Keybase **Book**

Go to keybase.io

Server

We've been working on Keybase.io for a little over half a year now, and we would like it succeed, but we're a little bit nervous. The more successful we are, the more valuable target we become.

Here are the attacks we are most concerned about:

1. Server DDOS'ed
2. Server compromised; attacker corrupts server-side code and keys to send bad data to clients
3. Server compromised; attacker distributes corrupted client-side code

We've taken some steps to protect the service from these attacks, and we wanted to describe them so you know what to look for.

What Keybase is Really Doing

Before we can describe how we protect keybase, we have to describe what it's actually doing, and what warrants protection. The central function of Keybase is to store, in a standardized format, public signatures for our users. The important signatures are of the form:

1. **Identity proofs**: "I am Joe on Keybase and MrJoe on Twitter"
2. **Follower statements**: "I am Joe on Keybase and I just looked at Chris's identity"
3. **Key ownership**: "I am Joe on Keybase and here's my public key"
4. **Revocations**: "I take back what I said earlier"

For instance, when Joe wants to establish a connection to an identity on Twitter, he would sign a statement of the first form, and then post that statement both on Twitter and Keybase. Outside observers can then reassure themselves that the accounts Joe on Keybase and MrJoe on Twitter are controlled by the same person. This person is usually the intended keyholder, but of course could be an attacker who broke into **both** accounts.

When an honest Joe signs such a proof, he also signs the hash of his previous signature. Thus, outside observers who want to verify all of Joe's signatures need only verify the last in the chain; the others follow. For example, I last signed a statement that I follow Keybase user al3x. I signed a JSON blob that contains relevant information about me and Alex, and also the key-value pair `"prev":"d0bd03..."`, where `d0bd03...` is the SHA-256 hash of the previous JSON blob I signed.

For a given user, the sum total of their signatures captures the state they wish to remember and to advertise to the world. For instance, my current profile shows that I am maxtaco on Twitter, that I was TacoPlusPlus on GitHub, but now I'm maxtaco there, too, and that I believe the Chris who is malgorithms on Twitter and malgorithms on GitHub is the "correct" Chris Coyne. Five signatures (one of which is a revocation) comprise this state; and an honest Keybase server should always show everyone these five signatures, so we can faithfully reconstruct my state in our clients.

Attacks 1 and 2: DDOS and Corrupted Data

We mentioned three attacks on this system. Consider the first two, which aim to prevent honest clients from retrieving signature data for honest users. A blunt attacker might DDoS Keybase's servers, preventing anyone from accessing Keybase's data. A more sophisticated attacker might root keybase's server, compromise its signing keys, and start sending back corrupted data to honest clients.

Two mechanisms, enforced by clients and third-party observers, defend against both attacks:

1. All user signature chains must grow monotonically, and can never be "rolled back"
2. Whenever a user posts an addition to a signature chain, the site must sign and advertise a change in global site state, and these updates are totally ordered.

Untrusted Mirrors

The first implication of these requirements is that untrusted third parties can mirror the site state, and clients can access data from either the Keybase server or the mirrors. By requirement (2), the server must publish and sign all site updates. A client doesn't care where these updates come from, as long as the signature verifies, and the site state jibes with the signature.

(We're not aware of third-party mirrors *yet*, and our reference client would need some modifications to handle a read-only server. However, we encourage all to scrape our APIs in preparation.)

Be Honest or Get Caught

The second implication of these requirements is that a compromised server has a choice of acting like an honest server, or making "mistakes" that honest users can detect. An attacker who gains control of the server can:

1. Selectively rollback a user's signature chain and/or suppress updates
2. Fake a "key update", and append signatures at the end of a user's chain
3. Show different versions of the site state to different users

Since version v0.3.0, the Keybase command-line client defends clients from these server attacks. Take the example of what happens when I "follow" Alex. My client downloads both of our signature chains from the server, and runs them through cryptographic verification, checking that our hash chains are well-formed and signed. It furthermore checks new data against cached data and complains if the server has "rolled back" either chain. My client prevents a compromised server from changing Alex's key the same way it prevents Eve from impersonating Alex: it checks for corroboration of Alex's identity and key proofs on other services (like Twitter, GitHub and DNS).

To prevent the server from "forking" my view of the site data from Alex's, my client checks that all signature chains are accurately captured in the site's global Merkle Tree data structure. It downloads the root of this tree from the server, and verifies it against the site's public key. If the check passes, it fetches the signed root block. My UID is `dbb165...`, so my client follows the `db...` path down the tree, which is block `68b5d3...`. Now, my leaf is visible, showing my signature chain finishing off at link 42, with hash `d0bd03...`, which matches the data it fetched earlier. My client does the same for Alex's chain. After all checks succeed, my client signs my chain, Alex's chain and also Merkle root at the time of the signature; it posts this signature as a follower statement.

A very sophisticated attacker could show my client and Alex's client different signed Merkle roots, but must maintain these forks permanently and can never merge. Users "comparing notes" out-of-band immediately expose server duplicity.

Keybase Client Integrity

Thus, the keybase clients in the wild play a crucial role in keeping the Keybase server honest. They check the integrity of user signature chains, and can find evidence of malicious rollback. They alert Alice when her following of Bob breaks, if either Bob or the server was compromised. They check the site's published Merkle tree root for consistency against known signature chains. And they sign proofs when all these checks complete, setting up known safe checkpoints to hold the server accountable to in the future.

So everything depends on the integrity of the Keybase clients, that they are functioning properly and aren't compromised. We offer several safeguards to protect client integrity. First, we keep an Open API and state that our open-source client is simply a *reference client*, and that developers are free to make new clients in different languages if they think we've done a bad job. Second, we sign all updates to the Keybase reference client, and provide an update mechanism to download new clients without trusting HTTPS, only the integrity of our key. We keep that private key offline, so that it wouldn't be compromised in the case of a server compromise.

We fully understand that users of the Keybase Web client don't get these guarantees. But our hope is that enough users will use the Keybase command-line client to keep the Web users safe, by catching server misbehavior in the case of a compromise.

Next Steps

The purpose of this article was to explain the security mechanisms the keybase system currently has in place. Going forward, it would be great if third parties were interested in hosting untrusted mirrors. These mirrors could eventually become auditors, too, allowing Alice and Bob to compare notes and convince themselves they're seeing a consistent view of the site's state.

And...an update! We're now publishing the merkle root into the bitcoin block chain.

Thanks for reading, and happy keybasing!

Keybase **Book**

Go to keybase.io

Meet your sigchain (and everyone else’s)

Every Keybase account has a public signature chain (called a *sigchain*), which is an ordered list of statements about how the account has changed over time. When you follow someone, add a key, or connect a website, your client signs a new statement (called a *link*) and publishes it to your sigchain.

As JSON (some fields removed), a sigchain looks like this:

```
[
	{
		"body": {
			"device": { "name": "squares" },
			"key": { "kid": "01208…" },
			"type": "eldest"
		},
		"prev": null,
		"seqno": 1
	},
	{
		"body": {
			"device": { "name": "squares" },
			"key": { "kid": "01208…" },
			"type": "web_service_binding",
			"service": { "name": "github", "username": "k
		},
		"prev": "038cd…",
		"seqno": 2
	},
	{
		"body": {
			"device": { "name": "rectangles" },
			"key": { "kid": "01208…" },
			"type": "sibkey",
			"sibkey": { "kid": "01204…", "reverse_sig": "
		},
                "prev": "192fe…",
                "seqno": 3
        },
        {
                "body": {
                        "device": { "name": "squares" },
                        "key": { "kid": "01208…" },
                        "type": "track",
                        "track": {
                                "basics": { "username": "cecileb" },
                                "key": { "kid": "01014…" },
                                "remote_proofs": [
                                        {
                                                "ctime": 1437414090,
                                                "remote_key_proof": {
                                                        "check_data_j
                                                                "name
                                                                "user
                                                        },
                                                },
                                        },
                                ]
                        },
                },
                "prev": "9fcc8…",
                "seqno": 3,
        }
]
```

This sigchain is from a user who…

1. Signed up for Keybase from a device called "squares" which generated a NaCl device key
2. Proved their GitHub account
3. Used squares to add another device called "rectangles" with its own key
4. Used rectangles to follow cecileb

You can try browsing a real sigchain online or through the API. Since sigchains are **public**, you can do this for any user on Keybase!

Every sigchain link is signed by one of the user's keys and includes a sequence number and the hash of the previous link. Because of this, the server can't create links on its own or omit links without invalidating the whole sigchain. We use a public Merkle tree to make it difficult for us to roll back a sigchain to an earlier state without being noticed.

Sibkeys

A Keybase account can have any number of sibling keys (called *sibkeys*) which can all sign links. This is different from PGP, which has a “master key” that you’re expected to keep tucked away in a fireproof safe — because if you misplace a device that has a copy of it, your only option is to revoke the whole key and start from scratch. We discuss this problem in a blog post.

You add and remove sibkeys by adding links to your sigchain. Since every link is checked against the state of the account *at that point in the sigchain*, old links remain valid even if their signing keys are revoked later. Revoking a key doesn’t affect your identity proofs, other keys, or followers.

Playback

To find the current state of an account (e.g. when you run `keybase id max`), the client starts out assuming that the key specified for the account in the Merkle tree is a sibkey, then *plays back* the sigchain link by link, keeping track of valid sibkeys and the effects of other links.

An implementation detail: since accounts can be reset, it actually starts playback at the most recent link whose `eldest_kid` matches the one in the Merkle tree.

Link structure

A complete version of the first link from the sigchain above looks like this:

```
{
        "body": {
                "device": {
                        "id": "ff07c…",
                        "kid": "01208…",
                        "name": "squares",
                        "status": 1,
                        "type": "desktop"
                },
                "key": {
                        "host": "keybase.io",
                        "kid": "01208…",
                        "uid": "e560f…",
                        "username": "sidney"
                },
                "type": "eldest",
                "version": 1
        },
        "client": {
                "name": "keybase.io go client",
                "version": "1.0.0"
        },
        "ctime": 1443241228,
        "expire_in": 504576000,
        "merkle_root": {
                "ctime": 1443217312,
                "hash": "06de9…",
                "seqno": 292102
        },
        "prev": null,
        "seqno": 1,
        "tag": "signature"
}
```

Some properties are common to every type of link. Here’s an overview:

- body – Information specific to the type of link, plus some common properties:
    - `type` – The type of the link
    - `device` – Optional details about the device that made the link
    - `key` – Information about the key that will sign the link. Contains these properties:

  - `host` – Currently always "keybase.io"
  - `eldest_kid` – The KID of the eldest key in this subchain. If missing, then eldest key is assumed to be the signing key (helps to identify account resets)
  - `kid` – The key's KID
  - `key_id`: For a PGP key, the last eight bytes of its fingerprint (legacy)
  - `fingerprint`: For a PGP key, its full fingerprint
  - `uid`: The user ID of the sigchain's owner
  - `username`: The username of the sigchain's owner

When a PGP key is being introduced or updated, there can also be a `full_hash` property which is a SHA-256 hash of an armored copy of the public key. This pins the key to a specific version.

- `client` – Optional version information about the client that made the link
- `ctime` – When the link was created, as a Unix timestamp
- `expire_in` – How long the statement made by the link should be considered valid, in seconds, or `0` if it doesn't expire
- `merkle_root` – The creation time, hash, and sequence number of the Merkle tree root at the time the link was created
- `prev` – The hash of the previous sigchain link when packed as canonical JSON, or `null` if this is the first link
- `seqno` – Specifies that this is the *n*th link in the user's sigchain
- `tag` – Currently always "signature". There may be other tag types in the future.

Properties have been added and deprecated over time, so there's some duplication and not all links in the wild have them all.

Link types

Each section below starts with an example body (and leaves out `key`, `device`, and `version`, which were described above).

`eldest`

```
{
        "type": "eldest"
}
```

Appears at the beginning of a sigchain or after an account reset (may not have been inserted by legacy clients). The link's signing key becomes the account's first sibkey.

sibkey

```
{
        "type": "sibkey",
        "sibkey": { "kid": "01204…", "reverse_sig": "g6Rib…" }
}
```

Add a new sibkey to the account. `reverse_sig` is a signature of the link by the new sibkey itself, made with the `reverse_sig` field set to `null`, and makes sure that a user can't claim another user's key as their own.

subkey

```
{
        "type": "subkey",
        "subkey": { "kid": "01216…", "parent_kid": "01204…" }
}
```

Add a new encryption-only *subkey* to the account. We plan to use these in the future.

pgp_update

```
{
        "type": "pgp_update",
        "pgp_update": {
                "kid": "01012ba0d60aa99320643f47eb787dc637821bc77cc89
                "key_id": "0DAA1A4AB1D88291",
                "fingerprint": "5e685e60eb8733654dcb00570daa1a4ab1d88
                "full_hash": "e02a1871c01285608c5bac3fb00be419b982c3c
        }
}
```

Update a PGP key to a new version (which may have new subkeys, revoked subkeys, new user IDs…). The `pgp_update` section contains the same properties a `key` section would. `full_hash` is expected to have changed, the other properties should be unchanged.

revoke

```
{
        "type": "revoke",
        "revoke": {
                "kids": [ "01201…", "01215…" ],
                "sig_ids": [ "038cd…", "f927c…" ]
        }
}
```

Remove the keys in `kids` from your account. Any previous links they've signed are still valid, but they can no longer sign new links and other users should no longer encrypt for them after seeing the `revoke` link. Also reverse the effects of the links in `sig_ids` — this can be used to remove, for instance, a `web_service_binding`.

web_service_binding

```
{
        "type": "web_service_binding",
        "service": { "name": "github", "username": "keybase" }
}
```

Claim, "I am `username` on the website `name`". The client will look for a copy of the link and signature on the website, in an appropriate place. The server searches for the proof when the link is first posted, and caches its permalink (e.g. the tweet, on Twitter, the Gist, on GitHub) so that the client doesn't have to rediscover it each time.

The `service` section can also look like this, which claims that you control the given domain name (the client looks for the proof in DNS):

```
{ "domain": "keybase.io", "protocol": "dns" }`
```

...or like this, which claims that you control the given website (the client looks for the proof at a known path):

```
{ "hostname": "keybase.io", "protocol": "http:" }`
```

track (to 'follow' someone)

We call this "follow" around the interface now, but our old word is "track"...so that's what you'll see in your sig chain:

```
{
        "type": "track",
        "track": {
                "id": "673a7…",
                "basics": {
                        "id_version": 30,
                        "last_id_change": 1440211236,
                        "username": "cecileb"
                },
                "key": {
                        "kid": "01018…",
                        "key_fingerprint": "6f989…"
                },
                "pgp_keys": [
                        {
                                "kid": "01018…",
                                "key_fingerprint": "6f989…"
                        }
                ],
                "remote_proofs": [
                        {
                                "ctime": 1437414090,
                                "curr": "ee483…",
                                "etime": 1595094090,
                                "remote_key_proof": {
                                        "check_data_json": {
                                                "name": "twitter",
                                                "username": "cecilebo
                                        },
                                        "proof_type": 2,
                                        "state": 1
                                },
                                "seqno": 1,
                                "sig_id": "02ad8…",
                                "sig_type": 2
                        }
                ]
        }
}
```

Make a snapshot of another user's identity that your other devices trust. The `track` section contains these properties:

- `id` – Their user ID
- `basics` – Contains their username and identity generation. The server bumps the identity generation whenever the state of any of their proofs changes, as a hint to the client that it should recheck them all and possibly alert the user to the change.
- `key` – Contains the KID and fingerprint (if applicable) of their eldest key.
- `pgp_keys` – An array of the KID and fingerprint of every one of their PGP keys
- `remote_proofs` – An array of objects which represent their proofs. Many properties are copied directly from the relevant links in the followee's sigchain, but there are some non-obvious ones:
    - `curr` – The hash of the link which contains the proof
    - `sig_type` – An integer representation of the proof's link type, currently always 2 (`web_service_binding`)
    - `remote_key_proof`
        - `check_data_json` – The `service` section of the identity proof link
        - `proof_type` – An integer representation of the account being proven (Twitter, GitHub, etc.)
        - `state` – An integer representation of whether the client could successfully verify the proof when making the tracking statement.

A repeat `follow` link for the same user replaces the previous one (the user may have re-followed due to a change in proofs).

`untrack` (to "unfollow" someone)

```
{
        "type": "untrack",
        "untrack": {
                "basics": { "username": "maria" },
                "id": "47968…"
        }
}
```

Stop following a user. Your other devices will resume checking their identity proofs and presenting them to you whenever you interact with them. `id` is the user's UID.

`cryptocurrency`

```
{
        "type": "cryptocurrency",
        "cryptocurrency": { "address": "1BYzr…", "type": "bitcoin" }
}
```

Advertise a cryptocurrency address. Currently Bitcoin, Zcash and Zcash sapling addresses are supported.

`per_user_key`

```
{
        "per_user_key": {
                "encryption_kid": "0121ef031c4b97e9e7febbfcce64952acb
                "generation": 15,
                "reverse_sig": "hKRib2R5hqhkZXRhY2hlZ...",
                "signing_kid": "0120eb42e0f5db28909adae170de9f5fc2401
```

```
    },
    "type": "per_user_key",
  }
```

Add or rotate a Per-user signing and encryption key. `reverse_sig` is the signature over the sigchain link with new per-user signing key itself. The `generation` number starts at one and increments whenever the per-user keys are rotated, typically after a device revocation.

footnote 1: PGP key servers and lying by omission

When someone changes a PGP key — to update its expiration date or add a signature, for example — they're expected to broadcast the change to a key server. That key server is responsible for forwarding the change to other key servers, and so on. Eventually, someone else can ask any other key server if there have been updates to the key, and receive them.

Notably, nothing stops someone from making a change to their PGP key on one computer, a different change on another computer, and sending each change to a different key server. The key servers are expected to share updates and offer their own combined versions of the key for download.

The design of PGP keys stops an attacker from creating fake updates, but a dishonest key server can still choose to ignore updates that revoke keys, revoke signatures, and add expiration dates, but publish updates that add new keys, add new signatures, and take away expiration dates.

Keybase sigchains aim to avoid this.

Understanding following (previously called "tracking")

We get some big questions about Keybase following:

- When should I follow?

- What does it get me?
- Is it a "web of trust?"

Hopefully this page can clarify and answer your q's.

```
<h3>But first, the goal of Keybase</h3>

<p>
  Keybase aims to provide public keys that can be trusted without a
  you should be able to get it, and know it's the right one, withou
</p>

<p>This is a daunting proposition: servers can be hacked or coerced
</p>



<p>
  When the Keybase server replies <b>"this is twitter user @maria29
</p>
```

Keybase **Book**

Go to keybase.io

Therefore, any cryptographic action on Maria follows 3 basic steps:

```
<ol>
  <li>The server provides maria's info</li>
  <li>Your client verifies her identity proofs on its own</li>
  <li>You perform a human review of her usernames</li>
</ol>
```

```
<p>Let's go over these three steps.</p>

  </div>



    <h2>Step 1: the request</h2>
    <p>
      When you wish to encrypt a message for your friend Maria, you mig
    </p>
    <hcode>bash
    keybase encrypt maria -m "grab a beer tonight?"
    </hcode>
    <p>
      So, first, your client asks the Keybase server who this mysteriou
    </p>
    <p>
      Keybase, the <i>server</i>, provides a response that explains its
      Technically speaking, it's a JSON object and there's a little mor
    </p>
    <hcode>json
    {
      "keybase_username": "maria",
      "public_key":       "---- BEGIN PGP PUBLIC KEY...",
      "twitter_username": "@maria2929",
      "twitter_proof":    "https://twitter.com/maria2929/2423423423"
    }</hcode>
    <p>
      Keybase has done its own server-side verification of maria, and i
    </p>
```

Step 2: the computer review

```
    <p>
      The keybase client does not trust the Keybase server. The server
      2, the client checks on its own.
    </p>
    <p>
      Fortunately, the server included a link to maria's tweet. The Key
    </p>

  </div>
  <div class="#{rcol}">
    <img src="/images/tracking/maria_twitter.jpg" class="img-responsive
  </div>
```

Keybase **Book**

Go to keybase.io

To satisfy the client, the tweet must be special. It must link to a signed statement which claims to be from maria on Keybase.

In simplest terms, the Keybase client guarantees that "maria" has access to three things: (1) the Keybase account, (2) the twitter account, and (3) the private key referenced back in step 1.

All this happens really fast in the client with no inconvenience to you. And it happens for all of maria's identities: her twitter account, her personal website, her github account, etc.

```
</div>



  <h2>Step 3: the human review</h2>

  <p>
    Recall, in Step 2 your client proved "maria" has a number of iden
    verified all of them.  Now you can review the usernames it verifi
  </p>
```

```
✓ maria2929 on twitter: https://twitter.com/2131231232133333...
✓ pasc4l_programmer on github: https://gist.github.com/pasc4...
✓ admin of mariah20.com via HTTPS: https://mariah20/keybase.tx...

Is this the maria you wanted? [y/N]
```

```
  <p>
    If it is, the Keybase client encrypts and you're done.
  </p>

  <h2>Finally: following</h2>

  <p>
    Steps 2 and 3 were easy enough, but it would stink to keep repeat
    the human review.
    Ideally, once you're satisfied with maria, you can just do this f
  </p>

  <hcode>bash
  # this should work with no interactivity
  keybase encrypt maria -m "another beer?"
  </hcode>

  <p>
    But we have a problem: recall, you don't trust the Keybase server
```

Go to keybase.io

```
      So how can you get maria's info when you switch machines, without
    </p>

    <p>
      <b>"Following" (which we used to call "tracking") is taking a sig
    </p>

    <p>
      Using your own private key, you can sign a snapshot of her identi
    </p>

    <p>
      When you switch computers, the Keybase server can provide you wit
    </p>

    <p>
      Your client can continue to perform the computer review as often
    </p>

    <p>

    </p>
```

The advantages of public following

```
    <p>
      When Maria is followed by 100 people, and they've all signed iden
    </p>

    <p>
      If some of these statements are months old, but your own is only
    </p>

    <p>
      This is not a web of trust. You can prove maria's identity, even
    </p>

  </div>
  <div class="col-sm-4">
    <img src="https://keybase.io/images/tracking/social.jpg" class="img
  </div>
```

```
    <h3>Why follow now?</h3>

    <p>
      As hinted above, an older follower statement is superior to a new
      maria's friends would surely notice.
    </p>
    <p>
      By comparison, if you started following Maria right now, today co
    </p>

    <p>
      Therefore, an older follower statement is a better one.
    </p>

    <p>
      <b>A gentle conclusion:</b> if you find someone interesting on Ke
    </p>
```

We hope this doc helps. We'll revise it as questions/suggestions arrive in our github issue #100 (I don't understand tracking).

footnote 1: the PGP web of trust

In the web of trust model, you know you have Maria's key because you trust John, and John signed a statement saying that another key belongs to his friend "Carla", and then Carla in turn signed a statement saying that Maria is someone whose driver's license and key fingerprint she reviewed at a party. Your trust of Maria's key is a function of these such connections.

you → john → carla → maria
you → herkimer → carla → maria

The PGP web of trust has existed for over 20 years. However it is very difficult to use, it requires in-person verifications, and it's hard to know what trust level to assign transitively. (Herkimer reports that Carla was drunk; John can't remember, but he was drunk too, and who's Carla again???)