot.go
Succeeded V1 decoding:
        Version: 1
Succeeded V2 decoding:
        Version: 2
```

This polyglot works because of how each decoder ignores extra bytes. When unpacking a msgpack blob into a structure, the msgpack decoder seemingly ignores extra fields. When unpacking the JSON blob, the JSON decoder ignores non-JSON bytes at the start or end of the blob. To create the polyglot, the JSON document containing arbitrary payload is created and converted to a sequence of bytes. This is then appended to the `OuterLinkV2` msgpack structure as a byte array. The `prev` and `curr` hashes must not contain any bytes that are interpreted as JSON characters, but a malicious client can find inputs that satisfy this constraint by brute forcing both hashes.

The first Python script is as follows:

```python
from base64 import b64encode
from os import urandom
from binascii import hexlify, unhexlify

import msgpack
import json

RANDOM_TEST = False

def js_payload():
    d = {
        "body": {"version": 1}
    }

    ds = json.dumps(d)
    return ds

def embed_in_msgpack (js):
    version = 2
    seqno = 47

    if RANDOM_TEST:
        # Sometimes the randomly chosen hash values contain characters that
        # interfere with JSON decoding, but often they don't. Enable
        # RANDOM_TEST to see how frequently a polyglot can be created in this way
        prev = urandom(32)
        curr = urandom(32)
    else:
        prev = unhexlify("e6e34a08990231e82cac11d6096c5dfd8d299a0e31983212950cf91a5479882e")
        curr = unhexlify("3a9285aa952eadf4d4943abef8e9ce3610a4f2f3b0417500b5edb1c28889ef64")

    type_ = 3
    seqno_type_ = 1
    ignore = False

    mpayload = [
        version,
        seqno,
        prev,
        curr,
        type_,
        seqno_type_,
        ignore,
        js
    ]

    payload = msgpack.packb(
        mpayload,
        use_bin_type=True
    )

    return payload


with open("payload.bin", "wb") as f:
    js = js_payload()
    payload = embed_in_msgpack(js)
    f.write(payload)
```

The Golang program is here:

```
package main
import (
        "fmt"
        "io/ioutil"
        "github.com/keybase/go-codec/codec"
        keybase1 "github.com/keybase/client/go/protocol/keybase1"

        "github.com/buger/jsonparser"
)

type LinkID []byte
type SigIgnoreIfUnsupported bool
type SigchainV2Type int

// OuterLinkV2 is the second version of Keybase sigchain signatures.
type OuterLinkV2 struct {
        _struct  bool           `codec:",toarray"`
        Version  int            `codec:"version"`
        Seqno    keybase1.Seqno `codec:"seqno"`
        Prev     LinkID         `codec:"prev"`
        Curr     LinkID         `codec:"curr"`
        LinkType SigchainV2Type `codec:"type"`
        // -- Links exist in the wild that are missing fields below this line.
        SeqType keybase1.SeqType `codec:"seqtype"`
        // -- Links exist in the wild that are missing fields below this line too.
        // Whether the link can be ignored by clients that do not support its link type.
        // This does _not_ mean the link can be ignored if the client supports the link type.
        // When it comes to stubbing, if the link is unsupported and this bit is set then
        // - it can be stubbed for non-admins
        // - it cannot be stubbed for admins
        IgnoreIfUnsupported SigIgnoreIfUnsupported `codec:"ignore_if_unsupported"`
}


func codecHandle() *codec.MsgpackHandle {
```

```go
        var mh codec.MsgpackHandle
        mh.WriteExt = true
        return &mh
}

func MsgpackDecode(dst interface{}, src []byte) (err error) {
        ch := codecHandle()
        return codec.NewDecoderBytes(src, ch).Decode(dst)
}

func unpackPayloadJSON(payload []byte) {
        var version int64
        var err error

        if version, err = jsonparser.GetInt(payload, "body", "version"); err != nil {
                fmt.Println("Failed V1 JSON decoding")
                return
        }

        fmt.Println("Succeeded V1 decoding:")
        fmt.Println("\tVersion:", version)
}


func decode_as_v2 (bs []byte) {
        var ol OuterLinkV2
        var err error

        if err = MsgpackDecode(&ol, bs); err != nil {
                fmt.Println("Failed V2 msgpack decoding")
                return
        }
        fmt.Println("Succeeded V2 decoding:")
        fmt.Println("\tVersion:", ol.Version)
}

func decode_as_v1 (bs []byte) {
        unpackPayloadJSON(bs)
}

func main() {
        payload, err := ioutil.ReadFile("payload.bin")
        if err != nil {
                fmt.Println("Failed to read from file")
                return
        }

        decode_as_v1(payload)
        decode_as_v2(payload)
}
```