Radicle Link

Kim Altintop (kim@monadic.xyz)
The Radicle Team (dev@radicle.xyz)

Revision rev2-draft, official/experimental

Contents

1 Introduction

2 Identities

In order to collaborate on repositories within a consensus-free network, we must be able to refer to them using a stable identifier. Note that this identifier is

a **statement of intent**: a repository can be described as a collection of ever-moving leaves of a DAG whose root element is the empty object. Therefore, the *content* of a repository is not enough to describe it – while two views on the repository may share objects, they may diverge substantially otherwise. Both views may however state their intent to eventually converge to the same state.

While in principle a random identifier with sufficient entropy would suffice for the purpose, this would put the burden of deciding which repository views are *legit* (for some definition of legit) entirely on the user. Instead, our approach is to establish an ownership proof, tied to the network identity of a peer (`PeerId`), or a set of peers, such that repository views can be replicated according to the trust relationships between peers ("tracking").

Our model is based on The Update Framework (TUF)[1], conceived as a means of securely distributing software packages.

2.1 Data Model

The identity of a repository is established using a document of the form:

```
struct Doc<T, D> {
    replaces: Option<Revision>,
    payload: T,
    delegations: D,
}
```

where:

- `Revision` is a cryptographic hash of a document's contents, such that this document is content-addressable by this hash within the storage system.
- `replaces` refers to the previous revision of the document, or none if it is the first revision
- `payload` is an extensible, forwards- and backwards-compatible datatype containing application-defined metadata about the repository. The protocol interprets *some* of the properties, as described in Doc Payload.
- `delegations` contains the public keys of key owners who are authorised to issue and approve new revisions of the document. The delegation format depends on the type of identity being established, as detailed below.

The `Doc` MUST be serialised in canonical form, e.g. Canonical JSON.

The authenticity of the `Doc` is captured by the following type:

```
struct Identity<T, D> {
    id: ContentId,
    root: Revision,
    revision: Revision,
    doc: Doc<T, D>,
    signatures: HashMap<PublicKey, Signature>,
}
```

where:

- `id` is the content-addressable hash of the `Identity` itself
- `root` is the initial revision of the identity `Doc`
- `revision` is the current revision of the identity `Doc`
- `doc` is the `Doc` described by `revision`
- `signatures` contains signatures over the document history, indexed by the public keys used. A signature is made over the cryptographic hash of the concatenation of the `Revisions` chain, from most recent to the `root`.

An `Identity` describes the attestation of a `Doc`s validity. Attestation chains MUST be stored as a hash-linked history (each `Identity` refers to its parent) – we omit the parent `id` as a field here, as it is not used in the following sections.

The `root` of a verified `Identity` is the stable identifier of the repository.

2.1.1 Radicle URNs

Identities are addressable within the Radicle Network by their stable identifier, encoded as a URN. Radicle URNs are syntactically and functionally equivalent to URNs as per [2], although we have no intention of registering our namespace with the IANA. r-, q-, and f-components are not currently honoured by the protocol, and MAY be discarded by recipients.

The syntax of a Radicle URN is defined as follows:

```
"rad" ":" protocol ":" root [ "/" path ]
```

where:

```
protocol = "git"
root     = MULTIBASE(MULTIHASH(id))
path     = pct-encoded
id       = BYTES
```

The `id` is the `root` field of a verified `Identity`, as specified previously. The `MULTIBASE` and `MULTIHASH` encodings are specified in [3] and [4], respectively. The preferred alphabet for the multibase encoding is [5]. `pct-encoded` is defined in [6], and the equivalence rules as per [2] apply.

Within the "git" `protocol` context, the `path` component is interpreted as a `git` reference (ref), and MUST thus conform to the refname rules as described in git-check-ref-format, as if no arguments were given. The prefix "refs/" MAY be omitted, but not the refs category (i.e. "heads/master" is permitted, but "master" is not). Valid ref categories are:

```
* heads
* tags
* remotes
* rad
```

An invalid ref category is treated as non-existent.

Without a `path`, a URN is to be treated as if the default value "rad/id" was given.

Before resolving a `path` to a ref in local storage, the percent-encoding SHALL be removed.

2.1.1.1 Examples

```
rad:git:hwd1yredksthny1hht3bkhtkxakuzfnjxd8dyk364prfkjxe4xpxsww3try
rad:git:hwd1yredksthny1hht3bkhtkxakuzfnjxd8dyk364prfkjxe4xpxsww3try/refs/heads/next
rad:git:hwd1yredksthny1hht3bkhtkxakuzfnjxd8dyk364prfkjxe4xpxsww3try/heads/next
rad:git:hwd1yredksthny1hht3bkhtkxakuzfnjxd8dyk364prfkjxe4xpxsww3try/tags/v0.0.1
rad:git:hwd1yredksthny1hht3bkhtkxakuzfnjxd8dyk364prfkjxe4xpxsww3try/remotes/hwd1yreb5ugudg6x
rad:git:hwd1yreb5ugudg6xewxiwotj97iaj31phjdphdiej44x1acer3i45uqnwxw/rad/id
```

2.1.2 Doc Payload

The `Doc` payload MUST include one of the following structures (but not both) for interpretation by the protocol:

```
struct Person {
    /// A short name (nickname, handle), without any prefix such as the `@`
    /// character
    name: String,
}

struct Project {
    /// A short name
    name: String,

    /// A slightly longer description (should fit in a headline)
    description: Option<String>,

    /// The default branch. "master" is assumed for git repositories if
    /// unspecified.
    default_branch: Option<String>,
}
```

There are currently no restrictions on the length (in bytes) of the fields.

Applications MAY add additional payload data, but MUST do so in a way which unambiguously preserves the shape of the above definitions (see also: Serialisation).

2.2 Verification

Document revisions, as well as authenticity attestations form a hash-linked chain leading to the initial revision of the document. In order to verify the authenticity of a given identity attestation (`Identity`), the chain of updates must be contiguous and subject to the following verification rules.

We can distinguish four levels of validity:

1. **Untrusted**

   The identity document is well-formed, and points to a root object which is retrievable from local storage.

2. **Signed**

   The identity carrier passes 1., and is signed by at least one key specified in the delegations of the document.

3. **Quorum**

   The identity carrier passes 2., and is signed by a quorum of the keys specified in the delegations of the document (`Q > D/2`).

4. **Verified**

   The identity carrier passes 3., and:

   - The document does not refer to a previous revision, and no previous revision is known
   - **Or**, the set of signatures forms a quorum of the delegations of the **previous** revision.

   It is an error if:

   - No previous revision is given, but a parent in the identity attestation chain is found
   - A previous revision is given, but the identity attestation chain does not yield a parent
   - A previous revision is given, but it is not the same the parent attestation refers to
   - The current and parent attestations refer to different `roots`

The verification process can now be described recursively:

```
/* State transitions, definitions elided for brevity */
fn signed(i: Untrusted<Identity>) -> Result<Signed<Identity>, Error>;
fn quorum(i: Signed<Identity>) -> Result<Quorum<Identity>, Error>;
fn verified(i: Quorum<Identity>, parent: Option<Verified<Identity>>) -> Result<Verified<Iden

fn verify(
    head: Untrusted<Identity>,
    mut parents: impl Iterator<Item = Untrusted<Identity>>,
```

```
) -> Result<Verified<Identity>, Error> {
    let head = quorum(signed(head)?)?;
    let parent = match parents.next() {
        None => Ok(None),
        Some(parent) => verify(parent, parents),
    }?;

    verified(head, parent)
}
```

Implementation Notes:

- The recursive definition is given for brevity, implementations may prefer to walk the history in reverse order.
- As an optimisation, implementations SHOULD store the result of a successful verification in persistent storage, such that verification of updates can start from the last-good state instead of the history root. Observe, however, that two different `Identity` objects (as identified by their `id`) may attest the same `Doc` revision – the persistent state SHOULD therefore allow retrieving the set of attesting `Identity`s by `Doc` revision.
- The `quorum` predicate above SHOULD actually skip over this parent instead of aborting the procedure – it is possible that a later `Identity` reaches the quorum.

2.3 Delegations

Radicle Link distinguishes two types of identities: personal and project. The first describes an actor in the system, while the second describes a (software) project on which one or more actors collaborate. Apart from their payload types `T`, they differ in their delegations type `D`:

Personal identities can only delegate to anonymous keys, while project identities MAY attach a personal identity to a key delegation.

More formally:

```
type Person<T> = Identity<T, HashSet<PublicKey>>;

enum ProjectDelegation<U> {
    Key(PublicKey),
    Person(Person<U>),
}

type Project<T, U> = Identity<T, ProjectDelegation<U>>;
```

Per identity document, the `PublicKeys` delegated to MUST form a set. It is, for example, an error if a `PublicKey` appears in both the document and a `Person` delegations, or within two different `Person` delegations.

If a `Project` delegates to a `Person`, it MUST do so by including a specific revision of the `Person` document. The replication rules ensure that the respective `Person` histories are replicated as sibling histories [insert link].

If one of these sibling histories is found to not include the delegated-to revision in its ancestry path, it is said to have **forked**, and signatures made by one of the `Person`s keys after the point of inclusion in the `Project` no longer count towards the quorum rules. Note that this is only recoverable if a quorum of valid keys remains on the project, otherwise a Key Recovery procedure must be invoked on the `Project`.

If a sibling history is NOT **forked** (i.e. it includes the attested revision in its ancestry path), AND it is **verified**, its key delegations are considered authoritative for all `Project` attestations between the point the `Person` was delegated to, and the currently known-good head.

> **Implementation Note**: If `Person` keys are not reused (i.e. revoked and later re-introduced) since the attestation point (which MUST be verified), it is sufficient to check for key revocations using the latest known heads of the respective histories.

When calculating the **Quorum** or **Verifed** threshold, multiple signatures made by the set of keys of a `Person` SHALL be counted as only one vote towards the quorum. This prevents unilateral decisions made by a single `Person`. We consider this simple scheme sufficient for the purpose, but more sophisticated delegations may be supported in the future, such as customising the quorum threshold, or key roles.

Note that a key revocation event in a sibling `Person` history may render the project unusable if the remaining keys cannot form a quorum. Also note that the `Project` SHOULD renew the attestation from time to time.

The `delegations` of a `Project` are also referred to as the project's **maintainers**.

2.4 Key Recovery

> **TODO** Define how to delegate trust to a set of keys `K \ D` for the purpose of recovering from key revocations in case the quorum rules can no longer be met with the remaining keys. (i.e. "Social Recovery")

2.5 Effect on Replication

Peers MUST NOT replicate repositories whose identities they are unable to verify. If an update to the identity cannot be verified, they SHOULD keep the data they already have, but refuse to update it. Peers SHOULD keep the identity histories regardless of their verification status.

When **cloning** a repository, the attestation history according to the remote peer

is fetched before all other repository contents, and the verification procedure is run on it. If this does not yield a verified status, the clone is aborted. The resulting repository state MUST include the attestation histories of at least a quorum of the delegates as per the remote peer's view of the identity document. In `git`, the claim that this will be the case can be determined before fetching the repository contents by examining the advertised remote refs. If these preconditions are not met, the clone is aborted, and already fetched data is pruned.

When **fetching** (i.e. updating) a repository, a predetermined set of verified delegates is already known – their views of the attestation history SHALL be fetched first, along with any advertised histories not yet present in the local repository (as those might be owned by future delegates). The fetching end MUST now find a valid `Doc` revision near the tips of the available delegate histories, and pick the most recent one. If two valid identities are found of which neither is in the ancestry path of the other, the attestation chain is said to have **forked**. In lieu of a consensus system, it is undecidable which side of the fork to commit to, and so replication of the attested repository SHALL refuse any further updates.

> **TODO** Fork detection (i.e. pairwise compare if in same ancestry path, either select most recent or flag as forked)

2.6 Git Encoding

In the `git` implementation, a `Doc` corresponds to a `blob` object, stored as the single entry of a `tree` object, such that its name (acc. to the `tree`) is equal to the `blob` hash of the *initial* version of the `Doc`, serialised in canonical form. That is:

```
let name = git hash-object -t blob doc.canonical_form()
```

An `Identity` corresponds to a `commit` object.

We map the fields as follows:

```
/* Simplified git object model */
struct Commit {
    id: Oid,
    tree: Tree,
    message: String,
}

struct Tree {
    id: Oid,
    entries: Vec<TreeEntry>,
}

struct TreeEntry {
    id: Oid,
```

```
    name: String,
    object: BlobOrTree,
}

struct Blob {
    id: Oid,
    content: Vec<u8>,
}

/* Mapping (trivial type conversions elided) */
let commit = /* .. */;
let identity = Identity {
    id: commit.id,
    root: commit.tree.entries[0].name,
    revision: tree.id,
    doc: deserialize(first_blob(commit.tree).content),
    signatures: from_trailers(commit.message),
};
```

Where:

- `first_blob` finds the first `TreeEntry` which is of type `blob`.
- `deserialize` is implemented by a standard JSON parser. `Person` delegations from a `Project` are specified in the `Project`'s `Doc` as URNs, which are resolved by parsing a `blob` object of the same name as the URN's `id` field below the `tree` entry of type directory named `delegations`.
- `from_trailers` interprets the commit message as per git-interpret-trailers, and extracts the signatures from trailers with the token `x-rad-signature`.

  TODO Specify exact format of signature trailer value (BASE64(public key || signature))

The commit chain is stored in a branch at `refs/rad/id`.

TODO Insert repository layout spec as per RFC here

2.6.1 Serialisation

The `Doc` is serialised in Canonical JSON format. However, all ASCII plane control characters (`U+0000 - U+007F`) MUST be escaped according to [7]. Hexadecimal escape sequences MUST be in lower-case. This contradicts Canonical JSON, but permits their claim that "Canonical JSON is parsable with any full JSON parser".

In addition to the shape defined above, a field `version` MUST be included with a value of `0` (zero) as of this version of the specification.

`Revision` values are serialised as JSON strings, encoded as a [4] value wrapped in a [3] encoding using the [5] alphabet.

`PublicKey` values are serialised as JSON strings, which are obtained by concatenating the `0` byte (as a version identifier) with the Ed25519 scalar (the public key) encoded as per [8], and then wrapping in a [3] encoding using the [5] alphabet.

`Person` delegations are serialised as a JSON array of `PublicKey` values. Duplicate elements MUST be a deserialisation error.

`Project` delegations are serialised as a JSON array of either `PublicKey` or URN values without tagging. Duplicate elements MUST be a deserialisation error.

The `payload` is encoded as a JSON object using [9] URLs as the keys, and JSON objects of the `radicle-link-` or user-specified payload objects as the values. Implementations MUST validate that a valid `radicle-link` payload is present in the object, and SHOULD preserve user extensions typed as the JSON object model. Duplicate URL keys are a deserialisation error.

The URL keys serve as namespaces, as well as version identifiers. Versioning simplifies schema evolution. However, implementations MUST ensure that they can interpret both unknown (future) and outdated (past) versions. This specifically means that the URL keys for `radicle-link` payloads MUST be considered equal iff their prefix sans the version identifier is equal, and duplicates rejected accordingly. It is RECOMMENDED that extension namespaces are also constructed such that the version identifier is the suffix, and `radicle-link` implementations SHOULD provide prefix-queries on the extension namespaces in a payload object.

Additionally, empty optional fields SHOULD be included with a value of `null`, rather than omitting them from the output.

This specification does not devise a schema resolution mechanism based on the payload URLs, nor does it mandate schema validation, although applications are free to implement both.

Pending self-hosting, which will allow precise versioning by content-address, the URLs for `radicle-link` payloads are:

Person https://radicle.xyz/link/identities/person/v1

Project https://radicle.xyz/link/identities/project/v1

The `radicle.xyz` domain is reserved for application payloads defined by the Radicle Core Team.

2.6.1.1 Examples

A simple example if an initial `Doc`, embedding a Decentralized Identifier (DID) document [10] in a `Person` payload:

```
{
    "version": 0,
```

```
    "replaces": null,
    "payload": {
        "https://radicle.xyz/link/identities/person/v1": {
            "name": "cloudhead"
        },
        "https://www.w3.org/ns/did": {
            "@context": "https://www.w3.org/ns/did/v1",
            "id": "did:example:123456789abcdefghi",
            "authentication": [{
                "id": "did:example:123456789abcdefghi#keys-1",
                "type": "Ed25519VerificationKey2018",
                "controller": "did:example:123456789abcdefghi",
                "publicKeyBase58": "H3C2AVvLMv6gmMNam3uVAjZpfkcJCwDwnZn6z3wXmqPV"
            }],
            "service": [{
                "id":"did:example:123456789abcdefghi#vcs",
                "type": "VerifiableCredentialService",
                "serviceEndpoint": "https://example.com/vc/"
            }]
        }
    },
    "delegations": [
        "hyn4hnppkiu61kpx91o9n5jtj37brujcgj7yp8d1derwz4fbk3tqjw",
        "hyb8kud543qkfdxkge6ecj6zziuam6w6fqhujgebbfuufmpdxt5uok"
    ]
}
```

Note that the example is **not** in canonical form.

2.6.2 Workflow

Our construction of the `Identity` from a git commit allows for multiple `id`s to describe the same revision of the document (and thus be equally valid). This means that the respective delegates' histories may diverge in their *commit* histories, but still converge to an agreement on the validity of the attested document revision.

While this allows for arbitrarily complex workflows (and history topologies), we RECOMMEND to converge to equal or joined histories. This can be achieved by selecting a leader to commit the quorum (and the followers adopting this commit), or settling the vote out-of-band.

As an example, a simple leader-based workflow could proceed as follows:

- Peer `A` proposes a new revision on her view of the `rad/id` branch. The revision is signed by `A`'s public key.
- Peers `B` and `C` receive this proposal, and sign it off on their own `rad/id` branches by creating a new commit over the same `tree` as the propsal,

but including their own signatures (they may also include `A`'s signature for posterity, but that is not required as the verification procedure will skip those commits).
- Peer `A` is also designated as the leader, so after receiving the updates of `B` and `C`, she creates a new commit on her branch, merging both `B`'s and `C`'s "votes" (an "octopus merge"), and preserving the signature trailers of all three.
- `B` and `C` receive this "finalisation" commit, and simply merge it into their own branches. This is a fast-forward merge.

2.6.3 Implementation Notes

Implementations are encouraged to store verification results (including detected forks) persistently, both for efficiency reasons, and to preserve this state even if the corresponding repositories are removed from local storage (e.g. because they reached an irrecoverable state).

This can be implemented without the need for a secondary storage system by leveraging git-notes:

In the private section of the "monorepo" (i.e. outside any namespaces), store a note for every verified tree object, with the set of commit OIDs which contributed to this verified state encoded in the note message (e.g. `x-rad-verified-by: <OID>`). Determining whether a given revision has been verified before is an `O(1)` operation: only the notes object needs to be consulted. Verification can proceed incrementally by skipping commits pointing to pre-verified trees. Note that the notes objects can be loaded into memory upfront, and that the skipping only requires to read the commit headers from disk.

Persisting forks requires to annotate commit objects as well, and to update the tree notes to reflect the new state.

2.7 Security Considerations

- Colluding key owners can force the network into following a *fork* of an already published identity. While peers are *encouraged* to persist information about detected forks, they are not incentivised to keep that information forever. After a period of interrupted replication, the fork may thus prevail.
- As outlined in Delegations, `Person` and `Project` histories are not causally related. This allows for censorship attacks, unless this relationship is securely established by an external system.
- The security of identity updates rests on the probability of an attacker to gain control over a quorum of keys. That is, we can operate on the assumption that the individual keys can be in "online" storage, so as long as the breach of a minority of the key delegations can be mitigated by the remaining ones.

This poses a usability problem, however, especially for personal identities: users can hardly be expected to have access to `n > 2` devices within an acceptable time window in order to finalise an identity update. In practice, many will have access to a portable hardware token which provides reasonable protection of the keys it stores, and will want to finalise updates immediately by only two signatures: by the current workstation + by the hardware token.

A future amendment to this document SHALL specify the rules for marking a key delegation as being considered "secure", such that the quorum threshold can be computed differently if such a "secure" signature is present.

- Implementations SHOULD limit the download of untrusted identity histories to a reasonable size in bytes.

3 Replication

4 Network Protocol

5 Content Discovery

Collaboration

00 [1] J. Cappos et.al., "The Update Framework Specification." 2019. https://github.com/theupdateframework/specification/blob/0cddec0a60f95f06d2e23ebadbd876eeb62c1df3/tuf-spec.md

[2] P. Saint-Andre and D. J. C. Klensin, "Uniform Resource Names (URNs)." RFC 8141; RFC Editor, Apr-2017. https://rfc-editor.org/rfc/rfc8141.txt

[3] https://github.com/multiformats/multibase/blob/f2d3c43f9d30d7dca178dc3220c5bf50963a1e70/README.md. [Accessed: 29-Jul-2019]

[4] https://github.com/multiformats/multihash/blob/cde1aef8158d193d73012b7d730013f05c2f7063/README.md. [Accessed: 26-Jul-2019]

[5] http://philzimmermann.com/docs/human-oriented-base-32-encoding.txt. [Accessed: 18-Aug-2020]

[6] T. Berners-Lee, R. T. Fielding, and L. M. Masinter, "Uniform Resource Identifier (URI): Generic Syntax." RFC 3986; RFC Editor, Jan-2005. https://rfc-editor.org/rfc/rfc3986.txt

[7] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format." RFC 8259; RFC Editor, Dec-2017. https://rfc-editor.org/rfc/rfc8259.txt

[8] S. Josefsson and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm

(EdDSA)." RFC 8032; RFC Editor, Jan-2017. https://rfc-editor.org/rfc/rfc8032.txt

[9] https://url.spec.whatwg.org/commit-snapshots/1979119c2082b1e0f7f77aead4651700841a4abe/. [Accessed: 20-Aug-2020]

[10] https://www.w3.org/TR/did-core/. [Accessed: 21-Jul-2020]