`docker sign` UX

EXTERNAL DOC

Overview

Use Cases
- "I want to sign my docker images"
- "I want to collaborate with my teammates on docker images"
- "I need to administer who can sign an image"
- "I want to create a threshold of required signers for an image to be deployable"
- "I want to inspect who can sign images in my repo"
- "I want to inspect what images are signed and who signed them"
- "I want to remove a compromised key everywhere it's used to sign"
- "I want to remove a signer from a repository"
- "How can I see where an image's signing configuration differs from the signing policy I've configured in UCP?"
- "A contributor left my organization, I need to remove them everywhere they're a signer"

Commands
- Initialization
- Signing
- Revoking Trust
- Importing, Generating, and Exporting Keys
- Introspection and Repo Info
- Adding, Removing, and Updating Delegations

Configuring Trust in Docker
- "Behind the Scenes" Details

Example: Setting up Signing Policy
- With docker trust command:
- Without docker trust command:

Overview

Docker the organization has an opinionated use of Notary that unfortunately leaks a number of abstractions that make content trust confusing and difficult for users. It has been the intent for a long time to integrate a `docker sign` or similar subcommand. Doing so provides the perfect opportunity to hide those confusing abstractions.

This document investigates a number of common use cases for content trust and suggests a command syntax for a subcommand on the docker binary that meets those use cases in an easy to understand format.

Use Cases

“I want to sign my docker images”

This is the input side of the most basic operation, allowing a publisher to provide authenticity and integrity over an image. A user should be able to sign their own images in a straightforward manner.

“I want to verify the signatures on all images I download”

Verification is the other side of the most basic operation. It allows a consumer to verify the provenance (who) and integrity (what) of an image. The underlying framework also provides freshness (is the image up-to-date) and survivable compromise guarantees, increasing the level of trust a consumer can have in the content they verify.

“I want to collaborate with my teammates on docker images”

Very few people work in isolation and among our customers, it is a certainty that teams of people will be involved in producing an organization's images. Moreover, when collaborating, users shouldn’t have to compromise on the security of the *provenance* of their images - it must be possible to verify signatures from their collaborators. Thus, should be simple to add a user to a repository as a signer.

“I need to administer who can sign an image”

It should be simple to add and remove signing keys and users. This allows for dynamic and fine-grained access control of which users should be allowed to sign for which images.

“I want to create a threshold of required signers for an image to be deployable”

UCP added signing policies in the second half of 2016 but these are difficult to set up and use. It needs to be much simpler to manage who can sign an image in the context of UCP signing policies. It also needs to be simpler for a user to add their signature on an image that already exists (and may or may not have been signed by other people).

“I want to inspect who can sign images in my repo”

“I want to inspect what images are signed and who signed them”

“I want to remove a compromised key everywhere it’s used to sign”

“I want to remove a signer from a repository”

Once I remove a signer from a repository, what happens to the images they pushed and signed already?
- What happens if I build another image that uses an image from that repo as base?
- Same as above, but I already have the base image pulled
- Can I deploy a new service using that image?
- What happens to services that are using those images? They should continue running, right?
- What happens if a container using that image stops, and you need to restart it? Will you be able to?
- What happens if the node running a service with that image crashes, and the service is rescheduled to run on a node that doesn't have that image yet?

“How can I see where an image’s signing configuration differs from the signing policy I’ve configured in UCP?”

“I want to use only signed base images when building my application images”

“A contributor left my organization, I need to remove them everywhere they’re a signer”

Functionally this is similar to “I want to remove a compromised key everywhere it’s used to sign.” The main difference is that if a user’s key is compromised, they may have multiple keys and only one should be removed. In this instance however, all of a users keys should be removed.

We should carefully consider how a role may continue to live on even if nobody should be able to sign it any more. Maybe we use an ephemeral key to sign it so it’s still valid but nobody holds the private key to continue updating it.

Commands

Before going in to any specific commands, I propose that while we have consistently talked about `sign` being the subcommand, `trust` parses better and more consistently. Therefore I will use `trust` for the below commands.

```
🐳 $ docker trust

Usage:     docker trust COMMAND

Sign images to establish trust

Options:
      --help   Print usage

Commands:
  config         Configure trust settings
  info           Display detailed information about keys and
signatures
  init           Initialize trust for an image without pushing
  key            Operates on signing keys
  revoke              Remove trust for an image
  sign           Sign an image
  signer         Manage trusted signers for an image

Run 'docker trust COMMAND --help' for more information on a command.
```

Initialization

All use cases require initialization of a repo. However in some use cases we have typically recommended that the notary binary is used for initialization. The notary binary is implemented as a pure TUF tool, with no knowledge or logic for docker's opinionated use of Notary. The new `trust` subcommand can implement docker's opinionated logic, but more importantly, `docker push` should perform a more opinionated initialization (it does a less opinionated initialization today).

The new command to be added is:

```
docker trust init endophage/example
```

Whether a repository is initialized via `docker push` or `docker trust init` (or implicitly with `docker trust sign` or `docker trust signer` discussed below), the below set of roles and delegations will be created:

- Root: local key
- Targets: local key
    - targets/releases: local key
    - targets/endophage: local key and same as targets/releases. Note “endophage” is an example, this would be the username of the user performing the `trust init` or `push`. If no username is known, for example the upstream servers are running without authentication, the string “owner” will be used in place of a username.
- Snapshot: key held in Notary Signer
- Timestamp: key held in Notary Signer

The first time a root and delegation key are created, a local configuration file is updated with the IDs of these keys. They are subsequently reused for all other repos initialized on this host.

Corresponding Notary commands:

This command encapsulates this sequence of existing notary commands:

```
notary init docker.io/endophage/example
```

```
notary publish docker.io/endophage/example
```

```
notary delegation add docker.io/endophage/example targets/releases
key.pem --all-paths
```

```
notary delegation add docker.io/endophage/example targets/endophage
key.pem --all-paths
```

```
notary publish docker.io/endophage/example
```

Signing

All images will be signed using one rule on which roles the image will be signed in to: an image will be signed in to all targets/[^/]* delegations for which the key is available at the time of signing. The regex [^/]* is used to indicate a single path segment, i.e. only delegations 1 level deep will be inspected.

As is done today, the `docker push` command will sign an image into the delegation roles meeting the above rule.

Additionally we propose the following new command which will perform an optimized signing for an image that already exists remotely:

If the image does not exist remotely, this command will look for the image locally. If the image is found only locally, this command will be equivalent to a `docker push` with content trust enabled. If trust data for the repository does not yet exist, this command will also initialize the trust data

```
docker trust sign endophage/example:latest --meta
apparmor:./myapp.apparmor.profile
```

This command also enables metadata to be attached to the custom field for the tag in Notary. In the above example, we are attaching an apparmor profile, giving it the name “apparmor”. The profile itself will be uploaded to the registry, with its checksum being stored against the given name as part of the custom data on the “latest” tag record in Notary.

NOTE: this command will also implicitly initialize a repo if it is not initialized already

Corresponding Notary commands:

This command encapsulates this sequence of existing notary commands:

```
notary add docker.io/endophage/example latest
```

```
notary publish docker.io/endophage/example
```

Revoking Trust

“Unsigning” an image tag will either require repushing the trust data without the signature, or removing all trust data from the associated notary server.

Removing a single signature:

```
docker trust revoke endophage/example:latest
```

Removing all trust data for all tags in an image:

```
docker trust revoke endophage/example
```

This command should probably have an interactive prompt for `yes` allow for a `-f` force flag. It will delete the local cache and the remote data. Open que stion is whether we have a `--local-only` flag to only delete the local cache and not any remote data.

Corresponding Notary commands:

This command encapsulates this sequence of existing notary commands for removing a single signature:

```
notary remove docker.io/endophage/example latest
```

```
notary publish docker.io/endophage/example
```

Additionally, for full repo revoke:

```
notary delete docker.io/endophage/example --remote
```

Importing, Generating, and Exporting Keys

The following command will load a private key from a file and store it, encrypted, in the local key storage. It will use the PEM headers to infer the role, but it can be overridden with CLI flags

```
docker trust key load key.pem
```

The following command will generate a new *delegation* key. It cannot be used to generate any other key types. The key may optionally be given a name which will be used in various local outputs to give the user a hint as to which key is being used. If no name is provided, the key ID will be used in these situations.

```
docker trust key generate [name]
```

The following command will export private delegation keys to stdout. If the --root flag is included, it will export the root keys for the given repos rather than the delegation keys. If the --output flag is provided, the data will be written to the given file path rather than stdout. The keys for all image repositories given will be exported as multiple consecutive pem blocks.

```
docker trust key save <image name> ... --root --output key.pem
```

The following command will rotate the key for the given repositories and role. If the --to flag is provided, it must point at a public key of certificate encoded in pem format. This is the key that the given role will be rotated to. If the --to flag is not provided, a new key will be generated.

```
docker trust key rotate <role> <image name> ... --to pub.pem
```

Introspection and Repo Info

After signing, it's a natural next step that users will want to see what they signed, and with which keys:

```
docker trust info endophage/example
```

By default, this should show:

- All signed tags and which keys signed them (can be multiple delegations)
- All keys and delegations known to the repo

This command will attempt to check an online notary server (https://notary.docker.io by default), but will fallback to trusted cache data if it is not expired.

I think we should allow for filtering flags if a user knows they only want to see a subset of this information. I've[Riyaz] proposed the following flags but perhaps we can tie them closer to docker image subcommands:

```
docker trust info endophage/example --tags-only
```

```
docker trust info endophage/example --keys-only
```

To see "global" state about all keys and configuration in the cache, the user can use the command without any arguments

```
docker trust info
```

This should show:

- All keys kept in the local cache
- The configuration settings being used:
    - Trust server
    - if the client is online
    - Cache directory
    - Trust pinning

Adding, Removing, and Updating Delegations

Still WIP: I'm (Riyaz) thinking we do something like Proposal 2 from Evan's issue: https://github.com/docker/notary/issues/1159

The trust subcommand will abstract delegations as "signers". For a given repo, an individual signer is specified by a name, and a collection of keys that the signer will use to sign an image. The operations for a signer at a high level are:

- Add a signer and their keys to the repo
- Remove a signer and all their keys from the repo
- Remove a key from all signers in case of compromise
- Add an additional key to a signer
- List all defined signers

Add signer using keys already loaded into the local storage and from external files. This will automatically add the keys to *both* targets/david and targets/releases delegation roles under the hood.

```
docker trust signer add endophage/example david --key key1
```

NOTE: this command will also implicitly initialize a repo if it is not initialized already

Note: we allow for multiple `--key` specifications in case a user would like to have multiple keys. The command will try to lookup a key as a named reference in our local cache. If that fails, it will try to read as a filepath. Example here:

```
docker trust signer add endophage/example david --key key1
```

Add additional keys to an existing signer:

```
docker trust signer add-key endophage/example david --key key2
```

Completely remove signer:

```
docker trust signer remove endophage/example david
```

Remove a specific key from all signers, in case of compromise:

```
docker trust signer remove-key endophage/example key1
```

List signers for a repo: all roles and keys associated with them. If the delegation keys have specified names, also use those:

```
docker trust signer list endophage/example
```

Configuring Trust in Docker

This section with deal with the two main areas of configuration for trust, which previous were either not configurable, or could only take a single global value when that was contextually undesirable. Specifically, those are trust pinning, and the notary server URL.

First up, configuring trust pinning. Trust pinning can operate at its simplest level in TOFUS (Trust On First Use Securely, essentially TOFU over TLS) mode, or in a strict pre-configured mode. The command to enable/disable TOFUS would be:

```
docker trust config tofu enable
docker trust config tofu disable
```

To pin specific certificates or CAs for a GUN or GUN prefix, the commands would be:

```
docker trust config pin endophage/example cert1.pem cert2.pem
docker trust config pin endophage/example ca.pem
docker trust config pin endophage/* cert.pem
docker trust config pin endophage/* ca.pem
```

Certificates always take precedent over a CA and the given pem will be inspected to determine if it is a leaf cert or a CA cert. The most specific configuration will also always be used. So if as in the commands above, we have configured certificates for “endophage/example” and “endophage/*” we will always prefer “endophage/example”.
To clear the trust pinning configuration for a given name, the command is:

```
docker trust config pin endophage/example --clear
docker trust config pin endophage/* --clear
```

This will delete all trust pinning configuration related to the provided GUN or GUN prefix

Updates to a trust pinning configuration are treated as overwrites, so in the following example, after running the second command, only cert2.pem would be trusted for the endophage/example repo.

```
docker trust config pin endophage/example cert1.pem
docker trust config pin endophage/example cert2.pem
```

To configure the notary server location for a given image name, the command will be

```
docker trust config server endophage/* https://notary.docker.io
```

The scheme may be omitted, i.e. “notary.docker.io”, however notary requires that servers use TLS so this scheme will always be added if not present. Providing the “http” scheme will return an error.

To clear a server configuration for a given GUN prefix, the command is

```
docker trust config server endophage/* --clear
```

“Behind the Scenes” Details

Default settings:

- Default trust server: `https://notary.docker.io`
- Default trust cache: `~/.docker/trust`
- Default username/password credentials to be shared with Docker
- TBD: default passphrase for encrypting keys at rest?

Example: Setting up Signing Policy

Let’s compare this command to the current status quo with Notary and DCT. Example: setting up a repository for UCP signing policy that requires that the QA team has signed off on an image:

With `docker trust` command:

Admin:
Add the qa team to their own `targets/qa` delegation and transparently also add them to `targets/releases`. Note that this implicitly initializes the repo, and that the key files will be provided by UCP or the users themselves:

1. `docker trust signer add dtr/repo qa --key alice.pem --key bob.pem`

QA Team Member:
Load their private key into their local cache so they can sign with it:

1. `docker trust key load alice.key`

Push and sign the dtr/repo image (assuming it is already built and tagged) into the targets/qa and targets/releases roles:

2. `docker trust sign dtr/repo:tag`

Without `docker trust` command:

Previous flow documented here, detailed steps below:
Admin:
Download notary if not on Docker 4 Mac/Windows:

1. `curl ...`

Initialize the repository without pushing any content. Note that we need to specify the notary server and trust cache, as well as an autopublish flag (if we don't specify this, we need a subsequent `notary publish` command). Also, note that the user will need to input their Docker login because notary does not read the credentials from the config or credentials store:

2. `notary -s https://dtr -d ~/.docker/trust init -p dtr/repo`
3. (If we didn't have -p we'd follow with: `notary -s https://dtr -d ~/.docker/trust publish dtr/repo`)

Now we need to rotate the snapshot key to the server so that delegations can publish without snapshot key access. This command also requires the user inputs their username/password:

4. `notary -s https://dtr -d ~/.docker/trust key rotate dtr/repo -r snapshot`

Add the qa team to their own `targets/qa` delegation. Note that the key files will be provided by UCP or the users themselves. We also need to specify the paths they can sign for (all-paths in this case). This command also requires the user inputs their username/password:

2. `notary -s https://dtr -d ~/.docker/trust delegation add -p dtr/repo targets/qa alice.pem bob.pem --all-paths`

Also add the qa team to the own `targets/releases` delegation. Note that the key files will be provided by UCP or the users themselves. We also need to specify the paths they can sign for (all-paths in this case). This command also requires the user inputs their username/password:

3. `notary -s https://dtr ~/.docker/trust delegation add -p dtr/repo targets/releases alice.pem bob.pem --all-paths`

QA Team Member:
Download notary if not on Docker 4 Mac/Windows:

1. `curl ...`

Import their private key into their local cache so they can sign with it:

2. `notary -d ~/.docker/trust key import alice.key -r user`

Set the DOCKER_CONTENT_TRUST env var:

3. `export DOCKER_CONTENT_TRUST=1`

Push and sign the dtr/repo image (assuming it is already built and tagged) into the targets/qa and targets/releases roles:

4. `docker push dtr/repo:tag`