Krill User Manual

NLnet Labs

Jun 01, 2021

CONTENTS

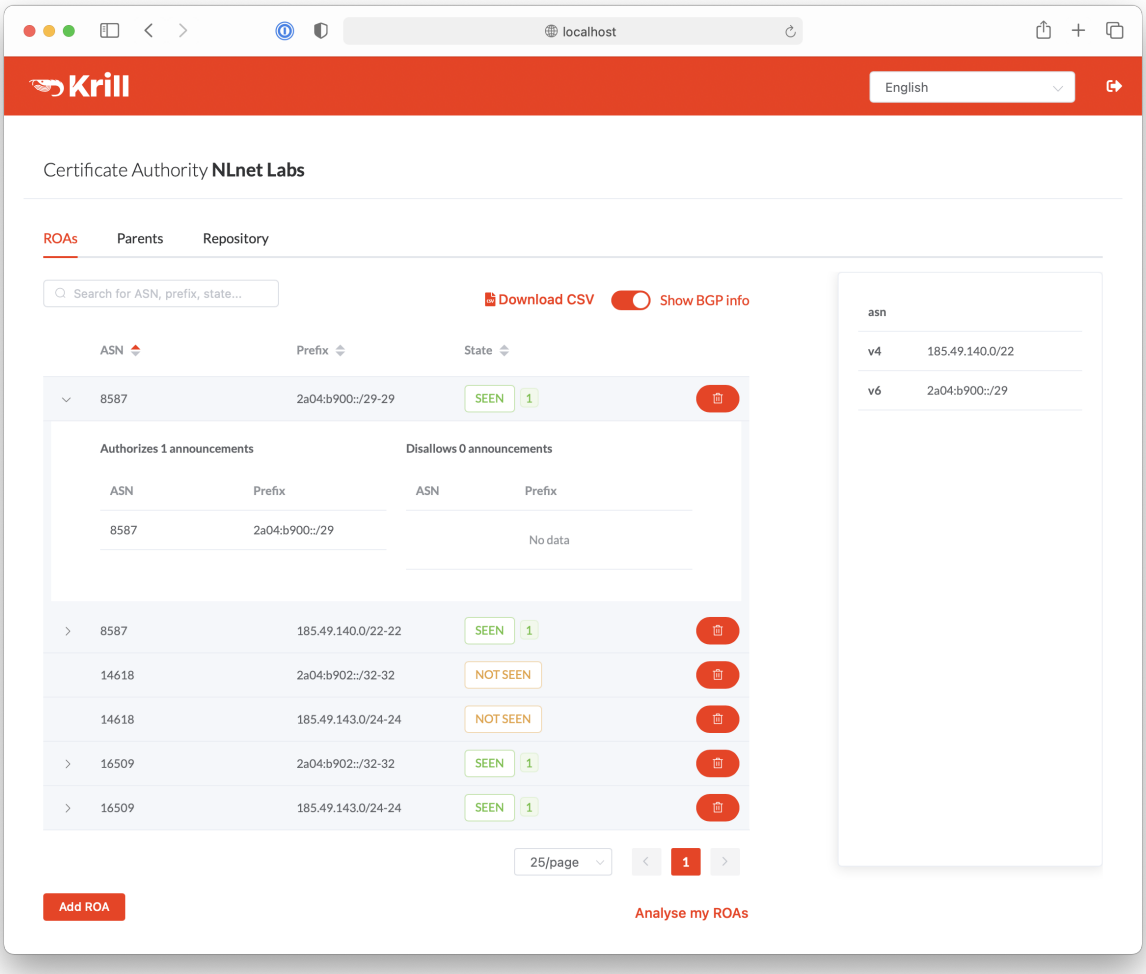
Krill is a free, open source Resource Public Key Infrastructure (RPKI) daemon, featuring a Certificate Authority (CA) and publication server, written by NLnet Labs.

Using Krill, you can run your own RPKI Certificate Authority as a child of one or more parent CAs, usually your Regional Internet Registry (RIR) or National Internet Registry (NIR).

Krill is especially convenient if your organisation holds address space in several RIR regions, or if your organisation represents multiple entities. All ASNs and IP resources you have across the various entities and RIR regions are presented as a single pool, allowing you to manage ROAs seamlessly.

Krill can also act as a parent for child CAs. This means you can delegate some of your resources down to children of your own, such as business units, departments or customers, who, in turn, manage ROAs themselves.

Krill can be managed with a web user interface, from the command line and through an API. The powerful user interface shows the RPKI validation status of your BGP announcements, warns about possible issues, and offers suggestions on ROAs you may want to create or remove. Prometheus endpoints offer monitoring of system status, ROA misconfigurations and possible BGP hijacks.



You are welcome to ask questions or post comments and ideas on our RPKI mailing list. If you find a bug in Krill, feel free to create an issue on GitHub. Krill is distributed under the Mozilla Public License 2.0.

Table of Contents

CHAPTER

ONE

BEFORE YOU START

RPKI is a very modular system and so is Krill. Which parts you need and how you fit them together depends on your situation. Before you begin with installing Krill, there are some basic concepts you should understand and some decisions you need to make.

1.1 The Moving Parts

With Krill there are two fundamental pieces at play. The first part is the Certificate Authority (CA), which takes care of all the cryptographic operations involved in RPKI. Secondly, there is the publication server which makes your certificate and ROAs available to the world.

In almost all cases you will need to run the CA that Krill provides under a parent CA, usually your Regional Internet Registry (RIR) or National Internet Registry (NIR). The communication between the parent and the child CA is initiated through the exchange of two XML files, which you need to handle manually: a child request XML and a parent response XML. This involves generating the request file, providing it to your parent, and giving the response file back to your CA.

After this initial exchange has been completed, all subsequent requests and responses are handled by the parent and child CA themselves. This includes the entitlement request and response that determines which resources you receive on your certificate, the certificate request and response, as well as the revoke request and response.

Important: The initial XML file exchange is the only manual step required to get started with Delegated RPKI. All other requests and responses, as well as re-signing and renewing certificates and ROAs are automated. **As long as Krill is running, it will automatically update the entitled resources on your certificate, as well as reissue certificates, ROAs and all other objects before they expire or become stale.** Note that even if Krill does go down, you have 8 hours to bring it back up before data starts going stale.

Whether you also run the Krill publication server depends on if you can, or want to use one offered by a third party. For the general wellbeing of the RPKI ecosystem, we would generally recommend to publish with your parent CA, if available. Setting this up is done in the same way as with the CA: exchanging a publisher request XML and a repository response XML.

1.2 Publishing With Your Parent

If you can use a publication server provided by your parent, the installation and configuration of Krill becomes extremely easy. After the installation has completed, you perform the XML exchange twice and you are done.

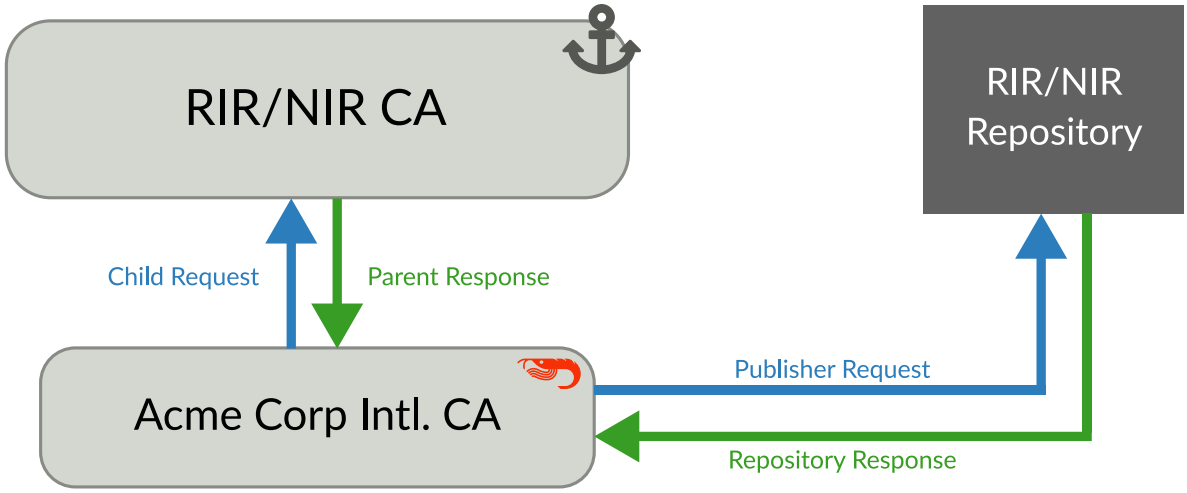


Fig. 1: A repository hosted by the parent CA, in this case the RIR or NIR.

Krill is designed to run continuously, but there is no strict uptime requirement for the CA. If the CA is not available you just cannot create or update ROAs. This means you can bring Krill down to perform maintenance or migration, as long as you bring it back up within 8 hours to ensure your cryptographic objects are re-signed before they go stale.

Note: This scenario illustrated here also applies if you use an RPKI publication server offered by a third party.

At this time, only APNIC and Brazilian NIR NIC.br offer a publication server for their members. Several RIRs have this functionality on their roadmap. This means that in most cases, you will have to publish yourself.

1.3 Publishing Yourself

Krill features a publication server, disabled by default, but which can be used to host a server for yourself, and others, such as customers or business units who run their own Krill CAs as children under your CA, and to whom you have delegated resource certificates.

If you run Krill as a publication server, you will be faced with running a public service with all related responsibilities, such as uptime and DDoS protection. This option is not recommended if you don't have a clear need to run your own server.

Read more about this option in *Running a Publication Server*

1.4 System Requirements

The system requirements for Krill are quite minimal. The cryptographic operations that need to be performed by the Certificate Authority have a negligible performance and memory impact on any modern day machine.

When you publish ROAs yourself using the Krill publication server in combination with Rsyncd and a web server of your choice, you will see traffic from several hundred relying party software tools querying every few minutes. The total amount of traffic is also negligible for any modern day situation.

Tip: For reference, NLnet Labs runs Krill in production and serves ROAs to the world using a 2 CPU / 2GB RAM / 60GB disk virtual machine. Although we only serve four ROAs and our repository size is 16KB, the situation would not be different if serving 100 ROAs.

CHAPTER

TWO

ARCHITECTURE

This section is intended to give you an overview of the architecture of Krill, which is important to keep in mind when deploying the application in your infrastructure. It will give you an understanding how and where data is stored, how to make your setup redundant and how to save and restore backups.

Warning: Krill does NOT support clustering at this time. You can achieve high availability by doing a fail-over to a standby *inactive* installation using the same data and configuration. However, you cannot have multiple active instances. This feature is on our long term roadmap.

2.1 Used Disk Space

Krill stores all of its data under the `DATA_DIR` specified in the configuration file. For users who will operate a CA under an RIR / NIR parent the following sub-directories are relevant:

Directory	Contents
data_dir/ssl	The HTTPS key and certificate used by Krill
data_dir/cas	The history of your CA(s) in raw JSON format
data_dir/pubd	If used, the history of your Publication Server

Note: Note that old versions of Krill also used the directories `data_dir/rfc8181` and `data_dir/rfc6492` for storing all protocol messages exchanged between your CAs and their parent and repository. If they are still present on your system, you can safely remove them and potentially save quite some space.

2.1.1 Archiving

Krill offers the option to archive old, less relevant, historical information related to publication. You can enable this by setting the option `archive_threshold_days` in your configuration file. If set Krill will move all publication events older than the specified number of days to a subdirectory called `archived` under the relevant data directory, i.e. `data_dir/pubd/0/archived` if you are using the Krill Publication Server and `data_dir/cas/<your-ca-name>/archived` for each of your CAs.

You can set up a cronjob to delete these events once and for all, but we recommend that you save them in long term storage if you can. The reason is that if (and only if) you have this data, you will be able to rebuild the complete Krill state based on its *audit* log of events, and irrevocably prove that no changes were made to Krill other than the changes recorded in the audit trail. We have no tooling for this yet, but we have an issue on our backlog.

2.2 Saving State Changes

You can skip this section if you're not interested in all the minute details. It is intended to explain how backup and restore works in Krill, and why a standby fail-over node can be used, but Krill's locking and storage mechanism needs to be changed in order to make multiple active nodes work.

State changes in Krill are tracked using *events*. Krill CA(s) and Publication Servers are versioned. They can only be changed by applying an *event* for a specific version. An *event* just contains the data that needs to be changed. Crucially, they cannot cause any side effects. As such, the overall state can always be reconstituted by applying all past events. This concept is called *event-sourcing*, and in this context the CAs and Publication Servers are so-called *aggregates*.

Events are not applied directly. Rather, users of Krill and background jobs will send their intent to make a change through the API, which then translates this into a so-called *command*. Krill will then lock the target aggregate and send the command to it. This locking mechanism is not aware of any clustering, and it's a primary reason why Krill cannot run as an active-active cluster yet.

Upon receiving a command the aggregate will do some work. In some cases a command can have a side-effect. For example it may instruct your CA to create a new key pair, after receiving entitlements from its parent. The key pair is random — applying a command again would result in a new random key pair. Remember that commands are not re-applied to aggregates, only their resulting events are. Thus, in this example there would be an event caused that contains the resulting key pair.

After receiving the command, the aggregate will return one of the following:

1. **An error** Usually this means that the command is not applicable to the aggregate state. For example, you may have tried to remove a ROA which does not exist.

   When Krill encounters such an error, it will store the command with some meta-information like the time the command was issued and a summary of the error, so that it can be seen in the history. It will then unlock the aggregate, so that the next command can be sent to it.

2. **No error, zero events** In this case the command turned out to be a *no-op*, and Krill just unlocks the aggregate. The command sequence counter is not updated, and the command is not saved. This is used as a feature whenever the 'republish' background job kicks in. A 'republish' command is sent, but it will only have an actual effect if there was a need to republish — e.g. a manifest would need to be re-issued before it would expire.

3. **One or more events** In this case there *is* a desired state change in a Krill aggregate. Krill will now apply and persist the changes in the following order:

   - Each event is stored. If an event already exists for a version, then the update is aborted. Because Krill cannot run as a cluster, and it uses locking to ensure that updates are done in sequence, this will only fail on the first event if a user tried to issue concurrent updates to the same CA.
   - On every fifth event a snapshot of the state is saved to a new file. If this is successful then the old snapshot (if there is one) is renamed and kept as a backup snapshot. The new snapshot is then renamed to the 'current' snapshot.
   - When all events are saved, the command is saved enumerating all resulting events, and including meta-information such as the time that the time that the command was executed. And when multiple users will be supported, this will also include *who* made a change.
   - Finally, the version information file for the aggregate is updated to indicate its current version, and command sequence counter.

Note: Krill will crash, **by design**, if there is any failure in saving any of the above files to disk. If Krill cannot persist its state it should not try to carry on. It could lead to disjoints between in-memory and on-disk state that are impossible

to fix. Therefore, crashing and forcing an operator to look at the system is the only sensible thing Krill can now do. Fortunately, this should not happen unless there is a serious system failure.

2.3 Loading State at Startup

Krill will rebuild its internal state whenever it starts. If it finds that there are surplus events or commands compared to the latest information state for any of the aggregates it will assume that they are present because, either Krill stopped in the middle of writing a transaction of changes to disk, or your backup was taken in the middle of a transaction. Such surplus files are backed up to a subdirectory called `surplus` under the relevant data directory, i.e. `data_dir/pubd/0/surplus` if you are using the Krill Publication Server and `data_dir/cas/<your-ca-name>/surplus` for each of your CAs.

2.4 Recover State at Startup

When Krill starts, it will try to go back to the last possible **recoverable** state if:

- it cannot rebuild its state at startup due to data corruption
- the environment variable: `KRILL_FORCE_RECOVER` is set
- the configuration file contains `always_recover_data = true`

Under normal circumstances performing this recovery will not be necessary. It can also take significant time due to all the checks performed. So, we do **not recommend** forcing recovery when there is no data corruption.

Krill will try the following checks and recovery attempts:

- Verify each recorded command and its effects (events) in their historical order.
- If any command or event file is corrupt it will be moved to a subdirectory called `corrupt` under the relevant data directory, and all subsequent commands and events will be moved to a subdirectory called `surplus` under the relevant data directory.
- Verify that each snapshot file can be parsed. If it can't then this file is moved to the relevant `corrupt` sub-directory.
- If a snapshot file could not be parsed, try to parse the backup snapshot. If this file can't be parsed, move it to the relevant `corrupt` sub-directory.
- Try to rebuild the state to the last recoverable state, i.e. the last known good event. Note that if this pre-dates the available snapshots, or, if no snapshots are available this means that Krill will try to rebuild state by replaying all events. If you had enabled archiving of events, it will not be able rebuild state.
- If rebuilding state failed, Krill will now exit with an error.

Note that in case of data corruption Krill may be able to fall back to an earlier recoverable state, but this state may be far in the past. You should always verify your ROAs and/or delegations to child CAs in such cases.

Of course, it's best to avoid data corruption in the first place. Please monitor available disk space, and make regular backups.

2.5 Backup and Restore

Backing up Krill is as simple as backing up its data directory. There is no need to stop Krill during the backup. To restore put back your data directory and make sure that you refer to it in the configuration file that you use for your Krill instance. As described above, if Krill finds that the backup contains an incomplete transaction, it will fall back to the state prior to it.

Warning: You may want to **encrypt** your backup, because the `data_dir/ssl` directory contains your private keys in clear text. Encrypting your backup will help protect these, but of course also implies that you can only restore if you have the ability to decrypt.

2.6 Krill Upgrades

All Krill versions 0.4.1 and upwards can be automatically upgraded to the current version. Any required data migrations will be performed automatically. To do so we recommend that you:

- backup your krill data directories
- install the new version of Krill
- stop the running Krill instance
- start Krill again, using the new binary, and the same configuration

If you want to test if data migrations will work correctly for your data, you can do the following:

- copy your data directory to another system
- set the env variable `KRILL_UPGRADE_ONLY=1`
- create a configuration file, and set `data_dir=/path/to/your/copy`
- start up Krill

Krill will then perform the data migrations, rebuild its state, and then exit before doing anything else.

Note: Downgrading Krill data is not supported. Downgrading can only be achieved by installing a previous version of Krill and restoring a backup that matches this version.

2.7 Proxy and HTTPS

Krill uses HTTPS and refuses to do plain HTTP. By default Krill will generate a 2048 bit RSA key and self-signed certificate in `/ssl` in the data directory when it is first started. Replacing the self-signed certificate with a TLS certificate issued by a CA works, but has not been tested extensively. By default Krill will only be available under `https://localhost:3000`.

If you need to access the Krill UI or API (also used by the CLI) from another machine you can use use a proxy server such as NGINX or Apache to proxy requests to Krill. This proxy can then also use a proper HTTPS certificate and production grade TLS support.

2.7.1 Proxy Krill UI

The Krill UI and assets are hosted directly under the base path `/`. So, in order to proxy to the Krill UI you should proxy ALL requests under `/` to the Krill back-end.

Note that although the UI and API are protected by a token, you should consider further restrictions in your proxy setup, such as restrictions on source IP or adding your own authentication.

2.7.2 Proxy Krill as Parent

If you delegated resources to child CAs then you will need to ensure that these children can reach your Krill. Child requests for resource certificates are directed to the `/rfc6492` directory under the `service_uri` that you defined in your configuration file.

Note that contrary to the UI you should not add any additional authentication mechanisms to this location. **RFC 6492** uses cryptographically signed messages sent over HTTP and is secure. However, verifying messages and signing responses can be computationally heavy, so if you know the source IP addresses of your child CAs, you may wish to restrict access based on this.

2.7.3 Proxy Krill as Publication Server

If you are running Krill as a Publication Server, then you should read *here* how to do the Publication Server specific set up.

> **Warning:** We recommend that you do **not** make Krill available to the public internet unless you really need remote access to the UI or API, or you are serving as parent CA or Publication Server for other CAs.

CHAPTER

THREE

INSTALL AND RUN

Getting started with Krill is quite easy by either installing a Debian and Ubuntu package, building from Cargo or using *Docker*. In case you intend to serve your RPKI certificate and ROAs to the world yourself or you want to offer this as a service to others, you will also need to have a public Rsyncd and HTTPS web server available.

3.1 Installing with Debian and Ubuntu Packages

Pre-built Debian/Ubuntu packages are available for recent operating system versions on x86_64 platforms. These can be installed using the standard `apt`, `apt-get` and `dpkg` commands as usual.

Unlike with installing with Cargo there is no need to have Rust or a C toolchain installed. Additionally, the packages come with systemd service file to easily start and stop the Krill daemon.

Note: For the oldest platforms, Ubuntu 16.04 LTS and Debian 9, the packaged krill binary is statically linked with OpenSSL 1.1.0 as this is the minimum version required by Krill and is higher than available in the official package repositories for those platforms.

To install Krill from the NLnet Labs package repository:

1. Run `cargo uninstall krill` if you previously installed Krill with Cargo.
2. Add the line below that corresponds to your operating system to `/etc/apt/sources.list` or `/etc/apt/sources.list.d/`:

```
deb [arch=amd64] https://packages.nlnetlabs.nl/linux/debian/ stretch main
deb [arch=amd64] https://packages.nlnetlabs.nl/linux/debian/ buster main
deb [arch=amd64] https://packages.nlnetlabs.nl/linux/ubuntu/ xenial main
deb [arch=amd64] https://packages.nlnetlabs.nl/linux/ubuntu/ bionic main
deb [arch=amd64] https://packages.nlnetlabs.nl/linux/ubuntu/ focal main
```

2. Add the repository signing key to the listed of trusted keys:

```
wget -qO- https://packages.nlnetlabs.nl/aptkey.asc | sudo apt-key add -
```

3. Install Krill using `sudo apt-get update` and `sudo apt-get install krill`.
4. Review the generated configuration file at `/etc/krill.conf`. Pay particular attention to the `service_uri` and `auth_token` settings.
5. Once happy with the settings use `sudo systemctl enable --now krill` to instruct systemd to enable the Krill service at boot and to start it immediately.

Tip: The configuration file was generated for you using the `krillc config simple` command. You can find a full example configuration file with defaults in the GitHub repository.

The krill daemon runs as user `krill` and stores its data in `/var/lib/krill`. You can manage the Krill daemon using the following commands:

- Review the Krill logs with `journalctl -u krill`, or view just the most recent entries with `systemctl status krill`.
- Stop Krill with `sudo systemctl stop krill`.
- Learn more about Krill using `man krill` and `man krillc`.
- Upgrade Krill by running `apt-get update` and `apt-get install krill`.

3.2 Installing with Cargo

There are three things you need for Krill: Rust, the C toolchain and OpenSSL. You can install Krill on any operating system where you can fulfil these requirements, but we will assume that you will run this on a UNIX-like OS.

3.2.1 Rust

The Rust compiler runs on, and compiles to, a great number of platforms, though not all of them are equally supported. The official Rust Platform Support page provides an overview of the various support levels.

While some system distributions include Rust as system packages, Krill relies on a relatively new version of Rust, currently 1.42 or newer. We therefore suggest to use the canonical Rust installation via a tool called **`rustup`**.

To install **`rustup`** and Rust, simply do:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Alternatively, visit the official Rust website for other installation methods.

You can update your Rust installation later by running:

```
rustup update
```

For some platforms, **`rustup`** cannot provide binary releases to install directly. The Rust Platform Support page lists several platforms where official binary releases are not available, but Rust is still guaranteed to build. For these platforms, automated tests are not run so it's not guaranteed to produce a working build, but they often work to quite a good degree.

One such example that is especially relevant for the routing community is OpenBSD. On this platform, patches are required to get Rust running correctly, but these are well maintained and offer the latest version of Rust quite quickly.

Rust can be installed on OpenBSD by running:

```
pkg_add rust
```

Another example where the standard installation method does not work is CentOS 6, where you will end up with a long list of error messages about missing assembler instructions. This is because the assembler shipped with CentOS 6 is too old.

You can get the necessary version by installing the Developer Toolset 6 from the Software Collections repository. On a virgin system, you can install Rust using these steps:

```
sudo yum install centos-release-scl
sudo yum install devtoolset-6
scl enable devtoolset-6 bash
curl https://sh.rustup.rs -sSf | sh
source $HOME/.cargo/env
```

3.2.2 C Toolchain

Some of the libraries Krill depends on require a C toolchain to be present. Your system probably has some easy way to install the minimum set of packages to build from C sources. For example, **`apt install build-essential`** will install everything you need on Debian/Ubuntu.

If you are unsure, try to run **`cc`** on a command line and if there's a complaint about missing input files, you are probably good to go.

3.2.3 OpenSSL

Your system will likely have a package manager that will allow you to install OpenSSL in a few easy steps. For Krill, you will need **`libssl-dev`**, sometimes called **`openssl-dev`**. On Debian-like Linux distributions, this should be as simple as running:

```
apt install libssl-dev openssl pkg-config
```

3.2.4 Building

The easiest way to get Krill is to leave it to cargo by saying:

```
cargo install --locked krill
```

If you want to update an installed version, you run the same command but add the `-f` flag, a.k.a. force, to approve overwriting the installed version.

The command will build Krill and install it in the same directory that cargo itself lives in, likely `$HOME/.cargo/bin`. This means Krill will be in your path, too.

3.2.5 Generate Configuration File

After the installation has completed, there are just two things you need to configure before you can start using Krill. First, you will need a data directory, which will store everything Krill needs to run. Secondly, you will need to create a basic configuration file, specifying a secret token and the location of your data directory.

The first step is to choose where your data directory is going to live and to create it. In this example we are simply creating it in our home directory.

```
mkdir ~/data
```

Krill can generate a basic configuration file for you. We are going to specify the two required directives, a secret token and the path to the data directory, and then store it in this directory.

```
krillc config simple --token correct-horse-battery-staple --data ~/data/ > ~/
↪data/krill.conf
```

Note: If you wish to run a self-hosted RPKI repository with Krill you will need to use a different `krillc config` command. See *Running a Publication Server* for more details.

You can find a full example configuration file with defaults in the GitHub repository.

3.2.6 Start and Stop the Daemon

There is currently no standard script to start and stop Krill. You could use the following example script to start Krill. Make sure to update the `DATA_DIR` variable to your real data directory, and make sure you saved your `krill.conf` file there.

```
#!/bin/bash
KRILL="krill"
DATA_DIR="/path/to/data"
KRILL_PID="$DATA_DIR/krill.pid"
CONF="$DATA_DIR/krill.conf"
SCRIPT_OUT="$DATA_DIR/krill.log"

nohup $KRILL -c $CONF >$SCRIPT_OUT 2>&1 &
echo $! > $KRILL_PID
```

You can use the following sample script to stop Krill:

```
#!/bin/bash
DATA_DIR="/path/to/data"
KRILL_PID="$DATA_DIR/krill.pid"

kill `cat $KRILL_PID`
```

CHAPTER

FOUR

GET STARTED WITH KRILL

Before you can start managing your own Route Origin Authorisations (ROAs) you need to do a one time setup where you:

- create your Certificate Authority (CA)
- connect to a Publication Server
- connect to a Parent CA (typically a Regional or National Internet Registry)

This can be easily achieved using the user interface. Connecting to the Publication Server and Parent CA is done by exchanging a couple of XML files. After this initial setup, and you can simply *manage your ROAs*.

If you just want to try out Krill (or a new version) you can use the testbed environment provided by NLnet Labs for this.

If you are using the defaults you can access the user interface in a browser on the server running Krill at `https://localhost:3000`. By default, Krill generates a self-signed TLS certificate, so you will have to accept the security warning that your browser will give you.

If you want to access the UI, or use the CLI, from another computer, you can either *set up a reverse proxy* on your server running Krill, or set up local port forwarding with SSH, for example:

```
ssh -L 3000:localhost:3000 user@krillserver.example.net
```

Here we will guide you through the set up process using the UI, but we will also link to the relevant subcommands of the *command line interface (CLI)*

4.1 Login

The login will ask you to enter the secret token you configured for Krill.

If you are using the CLI you will need to specify the token using the `--token` option. Because the CLI does not have a session, you will need to specify this for each command. Alternatively, you can set the the `KRILL_CLI_TOKEN` environment variable.

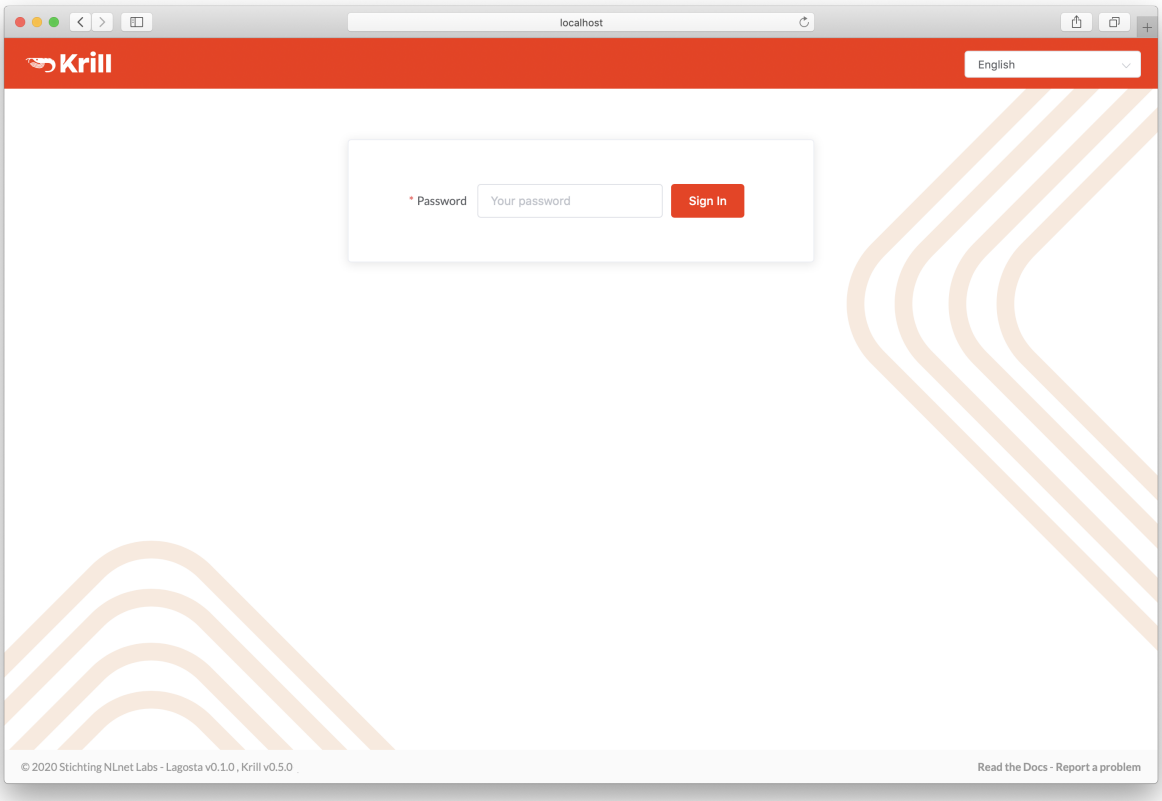


Fig. 1: Enter your secret token to access Krill

4.2 Create your Certificate Authority

Next, you will see the Welcome screen where you can create your Certificate Authority (CA). It will be used to configure delegated RPKI with one or multiple parent CAs, usually your Regional or National Internet Registry.

The handle you select is not published in the RPKI but used as identification to parent and child CAs you interact with. Please choose a handle that helps others recognise your organisation. Once set, the handle cannot be changed.

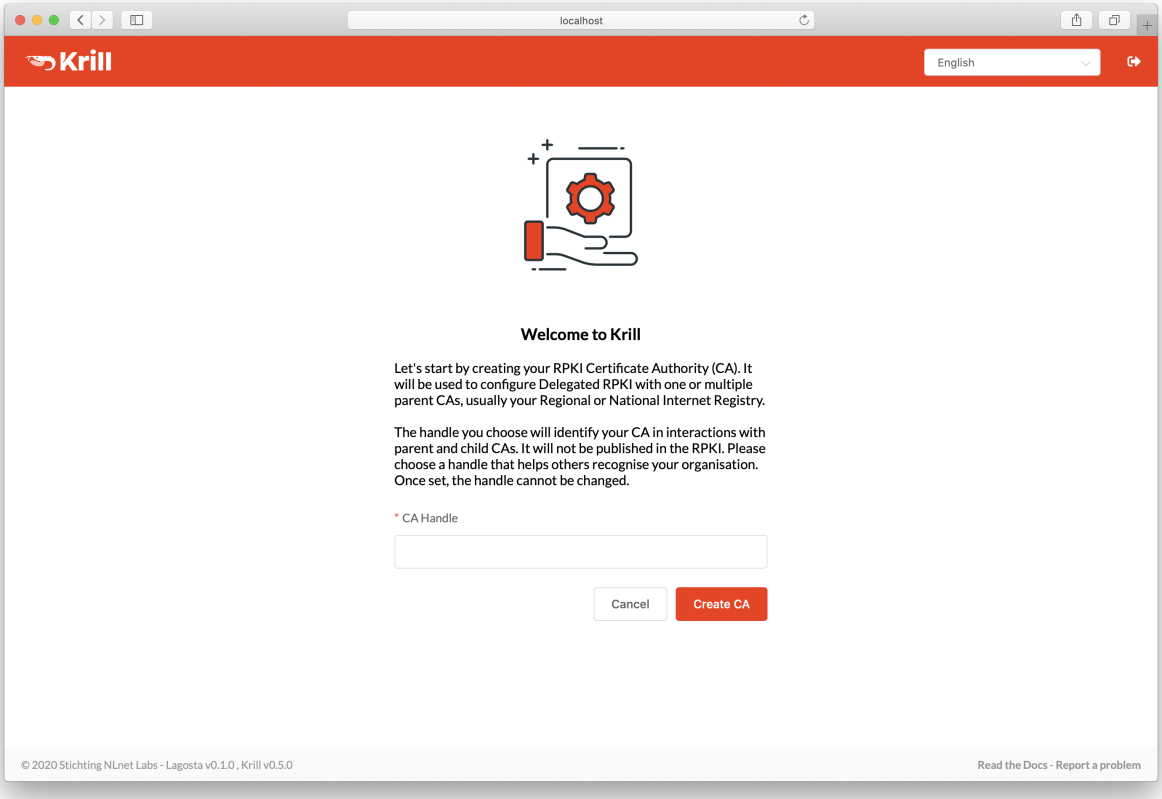


Fig. 2: Enter a handle for your Certification Authority

If you are using the CLI you can create your CA using the subcommand *krillc add*.

4.3 RIR and NIR Interactions

If you hold resources in one or more RIR or NIR regions, you will need to have access to the respective member portals and the permission to configure delegated RPKI.

AFRINIC https://my.afrinic.net

APNIC https://myapnic.net

ARIN https://account.arin.net

LACNIC https://milacnic.lacnic.net

RIPE NCC https://my.ripe.net

Most RIRs have a few considerations to keep in mind.

4.3.1 AFRINIC

AFRINIC have delegated RPKI available in their test environment, but it's not operational yet. Work to bring it to production is planned for 2021.

4.3.2 APNIC

If you are already using the hosted RPKI service provided by APNIC and you would like to switch to delegated RPKI, there is currently no option for this with MyAPNIC. Please open a ticket with the APNIC help desk to resolve this.

Please note that APNIC offers RPKI publication as a service upon request. It is highly recommended to make use of this, as it relieves you of the need to run a highly available repository yourself.

4.3.3 LACNIC

Although LACNIC offers delegated RPKI, it is not possible to configure this in their member portal yet. While the procedures are still being defined, please open a ticket via hostmaster@lacnic.net to get started.

4.3.4 RIPE NCC

When you are a RIPE NCC member who does not have RPKI configured, you will be presented with a choice if you would like to use hosted or non-hosted RPKI.

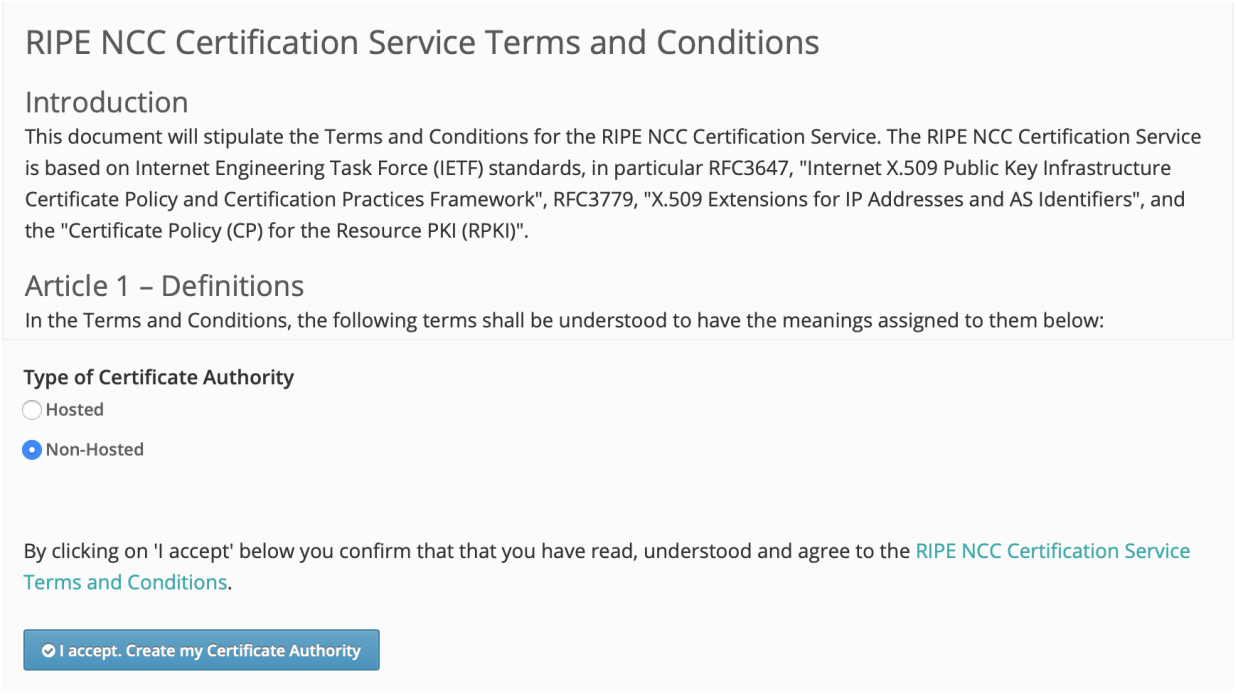


Fig. 3: RIPE NCC RPKI setup screen

If you want to set up delegated RPKI with Krill, you will have to choose non-hosted. If you are already using the hosted service and you would like to switch, then there is currently no option for that in the RIPE NCC portal.

Make a note of the ROAs you created and then send an email to rpki@ripe.net requesting your hosted CA to be deleted, making sure to mention your registration id. After deletion, you will land on the setup screen from where you can choose non-hosted RPKI.

4.4 Hosted Publication Server

Your RIR or NIR may also provide an RPKI publication server. You are free to publish your certificate and ROAs anywhere you like, so a third party may provide an RPKI publication server as well. Using an RPKI publication server relieves you of the responsibility to keep a public rsync and web server running at all times to make your certificate and ROAs available to the world.

Of the five RIRs, only APNIC currently offers RPKI publication as a service for their members, upon request. Most other RIRs have it on their roadmap. NIC.br, the Brazilian NIR, provides an RPKI repository server for their members as well. This means that in most cases you will have to publish your certificate and ROAs yourself, as described in the *Running a Publication Server* section.

4.5 Repository Setup

Before Krill can request a certificate from a parent CA, it will need to know where it will publish. You can add a parent before configuring a repository for your CA, but in that case Krill will postpone requesting a certificate until you have done so.

In order to register your CA as a publisher, you will need to copy the **RFC 8183** Publisher Request XML and supply it to your Publication Server. You can retrieve this file with the CLI subcommand *krillc repo request*, or you can simply use the UI:

Your publication server provider will give you a repository response XML. You can use the CLI subcommand krillc repo update to tell add this configuration to your CA, or you can simply use the UI:

4.6 Parent Setup

After successfully configuring the repository, the next step is to configure your parent CA. You will need to present your CA's **RFC 8183** child request XML file to your parent. You can get this file using the CLI subcommand *krillc parents request*, or you can simply use the UI:

Your RIR or NIR will provide you with a parent response XML. You can use the CLI subcommand *krillc parents add* for this, or you can simply paste or upload it using the UI:

After a few moments your parent will process your entitled resources and you will see them appearing on your certificate which is visible in the ROAs tab. Now you can start creating ROAs.

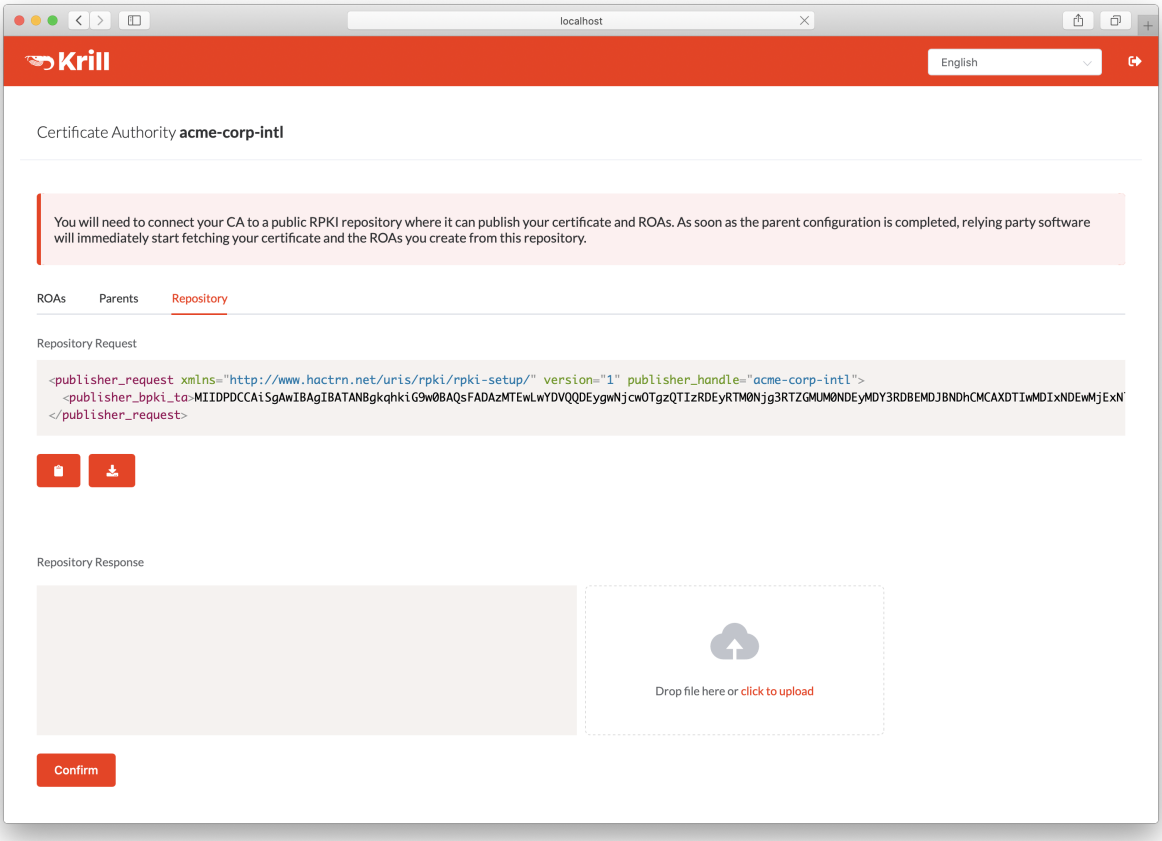


Fig. 4: Copy the publisher request XML or download the file

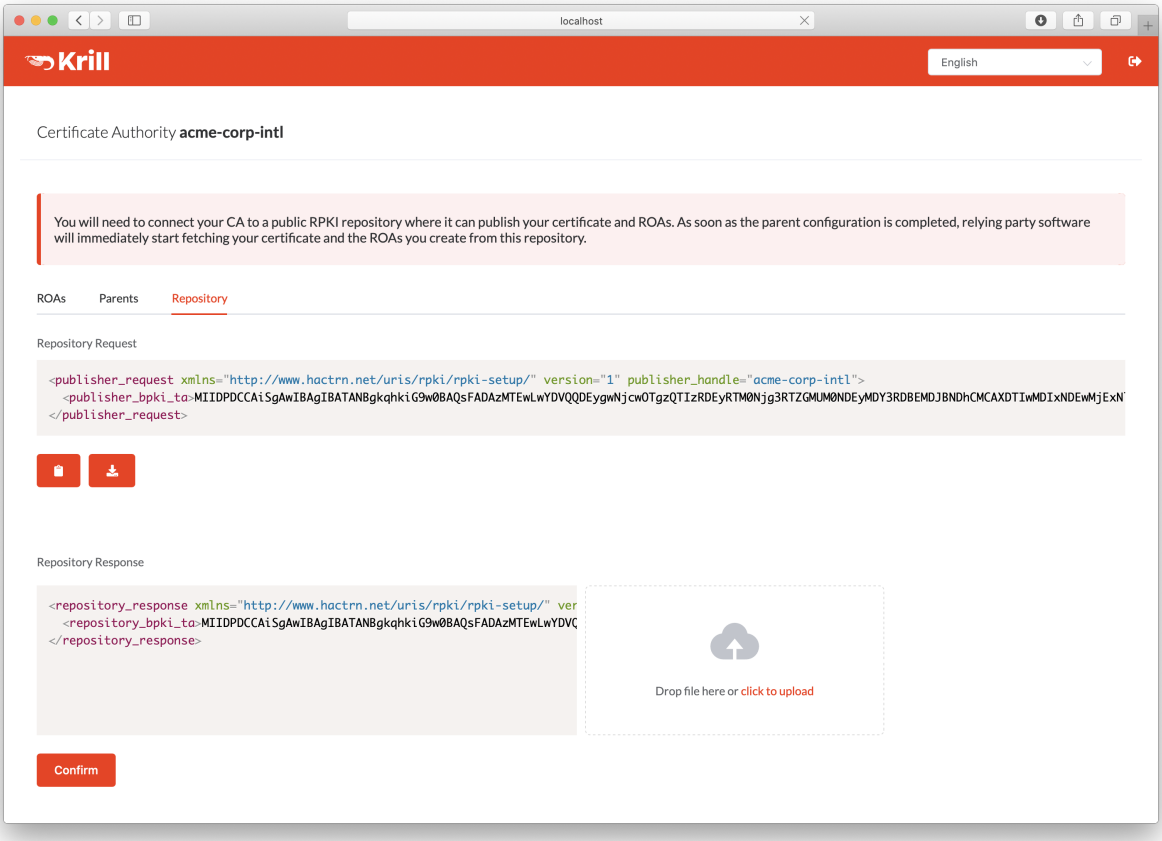


Fig. 5: Paste or upload the repository response XML

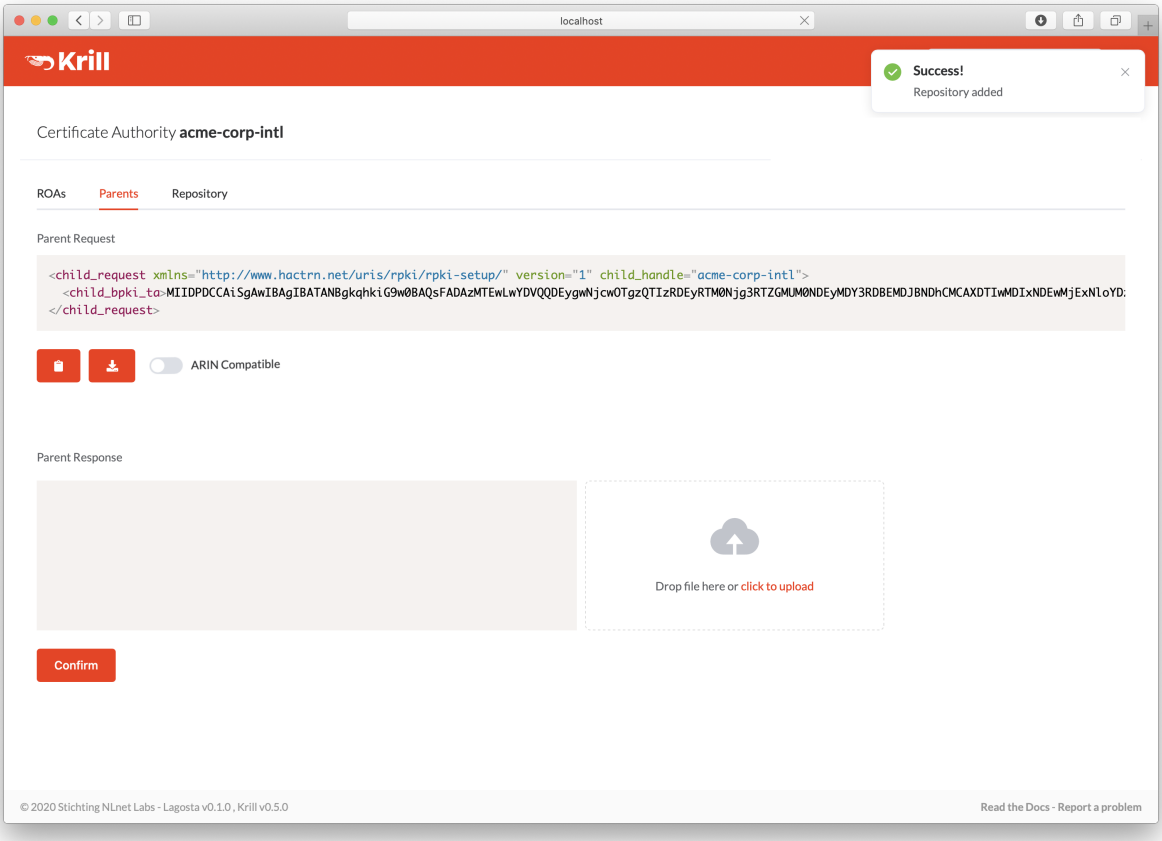


Fig. 6: Copy the child request XML or download the file

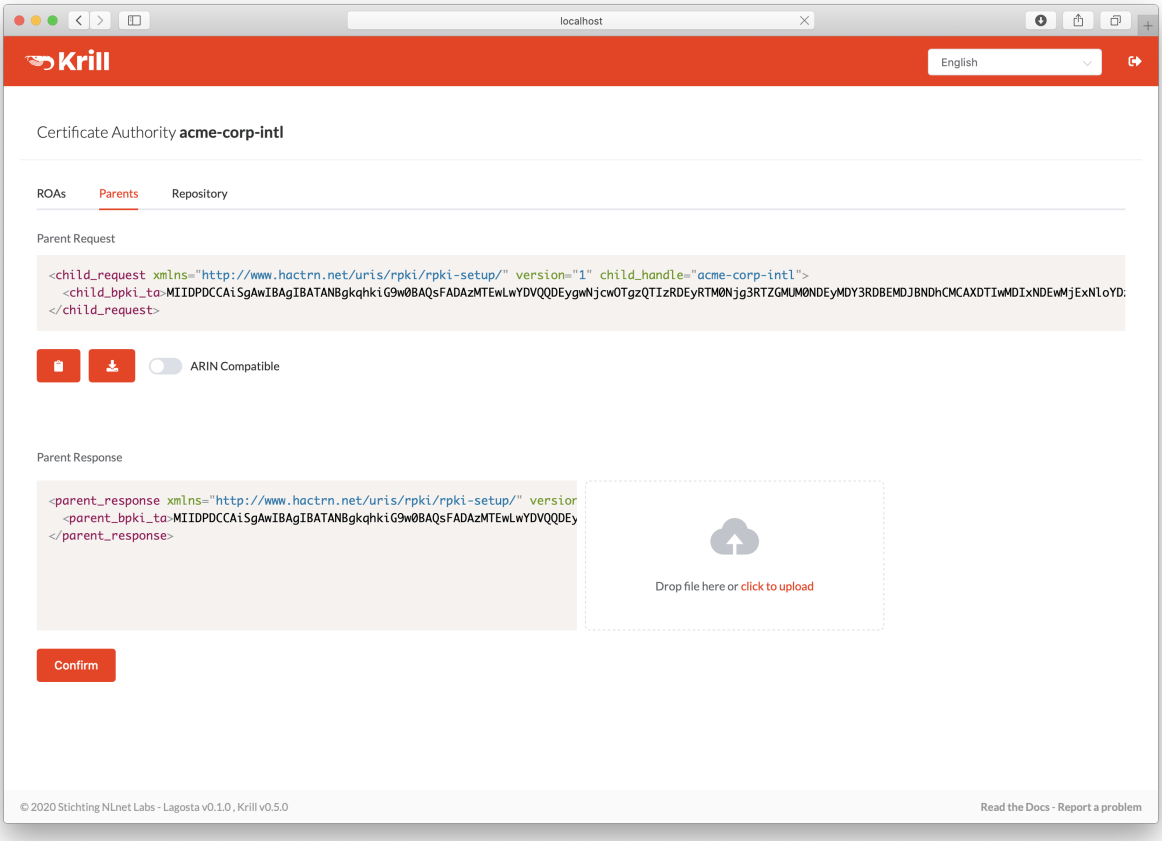


Fig. 7: Paste or upload the parent response XML

CHAPTER

FIVE

MANAGE ROAS

Once you have successfully *set up the parent and repository*, you are now running delegated RPKI. You can now start creating ROAs.

5.1 Show BGP Info

Krill automatically downloads BGP announcement information from the RIPE NCC Routing Information Service (RIS) and uses this to analyse the known BGP announcements for the address space on your resource certificate(s). This allows Krill to show the RPKI validation status of your announcements, warn about possible issues, and offer suggestions on ROAs you may want to create or remove.

If you just set up your Krill instance all your announcements will have the status *NOT FOUND*, as you have not created any ROAs covering them yet.

Once you start authorising BGP announcements made with your IP prefixes, Krill recognises the following 'States' in its analysis:

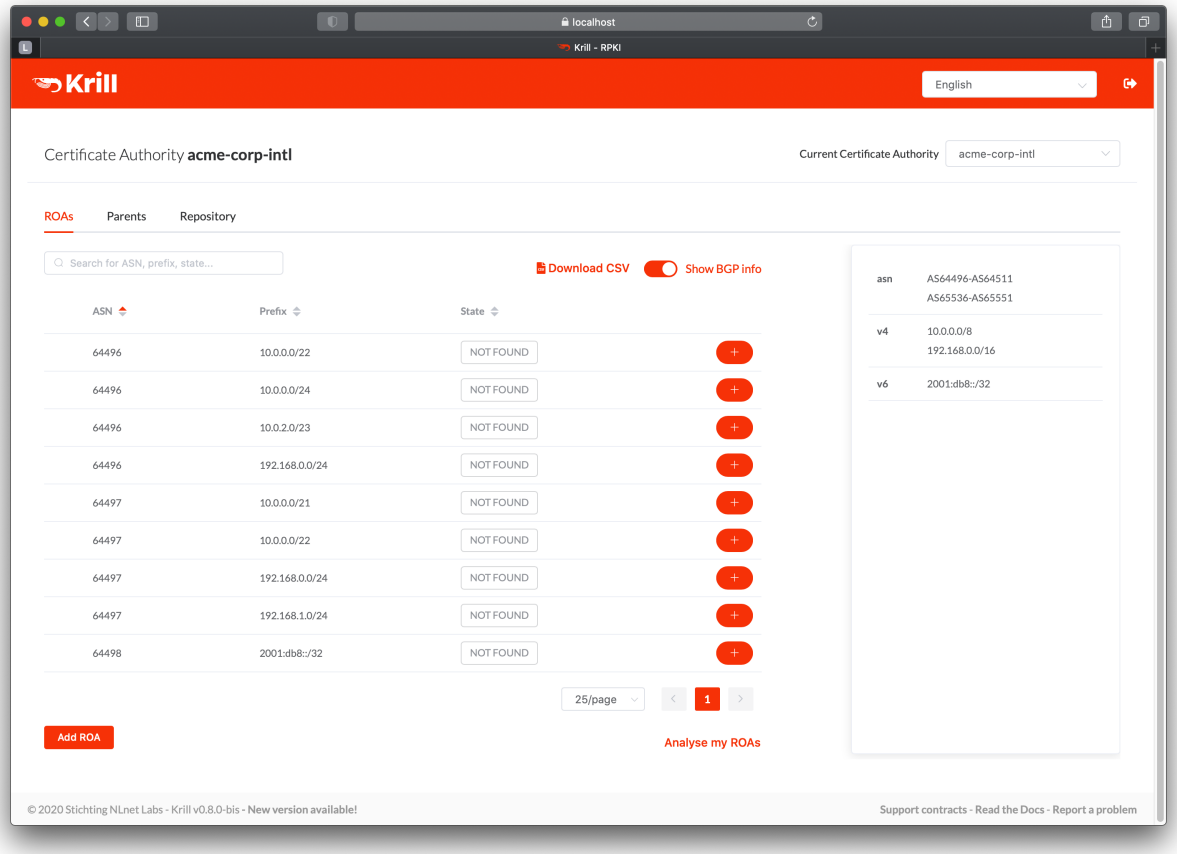


Fig. 1: When you first start, all your announcements are ‘NOT FOUND’

State	Explanation
NOT FOUND	This announcement is not covered by any of your ROAs
INVALID ASN	The prefix for this announcement is covered by one or more of your ROAs. However, none of those ROAs allow announcements of this prefix by this ASN.
INVALID LENGTH	The ASN for this announcement is covered by one or more of your ROAs. However, the prefix is more specific than allowed.
SEEN	This is a ROA you created which allows at least one known BGP announcement. Note it may also disallow one or more other announcements. You can show details if you click on the '>' icon.
TOO PERMISSIVE	This ROA uses the max length field to allow multiple announcements, but Krill does not see *all* most specific announcements in its BGP information.
REDUNDANT	This is a ROA you created which is included in full by at least one other ROA you created. I.e. you have a ROA for the same ASN, covering this prefix and including the maximum length.
NOT SEEN	This is a ROA you created but it does not cover any known announcements. This may be a ROA you created for a backup or planned announcement. On the other hand, this could also be a stale ROA in which case it is better to remove it.
DISALLOWING	This is a ROA for which no allowed announcements are seen, yet it **disallows** one or more announcements. If this is done on purpose it may be better to create a ROA for ASN 0 instead.
AS0	This is a ROA you created for a prefix with ASN 0. Since ASN 0 cannot occur in BGP such ROAs are effectively used to disallow announcements of prefixes

If you disable the *Show BGP Info* toggle, Krill will just show you your plain ROAs. You can also disable downloading the RIS dump files altogether if you set the following directive in your krill.conf file:

```
bgp_risdumps_enabled = false
```

5.2 ROA Suggestions

Warning: You should always verify the suggestions done by Krill. Krill bases its analysis on information collected by the RIPE NCC Routing Information Service (RIS) and saved in aggregated dumps every 8 hours. This means the announcements that Krill sees may be outdated. More importantly they may include announcements by others that you do **NOT** wish to allow. In addition, you may not see your own announcements if you inadvertently invalidated them, because such announcements are often rejected and therefore may not reach the RIS Route Collectors.

We plan to add support to use other data sources in future, which will allow you to inform Krill about the announcements that you do on your own eBGP sessions.

If you click *Analyse my ROAs* under the table in the ROAs tab, Krill will suggest the following changes for the following 'State' values:

State	Add/Remove	Notes
NOT FOUND	Add	
INVALID ASN	Add	Be careful when adding a ROA for a new ASN. The information is based on what is seen in BGP, but this may include malicious or accidental hijacks that you do **NOT** wish to allow. NOTE: Krill will not suggest to allow announcements for a new ASN if you created an AS0 ROA for the prefix.
INVALID LENGTH	Add	If you are sure that this announcement is valid, then you should create a ROA for it. However, there is a (remote) chance that this is a malicious hijack where your ASN was prepended. In such cases you should of course **NOT** allow it.
TOO PERMISSIVE	BOTH	Krill will suggest to remove the permissive ROA and replace it with ROAs for all specific announcements presently seen in BGP. This is inline with recommendations in this draft in the IETF. However, if you need to pre-provision specific announcements from your ASN, e.g. for anti DDoS purposes, then you may wish to keep the permissive ROA as is.
DISALLOWING	Remove	If you want to create a ROA to disallow announcements then it may be better to create an AS0 ROA instead.
NOT SEEN	Remove	Keep the ROA if it is for a planned or backup announcement.
REDUNDANT	Remove	

5.3 Add a ROA

Click the *Add ROA* button, then fill in the authorised ASN and one of your prefixes in the form. The maximum prefix length will automatically match the prefix you entered to follow best operational practices, but you can change it as desired.

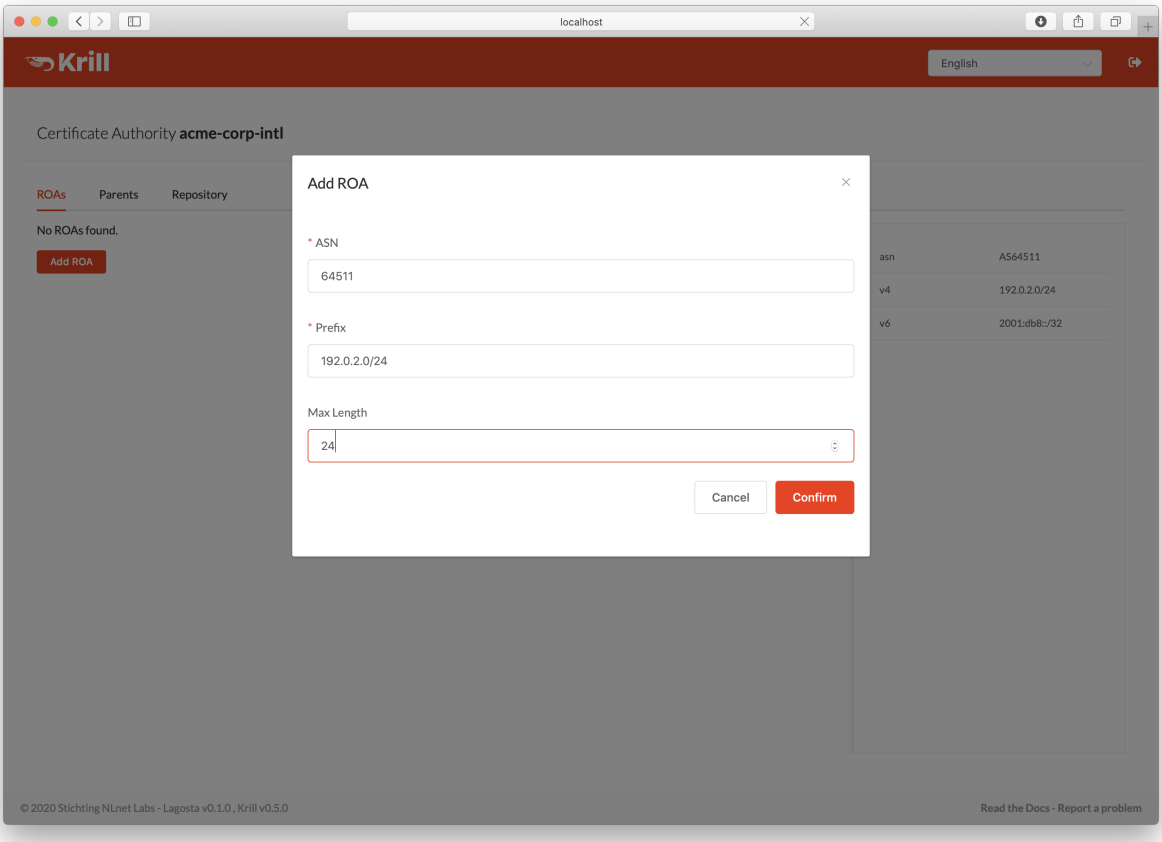


Fig. 2: Adding a new ROA

If you prefer to use the CLI then you can manage ROAs using the subcommand *krillc roas*.

CHAPTER

SIX

USING THE CLI OR API

6.1 Introduction

Every function of Krill can be controlled from the command line interface (CLI). The Krill CLI is a wrapper around the API which is based on JSON over HTTPS.

We will document all current functions below, providing examples of both the CLI and API.

Note that you can use the CLI from another machine, but then you will need to set up a proxy server in front of Krill and make sure that it has a real TLS certificate.

To use the CLI you need to invoke **`krillc`** followed by one or more subcommands, and some arguments. Help is built-in:

```
USAGE:
    krillc <subcommand..> [FLAGS] [OPTIONS]

FLAGS:
        --api         Only show the API call and exit. Or set env: KRILL_CLI_API=1
    -h, --help        Prints help information
    -V, --version     Prints version information

OPTIONS:
    -c, --ca <name>          The name of the CA you wish to control. Or set env: KRILL_
↪CLI_MY_CA
    -f, --format <type>      Report format: none|json|text (default). Or set env:␣
↪KRILL_CLI_FORMAT
    -s, --server <URI>       The full URI to the krill server. Or set env: KRILL_CLI_
↪SERVER
    -t, --token <string>     The secret token for the krill server. Or set env: KRILL_
↪CLI_TOKEN
```

6.2 Setting Defaults

As noted in the OPTIONS help text above you can set default values via environment variables for the most common arguments that need to be supplied to `krillc` subcommands. When setting environment variables note the following requirements:

- `KRILL_CLI_SERVER` must be in the form `https://<host:port>/`.
- `KRILL_CLI_MY_CA` must consist only of alphanumeric characters, dashes and underscores, i.e. `a-zA-Z0-9_`.

For example:

```
export KRILL_CLI_TOKEN="correct-horse-battery-staple"
export KRILL_CLI_MY_CA="Acme-Corp-Intl"
```

If you do use the command line argument equivalents, you will override whatever value you set in the ENV. Krill will give you a friendly error message if you did not set the applicable ENV variable, and don't include the command line argument equivalent.

6.3 Explore the API

The reference below documents the available `krillc` subcommands and the equivalent API functions by example.

You can also explore the CLI and API yourself:

- Each subcommand can be prefixed with `help` to access the CLI built-in help
- You can always use `--api` argument to make the CLI print out the API call that it would do, without actually sending it to the server.
- You can use `--format=json` to have the API print out the JSON returned by the server without reformatting or filtering information. Of course, be careful if you use this option for subcommands with side-effects, such as `krillc delete --ca <ca>`

If you want a to have a safe sandbox environment to test your Krill CA and really explore the API, then we recommend that you set up a local Krill testbed as described in doc_krill_testbed.

Tip: Click subcommand names in this section to jump to its detailed description.

Subcommands for managing your Krill server:

```
config          Creates a configuration file for krill and prints it to STDOUT
health          Perform an authenticated health check
info            Show server info
```

Subcommands for adding / removing CA instances in your Krill server:

```
add             Add a new CA
delete          Delete a CA and let it withdraw its objects and request␣
↪revocation. WARNING: Irreversible!
list            List the current CAs
```

Subcommands for initialising a CA:

```
parents         Manage parents for a CA.
repo            Manage the repository for a CA.
```

Subcommands for showing the details of a CA:

```
show            Show details of a CA.
issues          Show issues for a CA
history         Show the history of a CA
```

Manage ROAs:

```
roas            Manage ROAs for a CA.
```

Other operations:

```
bulk            Manually trigger refresh/republish/resync for all CAs
children        Manage children for a CA
keyroll         Perform a manual key rollover for a CA
```

6.4 krillc config

This subcommand is implemented on the CLI only and is intended to help generate a configuration file which can be used for your Krill server.

We currently support two subcommands for this: *krillc config simple* and *krillc config user*. The first can be used to generate general server configuration. The second can be used to generate user (*id*) entries to use if you want to have multiple local users access the Krill UI by their own name and password.

6.5 krillc health

Perform an authenticated health check. Verifies that the specified Krill server can be connected to, is able to verify the specified token and is, at least thus far, healthy. This does NOT check whether your CAs have any issues, please have a look at the *issues* subcommand for this.

Can be used in automation scripts by checking the exit code:

Exit Code	Meaning
0	the Krill server appears to be healthy.
non-zero	incorrect server URI, token, connection failure or server error.

Example CLI:

```
$ krillc health
$ echo $?
0
```

Example API:

```
$ krillc health --api
GET:
  https://localhost:3000/api/v1/authorized
Headers:
  Authorization: Bearer secret
```

If you need to do an unauthorized health check, then you can just call the following endpoint instead. This will always return a 200 OK response if the server is running:

```
GET:
  https://localhost:3000/health
```

6.6 krillc info

Show server info. Prints the version of the Krill *server* and the date and time that it was last started, e.g.:

Example CLI:

```
$ krillc info
Version: 0.9.0
Started: 2021-04-07T12:36:00+00:00
```

Example API call:

```
$ krillc info --api
  GET:
    https://localhost:3000/stats/info
  Headers:
    Authorization: Bearer secret
```

Example API resonse:

```
{
  "version": "0.9.0",
  "started": 1617798960
}
```

6.7 krillc add

Adds a new CA.

When adding a CA you need to choose a handle, essentially just a name. The term "handle" comes from **RFC 8183** and is used in the communication protocol between parent and child CAs, as well as CAs and publication servers. The handle may consist of alphanumeric characters, dashes and underscores, i.e. `a-zA-Z0-9_`.

The handle you select is not published in the RPKI but used as identification to parent and child CAs you interact with. You should choose a handle that helps others recognise your organisation. Once set, the handle cannot be be changed as it would interfere with the communication between parent and child CAs, as well as the publication repository.

When a CA has been added, it is registered to publish locally in the Krill instance where it exists, but other than that it has no configuration yet. In order to do anything useful with a CA you will first have to add at least one parent to it, followed by some Route Origin Authorisations and/or child CAs.

Example CLI:

```
$ krillc add --ca newca
```

Example API:

```
$ krillc add --ca newca --api
POST:
  https://localhost:3000/api/v1/cas
Headers:
  content-type: application/json
  Authorization: Bearer secret
Body:
```

(continues on next page)

(continued from previous page)

```
{
  "handle": "newca"
}
```

The API response is an empty 200 OK response, unless an issue occurred - e.g. the handle was already in use:

```
{"label":"ca-duplicate","msg":"CA 'newca' was already initialised","args":{"ca":"newca
↪"}}
```

6.8 krillc delete

Deletes a CA in your Krill server. The CA will try (best effort) to request revocation of its current certificates from its parents, and withdraw its objects from its repository.

> **Warning:** This action is irreversible!

Example CLI:

```
$ krillc delete --ca ca
```

Example API:

```
$ krillc delete --ca ca --api
DELETE:
   https://localhost:3000/api/v1/cas/ca
Headers:
   Authorization: Bearer secret
```

The API response is an empty 200 OK response, unless an issue occurred - e.g. the CA is not known:

```
{"label":"ca-unknown","msg":"CA 'unknown' is unknown","args":{"ca":"unknown"}}
```

6.9 krillc list

List the current CAs.

Example CLI:

```
$ krillc list
testbed
ta
```

Example API:

```
$ krillc list --api
GET:
  https://localhost:3000/api/v1/cas
Headers:
  Authorization: Bearer secret
```

Example API response:

```
{
  "cas": [
    {
      "handle": "testbed"
    },
    {
      "handle": "ta"
    }
  ]
}
```

6.10 krillc parents

Manage parents for a CA. You will need to add at least one parent, and a repository (see below), before your CA can request any resource certificate.

The Krill CLI and API have a number of subcommands to manage CA parents:

```
request     Show RFC8183 Publisher Request XML
add         Add a parent to this CA
statuses    Show overview of all parent statuses of a CA
contact     Show contact information for a parent of this CA
remove      Remove an existing parent from this CA
```

6.11 krillc parents request

Before you can add a parent to any CA, you will need to present an **RFC 8183** Publisher Request XML to that parent. Their response XML can then be used to add them as a parent.

For more information on how this is done through the UI see: *Parent Setup*.

Example CLI:

```
$ krillc parents request --ca newca
<child_request xmlns="http://www.hactrn.net/uris/rpki/rpki-setup/" version="1" child_
↪handle="newca">
  <child_bpki_ta>
↪MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyhFRjJENzgwRkNCRkU1QjZBMkExMjA1OUM0MDlDN0N
↪X3Ho5gC7uuKf4LYqbJj+Z9ltr/236/
↪hDYJfWMXZVcEuL+wUble1zhe2NKrgnAkpReVMSdiugoqZ9ICK2Fwkj5jCGc/qHiWOba7T78zfij8OlB/
↪dGlJvkAY8b/XTNKsTrLozi1uVAC8GqDrV5MEgY/NfzUvgA024yxx/
↪rC6QBDEoBjnP7wDFiaZ2lwvL2beVYu6/
↪hVcXQzsVN+ijy7cGdkE6zi0meXJLTHPEpoA88hi3Pi+pIDBIQ3wTcpQIOqAq/
↪SZuh4dbZK7BV8MCAwEAAaNTMFEwDwYDVR0TAQH/BAUwAwEB/
↪zAdBgNVHQ4EFgQU7y14D8v+W2oqEgWcQJx8kndUGVYwHwYDVR0jBBgwFoAU7y14D8v+W2oqEgWcQJx8kndUGVYwDQYJKoZIhvcN
↪gpJtONdgIWV1EqwEzhKKA2EP6tLDF9ejsdMFNYrYr+2hVWaoLsSuarfwfLFSgKDFqR6sh3IjIq0m1z9gdkjBOJsR9JyHFEtsDsF
↪YPMPVBpmG15Z9iKantzC1tck+E1xYW5awvj+YZqGVqyFdPJOZWmaYoS83kWvg4g4IucXTH6wwy23MQ7+0gyoK4wxfXRQmWjlXpI
↪d7srQA4IxZCRGh9B+JdAIOKuXBA0nncmMJLQN8Qpxlz2bxKKAgXBLdoDqjbTDVbXTPM8YLRqc=</child_
↪bpki_ta>
```
(continues on next page)

(continued from previous page)

```
</child_request>
```

The API can be called to return the Publisher Request in XML format if you use the following path scheme:

```
GET:
  https://localhost:3000/api/v1/cas/newca/id/child_request.xml
Headers:
  Authorization: Bearer secret
```

The API also supports a JSON equivalent of the response if the *child_request.json* is requested instead:

```
GET:
  https://localhost:3000/api/v1/cas/newca/id/child_request.json
Headers:
  Authorization: Bearer secret
```

6.12 krillc parents add

Add a parent to a CA. Or update the information for an existing parent.

In order to add a parent to a CA you will need to present the **RFC 8183** Parent Response. You will usually get this response in the standard RFC XML format. The Krill API supports submitting this file in its plain XML form, in which case the *local* name for the parent - i.e. the name that your CA will use for it in the presentation to you - will be derived from the path, or if it is not supplied there from the *parent_handle* in the XML.

The API also supports a JSON format where the parent *local handle* can be explicitly specified. If you use the CLI then it will expect that you provide this local handle, parse a supplied XML file, and then combine both in a JSON body sent to the server:

```
$ krillc parents add --parent my_parent --response ./data/new-ca-parent-response.xml -
↪-api
POST:
  https://localhost:3000/api/v1/cas/ca/parents
Headers:
  content-type: application/json
  Authorization: Bearer secret
Body:
{
  "handle": "my_parent",
  "contact": {
    "type": "rfc6492",
    "tag": null,
    "id_cert":
↪"MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyhFOTBDMjE3MzRDMkMzNzBBOTFBODQ3NUNCNEYwR
↪vVYxq1F1w2yQ/
↪VoTr1dvEHxJ+SDayMcFVktWCObiY8tcPhvWG+OdaX9ckDJhsOEEvdVEogwiGacNs7yXJPbqDBptJtbR8/
↪CauF9OqMqjkB/8xkGmBoY5OI/
↪V2832jkp7LPsbyET0RMQN7fgSpGbewvkaZVxGU3pHh5kT1nzPTXrwjxNMXgpunSEY7zR20vYCvsYYbxnSwFNbSMSL+Jgpa+HWPU
↪qlqlZK+tkppI2LkSBhTV5+n7cGA8ZsCAwEAAaNTMFEwDwYDVR0TAQH/BAUwAwEB/
↪zAdBgNVHQ4EFgQU6Qwhc0wsNwqRqEdctPDnXaTQ8L8wHwYDVR0jBBgwFoAU6Qwhc0wsNwqRqEdctPDnXaTQ8L8wDQYJKoZIhvcN
↪nVmqDlK/1sWZNUXFWP4dt1wLTjDWnceyS8mI7Yx8dH/Fez60m4lp4dD45eeaXfbjP2cWnh3n/
↪PLGE70Nj+G0AnUhUmwiTl0H6Px1xn8fZouhv9MEheaZJA+M4NF77+Nmkp2P3WI4cvIS7Te7R/
↪7XpwSr29lVNtYjmRlrBDXx/bMFSgFL61mrtj/
↪l6G8OB40w+sAwO0XKUj1vUUpfIXc3ISCo0LNT9JSPcgy1SZWfmLb98q4HuvxekhkIPRzW7vlb/
↪NBXGarZmKc+HQjE2aXcIewhen2OoTSNda2jSSuEWZuWzZu0aMCKwFBNHLqs=",
```

(continues on next page)

(continued from previous page)

```
    "parent_handle": "testbed",
    "child_handle": "newca",
    "service_uri": "https://localhost:3000/rfc6492/testbed"
  }
}
```

Note that whichever handle you choose, your CA will use the handles that the parent response included for itself *and* for your CA in its communication with this parent. I.e. you may want to inspect the response and use the same handle for the parent (parent_handle attribute), and do not be surprised or alarmed if the parent refers to your ca (child_handle attribute) by some seemingly random name. Some parents do this to ensure uniqueness.

In case you have multiple parents you may want to refer to them by names that make sense in your context, or to avoid name collisions in case they all like to go by the same the name.

In order to specify the parent 'handle' on the path it can simply be added as a path parameter in the call. This is primarily intended for XML in which case the path argument will be taken from here. If you submit a JSON body *and* specify a the handle as path parameter, then Krill will return an error in case the handles do not match.

Important: The API path for ADDING a parent is the same as the API path for updating a parent. This means that adding the same parent multiple times is idempotent. If you are unsure about The parents that your CA currently has, then have a look at the *show* subcommand.

6.13 krillc parents statuses

Show the current status between a CA and all of its parents.

> **Warning:** This command will return an empty result if you did not yet configure a repository for the CA. This is because Krill will not even attempt to contact parent CAs until it knows which URIs to use in certificate requests.

Example CLI:

```
$  krillc parents statuses --ca newca
Parent: my_parent
URI: https://localhost:3000/rfc8181/localname/
Status: success
Last contacted: 2021-04-08T11:20:00+00:00
Next contact on or before: 2021-04-08T11:30:00+00:00
Resource Entitlements: asn: AS65000, v4: 10.0.0.0/8, v6: 2001:db8::/32
  resource class: 0
  issuing cert uri: rsync://localhost/repo/ta/0/
↪0BA5C132B94891CB2D3A89EDE12F01ACA4BCD3DC.cer
  received certificate(s):
    published at: rsync://localhost/repo/testbed/0/
↪16B31C92EB116BC60026C50944AD44205DD9ACBD.cer
    resources:    asn: AS65000, v4: 10.0.0.0/8, v6: 2001:db8::/32
    cert PEM:

-----BEGIN CERTIFICATE-----
MIIFYDCCBEigAwIBAgIUN5PzATTKVrjgual4CpJMaggW2EIwDQYJKoZIhvcNAQELBQAwMzExMC8GA1UEAxMoMEJBNUMxMzJCOTQ4(
↪7OHN8TU6crIu1/
↪w1gkf6UCXFrv+poW9EJHnLonMa4ZFLSFsvQACIGUpxIuiQjaSYFltTbb+o2c9KWoKsX0kZqt5zOrgAP8cke8SFGHdqqenPInXK7
↪ngql4OV0bLkbb63J/26c8FZOThZAgMBAAGjggJqMIICZjAPBgNVHRMBAf8EBTADAQH/    (continues on next page)
↪MB0GA1UdDgQWBBQWsxyS6xFrxgAmxQlErUQgXdmsvTAfBgNVHSMEGDAWgBQLpcEyuUiRyy06ie3hLwGspLzT3DAOBgNVHQ8BAf8
↪BA4wDDAKBggrBgEFBQcOAjAsBggrBgEFBQcBBwEB/
↪wQdMBswCgQCAAEwBAMCAAowDQQCAAIwBwMFACABDbgwGgYIKwYBBQUHAQgBAf8ECzAJoAcwBQIDAP8CMA0GCSqGSIb3DQEBCwUA
↪Yja+sYhyg/pG1/
↪ZTvhOLIxGWap8JmqOnYa9XgX8uUlsV8LgJoEH3Gde3txcGtfLO99ugvbnKKGOcPxB8AX5hAhhfdiSnt3V06dEz3HUoTYdUKTV0k
↪K8349vN0QJUZGm3gAUsM5PlnAqbkM7VFIyu8g2Yp9g+M/
```

(continued from previous page)

```
-----END CERTIFICATE-----
```

Note that in case there are any issues, i.e. the status is "failure" then Krill will keep trying to resynchronise the CA with this parent automatically. There is usually no need to trigger this manually before the next planned contact, but you can use *krillc bulk refresh* if you are debugging an issue.

The JSON response returned by the server contains some additional information, in particular about the certificates used by parent CAs to sign the certificates of your CA:


```
{
  "my_parent": {
    "last_exchange": {
      "timestamp": 1617881400,
      "uri": "https://localhost:3000/rfc8181/localname/",
      "result": "Success"
    },
    "next_exchange_before": 1617882000,
    "all_resources": {
      "asn": "AS65000",
      "v4": "10.0.0.0/8",
      "v6": "2001:db8::/32"
    },
    "entitlements": {
      "0": {
        "parent_cert": {
          "uri": "rsync://localhost/repo/ta/0/
↪0BA5C132B94891CB2D3A89EDE12F01ACA4BCD3DC.cer",
          "cert_pem": "-----BEGIN CERTIFICATE-----\
↪nMIIHKDCCBhCgAwIBAgIUAgyEh9bfPbsXmR1LTAPsL045+tYwDQYJKoZIhvcNAQELBQAwggItMYICKTCCAiUGA1UEAxOCAhwzMD
↪q4q9w/gwwsAjoIP+cTm0FSmqhvqsc1GVI4DQ4mspjZ+O7esFqQywmcnU9MphnGq4EJwYKqT417fU8OQj/
↪WbiCfFhnTrVTiz/LdLdDB4+VaypGfDwPuHb8pavj2dysKiGjLcF8zdon7a/
↪xErHqOdetKlbY20TlvVvmLUeVVKfcnDkT8nsu2k+P+5BHBrb6oQoG4IhZ/w5n65m/
↪ozLsq7pfLrsLgFe2b4zTXhu8KdJ/W1vsshM73jkpUdkvKxif6+H4mBrlMnWg7Jo0bRuff/
↪C0dOAWdiPMXUs53Nw3+SBUjRxhXVWdbcHflkje58pcMkGKSBwIDAQABo4ICNjCCAjIwDwYDVR0TAQH/
↪BAUwAwEB/zAdBgNVHQ4EFgQUC6XBMrlIkcstOont4S8BrKS809wwHwYDVR0jBBgwFoAUS9B/
↪WEM89XrPSCIpOOkwBxZdNKQwDgYDVR0PAQH/
↪BAQDAgEGMFkGA1UdHwRSMFAwTqBMoEqGSHJzeW5jOi8vbG9jYWxob3N0L3JlcG8vdGEvMC80QkQwN0Y1ODQzM0NGNTdBQ0Y0OD
↪wQOMAwwCgYIKwYBBQUHDgIwJwYIKwYBBQUHAQcBAf8EGDAWMAkEAgABMAMDAQAwCQQCAAIwAwMBADAhBggrBgEFBQcBCAEB/
↪wQSMBCgDjAMMAoCAQACBQD/////
↪MA0GCSqGSIb3DQEBCwUAA4IBAQA3rQv0h6x5zX6iGfUZsH0wFSbQQrZgWoql8PsHANokm+Kaxeq3waemrp1/
↪LCzdsMF4+74m6ijDmdbDbHlPyiQwpu3L1vZafj4eBPMdI7xFYgEgabddAGR60b272BgVIO6yND3B6UMeT56NzcCOtOcPtjlgucU
↪yLU1zu3TmDP65+7zaIJebUxOpJ9/4HSG7HsKEU9NHXr414vknGUr8XXiQ0/
↪7f8DrpecGEK2fKu87kBYlewj4zNxJOeQ4heQ4/hJtEeS6dLKz+/VwaUbudlN9/c5QF5ow2bAsNM//
↪ieEWWRL+B0Srr9uNr\n-----END CERTIFICATE-----\n"
        },
        "received": [
          {
            "uri": "rsync://localhost/repo/testbed/0/
↪16B31C92EB116BC60026C50944AD44205DD9ACBD.cer",
            "resources": {
              "asn": "AS65000",
              "v4": "10.0.0.0/8",
              "v6": "2001:db8::/32"
            },
            "cert_pem": "-----BEGIN CERTIFICATE-----\
↪nMIIFYDCCBEigAwIBAgIUN5PzATTKVrjgual4CpJMaggW2EIwDQYJKoZIhvcNAQELBQAwMzExMC8GA1UEAxMoMEJBNUMxMzJCO
↪7OHN8TU6crIu1/
```


(continues on next page)


```
↪w1gkf6UCXFrv+poW9EJHnLonMa4ZFLSFsvQACIGUpxIuiQjaSYFltTbb+o2c9KWoKsX0kZ[illegible]FGHdqqenPInXK
↪ngql4OV0bLkbb63J/26c8FZOThZAgMBAAGjggJqMIICZjAPBgNVHRMBAf8EBTADAQH/
↪MB0GA1UdDgQWBBQWsxyS6xFrxgAmxQlErUQgXdmsvTAfBgNVHSMEGDAWgBQLpcEyuUiRyy06ie3hLwGspLzT3DAOBgNVHQ8BAf8
↪BA4wDDAK[illegible]sBggrBgEFBQcBBwEB/
↪wQdMBswCgQCAAEwBAMCAAowDQQCAAIwBwMFACABDbgwGgYIKwYBBQUHAQgBAf8ECzAJoAcwBQIDAP3oMA0GCSqGSIb3DQEBCwUA
↪Yja+sYhyg/pG1/
↪ZTvhOLIxGWap8JmqOnYa9XgX8uUlsV8LgJoEH3Gde3txcGtfLO99ugvbnKKGOcPxB8AX5hAhhfdiSnt3V06dEz3HUoTYdUKTV0k
```

(continued from previous page)

```
          }
        ]
      }
    }
  }
}
```

Example API:

```
$ krillc parents statuses --ca newca --api
GET:
  https://localhost:3000/api/v1/cas/newca/parents
Headers:
  Authorization: Bearer secret
```

6.14 krillc parents contact

Show contact information for a parent of this CA.

This can be useful for verifying that the parent contact information matches the **RFC 8183** Parent Response that is expected for the given parent.

The API returns the response in JSON format, but this is converted to XML by the CLI when the default text format is used.

```
$ krillc parents contact --ca newca --parent my_parent
```

Here we will show the JSON output:

```
{
  "type": "rfc6492",
  "tag": null,
  "id_cert":
↪"MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyhFOTBDMjE3MzRDMkMzNzBBOTFBODQ3NUNCNEYwR
↪vVYxq1F1w2yQ/
↪VoTr1dvEHxJ+SDayMcFVktWCObiY8tcPhvWG+OdaX9ckDJhsOEEvdVEogwiGacNs7yXJPbqDBptJtbR8/
↪CauF9OqMqjkB/8xkGmBoY5OI/
↪V2832jkp7LPsbyET0RMQN7fgSpGbewvkaZVxGU3pHh5kT1nzPTXrwjxNMXgpunSEY7zR20vYCvsYYbxnSwFNbSMSL+Jgpa+HWPU
↪qlqlZK+tkppI2LkSBhTV5+n7cGA8ZsCAwEAAaNTMFEwDwYDVR0TAQH/BAUwAwEB/
↪zAdBgNVHQ4EFgQU6Qwhc0wsNwqRqEdctPDnXaTQ8L8wHwYDVR0jBBgwFoAU6Qwhc0wsNwqRqEdctPDnXaTQ8L8wDQYJKoZIhvcN
↪nVmqDlK/1sWZNUXFWP4dt1wLTjDWnceyS8mI7Yx8dH/Fez60m4lp4dD45eeaXfbjP2cWnh3n/
↪PLGE70Nj+G0AnUhUmwiTl0H6Px1xn8fZouhv9MEheaZJA+M4NF77+Nmkp2P3WI4cvIS7Te7R/
↪7XpwSr29lVNtYjmRlrBDXx/bMFSgFL61mrtj/
↪l6G8OB40w+sAwO0XKUj1vUUpfIXc3ISCo0LNT9JSPcgy1SZWfmLb98q4HuvxekhkIPRzW7vlb/
↪NBXGarZmKc+HQjE2aXcIewhen2OoTSNda2jSSuEWZuWzZu0aMCKwFBNHLqs=",
  "parent_handle": "testbed",
  "child_handle": "newca",
  "service_uri": "https://localhost:3000/rfc6492/testbed"
}
```

Example API:

```
$ krillc parents contact --ca newca --parent my_parent --api
GET:
```

(continues on next page)

(continued from previous page)

```
  https://localhost:3000/api/v1/cas/newca/parents/my_parent
Headers:
  Authorization: Bearer secret
```

6.15 krillc parents remove

Remove an existing parent from this CA.

The CA will do a best effort attempt to request revocation of any certificate received under the parent - meaning that if the parent cannot be reached the operation just continues without error. After all a parent may well be removed *because* it is no longer reachable. Furthermore any RPKI published under those certificate(s) will be withdrawn.

Note that although revocations are requested the parent may not be aware that they have been removed. You may want to notify them through different channels. The RPKI provisioning protocol **RFC 6492** does not have verbs by which a child CA can ask the parent to be removed completely.

Example CLI:

```
$ krillc parents remove --ca newca --parent my_parent
```

Example API:

```
$ krillc parents remove --ca newca --parent my_parent --api
DELETE:
  https://localhost:3000/api/v1/cas/newca/parents/my_parent
Headers:
  Authorization: Bearer secret
```

6.16 krillc repo

Manage the repository where a CA will publish its objects. There are a number of subcommands provided for this:

```
USAGE:
    krillc repo [SUBCOMMAND]

SUBCOMMANDS:
    request    Show RFC8183 Publisher Request
    configure     Configure which repository this CA uses
    show       Show current repo config
    status      Show current repo status
```

6.17 krillc repo request

Show the RFC 8183 Publisher Request XML for a CA. You will need to hand this over to your repository so that they can add your CA.

Example CLI:

```
$ krillc repo request --ca newca
<publisher_request xmlns="http://www.hactrn.net/uris/rpki/rpki-setup/" version="1"␣
↪publisher_handle="newca">
  <publisher_bpki_ta>
↪MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyhFRjJENzgwRkNCRkU1QjZBMkExMjA1OUM0MDlDN0M
↪X3Ho5gC7uuKf4LYqbJj+Z9ltr/236/
↪hDYJfWMXZVcEuL+wUble1zhe2NKrgnAkpReVMSdiugoqZ9ICK2Fwkj5jCGc/qHiWOba7T78zfij8OlB/
↪dGlJvkAY8b/XTNKsTrLozi1uVAC8GqDrV5MEgY/NfzUvgA024yxx/
↪rC6QBDEoBjnP7wDFiaZ2lwvL2beVYu6/
↪hVcXQzsVN+ijy7cGdkE6zi0meXJLTHPEpoA88hi3Pi+pIDBIQ3wTcpQIOqAq/
↪SZuh4dbZK7BV8MCAwEAAaNTMFEwDwYDVR0TAQH/BAUwAwEB/
↪zAdBgNVHQ4EFgQU7y14D8v+W2oqEgWcQJx8kndUGVYwHwYDVR0jBBgwFoAU7y14D8v+W2oqEgWcQJx8kndUGVYwDQYJKoZIhvcN
↪gpJtONdgIWV1EqwEzhKKA2EP6tLDF9ejsdMFNYrYr+2hVWaoLsSuarfwfLFSgKDFqR6sh3ljYq6mIz9gdkjBOJsR9JyHFEtsDs
↪YPMPVBpmG15Z9iKantzC1tck+E1xYW5awvj+YZqGVqyFdPJOZWmaYoS83kWvg4g4IucXTH6wwy23MQ7+0gyoK4wxfXRQmWjlXp
↪d7srQA4IxZCRGh9B+JdAIOKuXBA0nncmMJLQN8Qpxlz2bxKKAgXBLdoDqjbTDVbXTPM8YLRgc=</
↪publisher_bpki_ta>
</publisher_request>
```

The CLI will present the Publisher Request in its RFC XML format by default. The API supports both the XML and an equivalent JSON format dependent on the file extension used in the request URI:

XML:

```
GET:
  https://localhost:3000/api/v1/cas/newca/id/publisher_request.xml
Headers:
  Authorization: Bearer secret
```

JSON:

```
GET:
  https://localhost:3000/api/v1/cas/newca/id/publisher_request.json
Headers:
  Authorization: Bearer secret
```

6.18 krillc repo configure

This is used to configure the repository used by a CA.

Your CA needs a repository configuration before it will request any certificates from parents. You can chose to configure a repository first and then add the first parent to your CA, or vice versa. The order does not matter, but both are needed for your CA to function.

You can use the CLI to configure a repository by submitting the RFC 8183 Repository Response XML to your CA. Before committing the configuration Krill checks whether the Publication Server can be reached and responds to a query sent by your CA. If this fails, then an error is reported and the configuration is aborted. You can try again when you think the issue has been resolved.

Example CLI:

```
$ krillc repo configure --ca newca --response ./data/new-ca-repository-response.xml
```

The API will accept the plain **RFC 8183** Repository Response XML if it's posted to the API path for the CA in question, but the CLI will post the XML formatted as its JSON equivalent:

Example API:

```
$ krillc repo configure --ca newca --response ./data/new-ca-repository-response.xml --
↪api
POST:
  https://localhost:3000/api/v1/cas/newca/repo
Headers:
  content-type: application/json
  Authorization: Bearer secret
Body:
{
  "repository_response": {
    "tag": null,
    "publisher_handle": "localname",
    "id_cert":
↪"MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyg4OEJBMzA2QkMzMUVFRkU3NzRDNzYzRUY1N0VBNU
↪c7x5sy3JbmUWaQHtkl6N9l8vcRlQQfhk0vwlVCHcQQrcMViJ5GmGtEjo7+Uf9e0TDA+rrkdqOkpOLcGRKjs1SZNqCRktubQU7No
↪gYAnYssX26kObXan0fD9rgv4aWK0Xzp5hwz1ECAwEAAaNTMFEwDwYDVR0TAQH/BAUwAwEB/
↪zAdBgNVHQ4EFgQUiLowa8Me7+d0x2PvV+pf3HoZ3iUwHwYDVR0jBBgwFoAUiLowa8Me7+d0x2PvV+pf3HoZ3iUwDQYJKoZIhvcN
↪Yz1VkQUTjLn2x7DKwuL9A8+IrYELSth4aCNSgPkhZfDL238MflAxptNRAoIeRGn8l3oSg4AUzBuScErwvBbHWShO66nV0wzVFb+
↪GMiZHI/MwGZpj86Q/8wvyyw2C0b0ddWaoXwDyJjuxja0nHPDHVriJ8/
↪xsOfBk144n1zyP++apQXmXorCy4hs9GPyr+HGeoL6kNydDxdwzJLCqWW7u3wSnxjCJk+hfGq82qNm90ALv5PaOb58fDgWwBwuv
↪",
    "service_uri": "https://localhost:3000/rfc8181/localname/",
    "repo_info": {
      "base_uri": "rsync://localhost/repo/localname/",
      "rpki_notify": "https://localhost:3000/rrdp/notification.xml"
    }
  }
}
```

Important: In Krill 0.9.0 you cannot update the configuration of the repository used by your CA after it has been set.

Normally there should be no need to update this configuration after it has been set up initially. However, there may be a use case to do this if for example you chose to run your own Publication Server, but you can now use a Publication Server provided by a third party such as your RIR or NIR.

We have an open issue to address this and we plan to support migrating repositories as soon as possible.

6.19 krillc repo status

This subcommand can be used to verify the status between a CA and its repository. Note that Krill will keep trying to re-sync CAs with their repositories in case of any issues and the response includes an indication of the next planned moment for this. In other words, there should not be a need to trigger this synchronisation manually, but for the impatient, you can use *krillc bulk sync*.

Example CLI:

```
$ krillc repo status --ca newca
URI: https://localhost:3000/rfc8181/localname/
Status: success
Last contacted: 2021-04-08T09:53:27+00:00
Next contact on or before: 2021-04-09T01:53:27+00:00
```

So the CLI text output does NOT include the files which are published. If you want to see these files then you can look at the JSON response instead:


```
{
  "last_exchange": {
    "timestamp": 1617875607,
    "uri": "https://localhost:3000/rfc8181/localname/",
    "result": "Success"
  },
  "next_exchange_before": 1617933207,
  "published": [
    {
      "base64":
↪"MIIJTQYJKoZIhvcNAQcCoIIJPjCCCToCAQMxDTALBglghkgBZQMEAgEwgZsGCyqGSIb3DQEJEAEaoIGLBIGIMIGFAgEBGA8yM
↪NWi2U6R7ffaCCBtgwggbUMIIFvKADAgECAhQfMMPbsoNAMCZM8zHTPf4QMZ06vjANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDE
↪q+Cnm2o/yJFjBfZSYNpE2xo4+nwz4HT6ShXoeSvcQAa6xjeP5z9QwbShfsoGWG0PTert08L5nMpnN4WB/
↪uyfT2l6cHua+M36HNpd23Bx57aaOwRhugMph47pkADkiq5HICZIt/a1/
↪UeFqtsx4D5LMXE6P4XnSPG69K2mQhroGLIIrr6x0LAXlE0uNAkYEcHnmdJwWssG0RoMRGf4myWPDk9pSrkLAgMBAAGjggHiMIIB
↪6TJwtHRc735eady5RAOIwHwYDVR0jBBgwFoAUFrMckusRa8YAJsUJRK1EIF3ZrL0wDgYDVR0PAQH/
↪BAQDAgeAMGAGA1UdHwRZMFcwVaBToFGGT3JzeW5jOi8vbG9jYWxob3N0L3JlcG8vbG9jYWxuYW1lLzAvMTZCMzFDOTJFQjExNk
↪BA4wDDAKBggrBgEFBQcOAjAhBggrBgEFBQcBBwEB/
↪wQSMBAwBgQCAAEFADAGBAIAAgUAMBUGCCsGAQUFBwEIAQH/
↪BAYwBKACBQAwDQYJKoZIhvcNAQELBQADggEBAJQiHZ91d7/
↪a52qM0DyXp7jbkygm2MkT5tc6pp6sxHv6pDfXxAzJS8OtgcFCTDKC57pKvVvw8THE079nbMSxfaA5nP8egedxeuTzrj8iOh9nH
↪X4pWhIWsAvNgiTebYj+Eax97MmRWAkDgxWpDQ+CWQBl2gBstLmBKCBTw6cFlkGrBCLVe+gSDTnHpy4ltza6pD+EawTNrGLBnFn
↪+dgzx/GA2qbRXiBLm2/R4HR7zI/QYy+wWDoaZraCu6dUZEF4WomS99aihEyNp8tzyuEntmmMfw0z/
↪xYt1I7VN1pzc5umEPksSRvILmA3eJO3Khw2xWZzYjYcVyZAo0QbujdExggGqMIIBpgIBA4AU0iBYgw/
↪6TJwtHRc735eady5RAOIwCwYJYIZIAWUDBAIBoGswGgYJKoZIhvcNAQkDMQ0GCyqGSIb3DQEJBTEPFw
↪MgJG2G+mOfNzHsAV6ysGcb89bMa7KTEmTANBgkqhkiG9w0BAQEFAASCAQCHxZ3CeXicDOpmXZ/
↪uhEGtvsuzpepVryk58zBLnSpbKfjnJWwiL0t3PsvlQuKAXgW0Xc5cC4Bbvb8Aysr4W1c0SKjnWz4dPLqgNCzvIVJRToc2xHFd6
↪DF2j3h8eZpEM1Y09kOTQfwkn297UYOv9Hi74iKIzhS3+8FmfSP0UTA207+U7HBQp9SNkK2HjFa3milgV+hJHOPutNsbgvwd5YP
↪qvfTN0hZlGafx5ODIppv+tqJ76zts9wzgVXrpl6tKQ",
      "uri": "rsync://localhost/repo/localname/0/
↪16B31C92EB116BC60026C50944AD44205DD9ACBD.mft"
    },
    {
      "base64":
↪"MIIBrzCBmAIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEygxNkIzMUM5MkVCMTE2QkM2MDAyNkM1MDk0NEFENDQyMDVERD
↪OQx341/
↪wWYrBrEAQ56NE6AVN+r0qjmO2mhNgVNQ1VdCLjo67ilQufmxGhtUQxBS625f1hr69cYw1l15wHDP4SFpXO96ysTxBhxpLGL215n
↪",
      "uri": "rsync://localhost/repo/localname/0/
↪16B31C92EB116BC60026C50944AD44205DD9ACBD.crl"
    }
  ]
}
```


Example API:

```
$ krillc repo status --ca newca --api
GET:
```

(continues on next page)

(continued from previous page)

```
  https://localhost:3000/api/v1/cas/newca/repo/status
Headers:
  Authorization: Bearer secret
```

6.20 krillc repo show

Show the repository configuration for your CA.

Example CLI:

```
$ krillc repo show --ca newca
Repository Details:
  service uri: https://localhost:3000/rfc8181/localname/
  base_uri:    rsync://localhost/repo/localname/
  rpki_notify: https://localhost:3000/rrdp/notification.xml
```

Example API:

```
$ krillc repo show --ca newca --api
GET:
  https://localhost:3000/api/v1/cas/newca/repo
Headers:
  Authorization: Bearer secret
```

6.21 krillc show

Shows lots of details of a CA. Note, we may still extend the JSON response in future but we will aim to add new information only.

Example CLI:

```
$ krillc show --ca newca
Name:     newca

Base uri: rsync://localhost/repo/localname/
RRDP uri: https://localhost:3000/rrdp/notification.xml

ID cert PEM:
-----BEGIN CERTIFICATE-----
MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyhFRjJE
NzgwRkNCRkU1QjZBMkExMjA1OUM0MDlDN0M5Mjc3NTQxOTU2MB4XDTIxMDQwNzE0
MzUxNFoXDTM2MDQwNzE0NDAxNFowMzExMC8GA1UEAxMoRUYyRDc4MEZDQkZFNUI2
QTJBMTIwNTlDNDA5QzdDOTI3NzU0MTk1NjCCASIwDQYJKoZIhvcNAQEBBQADggEP
ADCCAQoCggEBANuBsEO4C9n7PlYcDT0PTeZntR5l778lZQDsgxiB7ofLrg8lKcf8
ugFiYI4vRqR+gDMHhR3t/X3Ho5gC7uuKf4LYqbJj+Z9ltr/236/hDYJfWMXZVcEu
L+wUble1zhe2NKrgnAkpReVMSdiugoqZ9ICK2Fwkj5jCGc/qHiWOba7T78zfij8O
lB/dGlJvkAY8b/XTNKsTrLozi1uVAC8GqDrV5MEgY/NfzUvgA024yxx/rC6QBDEo
BjnP7wDFiaZ2lwvL2beVYu6/hVcXQzsVN+ijy7cGdkE6zi0meXJLTHPEpoA88hi3
Pi+pIDBIQ3wTcpQIOqAq/SZuh4dbZK7BV8MCAwEAAaNTMFEwDwYDVR0TAQH/BAUw
AwEB/zAdBgNVHQ4EFgQU7y14D8v+W2oqEgWcQJx8kndUGVYwHwYDVR0jBBgwFoAU
```

(continues on next page)

(continued from previous page)

```
7y14D8v+W2oqEgWcQJx8kndUGVYwDQYJKoZIhvcNAQELBQADggEBAArqsa/gpJtO
NdgIWV1EqwEzhKKA2EP6tLDF9ejsdMFNYrYr+2hVWaoLsSuarfwfLFSgKDFqR6sh
3ljYq6mIz9gdkjBOJsR9JyHFEtsDsRpf8Hs1WlbIb8bWb73Cp/YPMPVBpmG15Z9i
KantzC1tck+E1xYW5awvj+YZqGVqyFdPJOZWmaYoS83kWvg4g4IucXTH6wwy23MQ
7+0gyoK4wxfXRQmWjlXpLueCOsJo7ZXopsDAmXHLoFKZVEXn1ocQNc911521BEQ6
t/d7srQA4IxZCRGh9B+JdAIOKuXBA0nncmMJLQN8Qpxlz2bxKKAgXBLdoDqjbTDV
bXTPM8YLRgc=
-----END CERTIFICATE-----

Hash: 992ac17d85fef11d8be4aa37806586ce68b61fe9cf65c0965928dbce0c398a99

Total resources:
    ASNs: AS65000
    IPv4: 10.0.0.0/8
    IPv6: 2001:db8::/32

Parents:
Handle: my_parent Kind: RFC 6492 Parent

Resource Class: 0
Parent: my_parent
State: active    Resources:
    ASNs: AS65000
    IPv4: 10.0.0.0/8
    IPv6: 2001:db8::/32

Children:
<none>
```

Example JSON response of the API:


```
{
  "handle": "newca",
  "id_cert": {
    "pem": "-----BEGIN CERTIFICATE-----\
↪nMIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyhFRjJE\
↪nNzgwRkNCRkU1QjZBMkExMjA1OUM0MDlDN0M5Mjc3NTQxOTU2MB4XDTIxMDQwNzE0\
↪nMzUxNFoXDTM2MDQwNzE0NDAxNFowMzExMC8GA1UEAxMoRUYyRDc4MEZDQkZFNUI2\
↪nQTJBMTIwNTlDNDA5QzdDOTI3NzU0MTk1NjCCASIwDQYJKoZIhvcNAQEBBQADggEP\
↪nADCCAQoCggEBANuBsEO4C9n7PlYcDT0PTeZntR5l778lZQDsgxiB7ofLrg8lKcf8\
↪nugFiYI4vRqR+gDMHhR3t/X3Ho5gC7uuKf4LYqbJj+Z9ltr/236/hDYJfWMXZVcEu\
↪nL+wUble1zhe2NKrgnAkpReVMSdiugoqZ9ICK2Fwkj5jCGc/qHiWOba7T78zfij8O\nlB/dGlJvkAY8b/
↪XTNKsTrLozi1uVAC8GqDrV5MEgY/NfzUvgA024yxx/rC6QBDEo\nBjnP7wDFiaZ2lwvL2beVYu6/
↪hVcXQzsVN+ijy7cGdkE6zi0meXJLTHPEpoA88hi3\nPi+pIDBIQ3wTcpQIOqAq/
↪SZuh4dbZK7BV8MCAwEAAaNTMFEwDwYDVR0TAQH/BAUw\nAwEB/
↪zAdBgNVHQ4EFgQU7y14D8v+W2oqEgWcQJx8kndUGVYwHwYDVR0jBBgwFoAU\
↪n7y14D8v+W2oqEgWcQJx8kndUGVYwDQYJKoZIhvcNAQELBQADggEBAArqsa/gpJtO\
↪nNdgIWV1EqwEzhKKA2EP6tLDF9ejsdMFNYrYr+2hVWaoLsSuarfwfLFSgKDFqR6sh\
↪n3ljYq6mIz9gdkjBOJsR9JyHFEtsDsRpf8Hs1WlbIb8bWb73Cp/YPMPVBpmG15Z9i\
↪nKantzC1tck+E1xYW5awvj+YZqGVqyFdPJOZWmaYoS83kWvg4g4IucXTH6wwy23MQ\
↪n7+0gyoK4wxfXRQmWjlXpLueCOsJo7ZXopsDAmXHLoFKZVEXn1ocQNc911521BEQ6\nt/
↪d7srQA4IxZCRGh9B+JdAIOKuXBA0nncmMJLQN8Qpxlz2bxKKAgXBLdoDqjbTDV\nbXTPM8YLRgc=\n-----
↪END CERTIFICATE-----\n",
    "hash": "992ac17d85fef11d8be4aa37806586ce68b61fe9cf65c0965928dbce0c398a99"
  },
  "repo_info": {
    "base_uri": "rsync://localhost/repo/localname/",
```


(continues on next page)

(continued from previous page)


```
    "rpki_notify": "https://localhost:3000/rrdp/notification.xml"
  },
  "parents": [
    {
      "handle": "my_parent",
      "kind": "rfc6492"
    }
  ],
  "resources": {
    "asn": "AS65000",
    "v4": "10.0.0.0/8",
    "v6": "2001:db8::/32"
  },
  "resource_classes": {
    "0": {
      "name_space": "0",
      "parent_handle": "my_parent",
      "keys": {
        "active": {
          "active_key": {
            "key_id": "16B31C92EB116BC60026C50944AD44205DD9ACBD",
            "incoming_cert": {
              "cert":
↪"MIIFYDCCBEigAwIBAgIUN5PzATTKVrjgual4CpJMaggW2EIwDQYJKoZIhvcNAQELBQAwMzExMC8GA1UEAxMoMEJBNUMxMzJCO
↪7OHN8TU6crIu1/
↪w1gkf6UCXFrv+poW9EJHnLonMa4ZFLSFsvQACIGUpxIuiQjaSYFltTbb+o2c9KWoKsX0kZqt5zOrgAP8cke8SFGHdqqenPInXK
↪ngql4OV0bLkbb63J/26c8FZOThZAgMBAAGjggJqMIICZjAPBgNVHRMBAf8EBTADAQH/
↪MB0GA1UdDgQWBBQWsxyS6xFrxgAmxQ1ErUQgXdmsvTAfBgNVHSMEGDAWgBQLpcEyuUiRyy06ie3hLwGspLzT3DAOBgNVHQ8BAf8
↪BA4wDDAKBggrBgEFBQcOAjAsBggrBgEFBQcBBwEB/
↪wQdMBswCgQCAAEwBAMCAAowDQQCAAIwBwMFACABDbgwGgYIKwYBBQUHAQgBAf8ECzAJoAcwBQIDAP3oMA0GCSqGSIb3DQEBCwUA
↪Yja+sYhyg/pG1/
↪ZTvhOLIxGWap8JmqOnYa9XgX8uUlsV8LgJoEH3Gde3txcGtfLO99ugvbnKKGOcPxB8AX5hAhhfdiSnt3V06dEz3HUoTYdUKTV0k
↪K8349vN0QJUZGm3gAUsM5PlnAqbkM7VFIyu8g2Yp9g+M/
↪iwaHar8CqABKxLBThYgqrPLLv6CsZD3mjk5BkXVZh6R9dBcR7sPbSfGBWPWCv8SwLknyQDOvsWTho1Ga6AibjUQp
↪",
              "uri": "rsync://localhost/repo/testbed/0/
↪16B31C92EB116BC60026C50944AD44205DD9ACBD.cer",
              "resources": {
                "asn": "AS65000",
                "v4": "10.0.0.0/8",
                "v6": "2001:db8::/32"
              }
            },
            "request": null
          }
        }
      }
    }
  },
  "children": []
}
```


Example API call:

```
$ krillc show --ca newca --api
GET:
  https://localhost:3000/api/v1/cas/newca
```

(continues on next page)

(continued from previous page)

```
Headers:
  Authorization: Bearer secret
```

6.22 krillc issues

Show issues for CAs. The response will be empty unless there are actual current issues.

Example CLI:

```
$ krillc issues --ca newca
no issues found
```

Example JSON response with issues:

```
{
  "repo_issue": {
    "label": "sys-http-client",
    "msg": "HTTP client error: Access Forbidden",
    "args": {
      "cause": "Access Forbidden"
    }
  },
  "parent_issues": [
    {
      "parent": "parent",
      "issue": {
        "label": "rfc6492-invalid-signature",
        "msg": "Invalidly signed RFC 6492 CMS",
        "args": {}
      }
    }
  ]
}
```

Example API call:

```
$ krillc issues --ca newca --api
GET:
  https://localhost:3000/api/v1/cas/newca/issues
Headers:
  Authorization: Bearer secret
```

6.23 krillc history

Show the history of a CA. Using this command you can show the history of all the things that happened to your CA.

There are two subcommands for this:

```
USAGE:
    krillc history [SUBCOMMAND]

SUBCOMMANDS:
    commands    Show the commands sent to a CA
    details     Show details for a command in the history of a CA
```

6.24 krillc history commands

With this subcommand you can look at an overview of all commands that were sent to a CA.

Example CLI:

```
$ krillc history commands --ca newca
time::command::key::success
2021-04-07T15:25:01Z::Add parent 'my_parent' as 'RFC 6492 Parent' ::command--
↪1617809101--1--cmd-ca-parent-add::OK
2021-04-08T09:53:23Z::Update repo to server at: https://localhost:3000/rfc8181/
↪localname/ ::command--1617875603--2--cmd-ca-repo-update::OK
2021-04-08T09:53:24Z::Update entitlements under parent 'my_parent': 0 => asn: AS65000,
↪ v4: 10.0.0.0/8, v6: 2001:db8::/32  ::command--1617875604--3--cmd-ca-parent-
↪entitlements::OK
2021-04-08T09:53:25Z::Update received cert in RC '0', with resources 'asn: 1 blocks,␣
↪v4: 1 blocks, v6: 1 blocks' ::command--1617875605--4--cmd-ca-rcn-receive::OK
```

The JSON response includes some data which we do not (yet) show in the text output - e.g. the name of the user who sent a command. This will become more relevant in future as people start using the multi-user feature of the Krill UI:


```
{
  "offset": 0,
  "total": 4,
  "commands": [
    {
      "key": "command--1617809101--1--cmd-ca-parent-add",
      "actor": "master-token",
      "timestamp": 1617809101616,
      "handle": "newca",
      "version": 1,
      "sequence": 1,
      "summary": {
        "msg": "Add parent 'my_parent' as 'RFC 6492 Parent'",
        "label": "cmd-ca-parent-add",
        "args": {
          "parent": "my_parent",
          "parent_contact": "RFC 6492 Parent"
        }
      },
      "effect": {
        "result": "success",
```


(continues on next page)

(continued from previous page)


```
        "events": [
          1
        ]
      }
    },
    {
      "key": "command--1617875603--2--cmd-ca-repo-update",
      "actor": "master-token",
      "timestamp": 1617875603613,
      "handle": "newca",
      "version": 2,
      "sequence": 2,
      "summary": {
        "msg": "Update repo to server at: https://localhost:3000/rfc8181/localname/",
        "label": "cmd-ca-repo-update",
        "args": {
          "service_uri": "https://localhost:3000/rfc8181/localname/"
        }
      },
      "effect": {
        "result": "success",
        "events": [
          2
        ]
      }
    },
    {
      "key": "command--1617875604--3--cmd-ca-parent-entitlements",
      "actor": "krill",
      "timestamp": 1617875604550,
      "handle": "newca",
      "version": 3,
      "sequence": 3,
      "summary": {
        "msg": "Update entitlements under parent 'my_parent': 0 => asn: AS65000, v4:␣
↪10.0.0.0/8, v6: 2001:db8::/32 ",
        "label": "cmd-ca-parent-entitlements",
        "args": {
          "parent": "my_parent"
        }
      },
      "effect": {
        "result": "success",
        "events": [
          3,
          4
        ]
      }
    },
    {
      "key": "command--1617875605--4--cmd-ca-rcn-receive",
      "actor": "krill",
      "timestamp": 1617875605662,
      "handle": "newca",
      "version": 5,
      "sequence": 4,
      "summary": {
```


(continues on next page)

(continued from previous page)

```
        "msg": "Update received cert in RC '0', with resources 'asn: 1 blocks, v4: 1␣
↪blocks, v6: 1 blocks'",
        "label": "cmd-ca-rcn-receive",
        "args": {
          "asn_blocks": "1",
          "class_name": "0",
          "ipv4_blocks": "1",
          "ipv6_blocks": "1",
          "resources": "asn: AS65000, v4: 10.0.0.0/8, v6: 2001:db8::/32"
        }
      },
      "effect": {
        "result": "success",
        "events": [
          5
        ]
      }
    }
  ]
}
```

The CLI and API support pagination:

```
--after <<RFC 3339 DateTime>>      Show commands issued after date/time in RFC 3339␣
↪format, e.g. 2020-04-
                                   09T19:37:02Z
--before <<RFC 3339 DateTime>>     Show commands issued after date/time in RFC 3339␣
↪format, e.g. 2020-04-
                                   09T19:37:02Z
--offset <<number>>                Number of results to skip
--rows <<number>>                  Number of rows (max 250)
```

And these values are converted to path parameters in the API call:

```
$ krillc history commands --ca newca --after 2020-12-01T00:00:00Z --before 2021-04-
↪09T00:00:00Z --rows 2 --offset 1 --api
GET:
  https://localhost:3000/api/v1/cas/newca/history/commands/2/1/1606780800/1617926400
Headers:
  Authorization: Bearer secret
```

6.25 krillc history details

Show details for a specific historic CA command. This subcommand expects the command key as reported by *krillc history commands*.

The text output of the CLI will show a summary of the command details, and the state changes in the CA (called events) that followed:

```
$ krillc history details --ca newca --key command--1617875604--3--cmd-ca-parent-
↪entitlements
Time:   2021-04-08T09:53:24Z
Action: Update entitlements under parent 'my_parent': 0 => asn: AS65000, v4: 10.0.0.0/
↪8, v6: 2001:db8::/32
```

(continues on next page)

(continued from previous page)

```
Changes:
  added resource class with name '0'
  requested certificate for key (hash) '16B31C92EB116BC60026C50944AD44205DD9ACBD'␣
↪under resource class '0'
```

If you want to see the full details, then have a look at the JSON response instead:


```
{
  "command": {
    "actor": "krill",
    "time": "2021-04-08T09:53:24.550017Z",
    "handle": "newca",
    "version": 3,
    "sequence": 3,
    "details": {
      "type": "update_resource_entitlements",
      "parent": "my_parent",
      "entitlements": [
        {
          "resource_class_name": "0",
          "resources": {
            "asn": "AS65000",
            "v4": "10.0.0.0/8",
            "v6": "2001:db8::/32"
          }
        }
      ]
    },
    "effect": {
      "result": "success",
      "events": [
        3,
        4
      ]
    }
  },
  "result": {
    "Events": [
      {
        "id": "newca",
        "version": 3,
        "details": {
          "type": "resource_class_added",
          "resource_class_name": "0",
          "parent": "my_parent",
          "parent_resource_class_name": "0",
          "pending_key": "16B31C92EB116BC60026C50944AD44205DD9ACBD"
        }
      },
      {
        "id": "newca",
        "version": 4,
        "details": {
          "type": "certificate_requested",
          "resource_class_name": "0",
          "req": {
            "class_name": "0",
```


(continues on next page)

(continued from previous page)

```
            "limit": {
              "asn": "none",
              "v4": "none",
              "v6": "none"
            },
            "csr":
↪"MIIDjzCCAncCAQAwMzExMC8GA1UEAxMoMTZCMzFDOTJFQjExNkJDNjAwMjZDNTA5NDRBRDQ0MjA1REQ5QUNCRDCCASIwDQYJK
↪6EvY4Uo0ICel3vWE5cwE1db/s4c3xNTpysi7X/DWCR/pQJcWu/
↪6mhb0QkecuicxrhkUtIWy9AAIgZSnEi6JCNpJgWW1Ntv6jZz0pagqxfSRmq3nM6uAA/
↪xyR7xIUYd2qp6c8idcpODKyz2QKz2kW0yToEhroqNG+oVVkh/
↪rEbnfK0ncmqwaO8SjyqgdjGS+Qy1uOGOWZbT3uLoN4LXbjVfdIbrgNIyTUI2/
↪XG3kxJe5svNY4P7aUsh3+eCqXg5XRsuRtvrcn/
↪bpzwVk5OFkCAwEAAaCCARUwggERBgkqhkiG9w0BCQ4xggECMIH/MA8GA1UdEwEB/
↪wQFMAMBAf8wDgYDVR0PAQH/
↪BAQDAgEGMIHbBggrBgEFBQcBCwEBAASByzCByDAvBggrBgEFBQcwBYYjcnN5bmM6Ly9sb2NhbGhvc3QvcmVwby9sb2NhbG5hbWU
↪hxjAia3vtLrYtET1InOF/5UtRClDX5EWl34JRCXEIkDgWWbCVmxQyTw0VfqKImT/JqzC/
↪NXrWMJBVJ27JgkHH5TITHGgfIjDRS19+JOFdiCBlQWgU3V5zfMGlB0263xRteX7A1kLedLuvt51DgNMwyWFgp/
↪PkJKUCTEYi27j6DOF5J8jZ7JD5lMBs7gOGAiUJSzCBY7XfjEeVmePRLJ8hB0Wa/
↪n3h+ni6UTOF6itKPmHqddxpiEb8ij987gCTjuZQisi9j+JKoPqzXon2vOx+GJjo4Sb++HD0buatiEmj5SvUmV8gl0F/
↪msh4F4a5YG8r"
          },
          "ki": "16B31C92EB116BC60026C50944AD44205DD9ACBD"
        }
      }
    ]
  }
}
```

Example API call:

```
$ krillc history details --ca newca --key command--1617875604--3--cmd-ca-parent-
↪entitlements --api
GET:
  https://localhost:3000/api/v1/cas/newca/history/details/command--1617875604--3--cmd-
↪ca-parent-entitlements
Headers:
  Authorization: Bearer secret
```

6.26 krillc roas

Manage ROAs for your CA.

Krill lets users create Route Origin Authorisations (ROAs), the signed objects that state which Autonomous System (AS) is authorised to originate one of your prefixes, along with the maximum prefix length it may have.

Important: Krill CAs let operators configure which authorizations they want to have on ROA **objects**. But it's Krill that will figure out which objects to create for this. I.e. users just configure their intent to authorise an ASN to originate a prefix, but they do not need to worry about things like the actual ROA encoding, before and after times, object renewals, publishing, and under which parent the ROA is to be created - if there are multiple. However, we will refer to these authorizations as ROAs, because for all intent and purposes this difference is an implementation detail that Krill, by design, abstracts away from the operator.

```
USAGE:
    krillc roas [SUBCOMMAND]

SUBCOMMANDS:
    list       Show current authorizations
    update     Update authorizations
    bgp        Show current authorizations in relation to known announcements
```

6.27 krillc roas list

Show current authorizations.

```
USAGE:
    krillc roas list [FLAGS] [OPTIONS]

OPTIONS:
    -c, --ca <name>          The name of the CA you wish to control. Or set env: KRILL_
↪CLI_MY_CA
```

Example:

You can list ROAs in the following way:

```
$ krillc roas list
192.0.2.0/24 => 64496
2001:db8::/32-48 => 64496
```

6.28 krillc roas update

Update ROAs.

The CLI supports adding or removing individual ROAs as well as submitting a file with a delta of additions and removals as an atomic delta. In terms of the API these options will call the same API end-point and always submit a JSON body with a delta.

- Add a single ROA

Example CLI usage to add a ROA:

```
$ krillc roas update --ca newca --add "192.168.0.0/16 => 64496"
```

This will submit the following JSON to the API:

```
$ krillc roas update --add "192.168.0.0/16 => 64496" --api
POST:
  https://localhost:3000/api/v1/cas/ca/routes
Headers:
  content-type: application/json
  Authorization: Bearer secret
Body:
{
  "added": [
    {
      "asn": 64496,
```

(continues on next page)

(continued from previous page)

```
      "prefix": "192.168.0.0/16"
    }
  ],
  "removed": []
}
```

- Remove a single ROA

Example CLI usage to remove a ROA:

```
$ krillc roas update --ca newca --remove "192.168.0.0/16 => 64496"
```

This will submit the following JSON to the API:

```
$ krillc roas update --ca newca --remove "192.168.0.0/16 => 64496" --api
POST:
  https://localhost:3000/api/v1/cas/newca/routes
Headers:
  content-type: application/json
  Authorization: Bearer secret
Body:
{
  "added": [],
  "removed": [
    {
      "asn": 64496,
      "prefix": "192.168.0.0/16"
    }
  ]
}
```

- Update multiple ROAs

You can also update multiple ROAs as a single delta. The CLI can do deltas if you provide it with a file using the following format:

```
# Some comment
  # Indented comment

A: 10.0.0.0/24 => 64496
A: 10.1.0.0/16-20 => 64496   # Add prefix with max length
R: 10.0.3.0/24 => 64496      # Remove existing authorization
```

And then call the CLI with the `--delta` option. The CLI will parse the delta file and submit a JSON body containing multiple changes:

```
krillc roas update --delta ./data/roa-delta.txt --ca newca --api
POST:
 https://localhost:3000/api/v1/cas/newca/routes
Headers:
 content-type: application/json
 Authorization: Bearer secret
Body:
{
 "added": [
   {
     "asn": 64496,
```

(continues on next page)

(continued from previous page)

```
      "prefix": "10.0.0.0/24"
    },
    {
      "asn": 64496,
      "prefix": "10.1.0.0/16",
      "max_length": 20
    }
  ],
  "removed": [
    {
      "asn": 64496,
      "prefix": "10.0.3.0/24"
    }
  ]
}
```

- Errors

You will get an error response if ROA updates cannot be applied. For example adding a duplicate ROA will result in the following error:

```
$ krillc roas update --ca newca --add "192.168.0.0/16 => 64496"
Delta rejected:

Cannot add the following duplicate ROAs:
  192.168.0.0/16-16 => 64496
```

The returned JSON for an error with with the label "ca-roa-delta-error" has a format similar to the normal error response, but with the addition of a *delta_error* entry with details. There you can expect 4 categories of errors:

duplicates	You are trying to add a ROA that already exists
notheld	You are trying to add a ROA for a prefix you don't hold
unknowns	You are trying to remove a ROA that does not exist
invalid_length	You specified an invalid length/max_length for a prefix

Example:

```
{
  "label": "ca-roa-delta-error",
  "msg": "Delta rejected, see included json",
  "args": {},
  "delta_error": {
    "duplicates": [
      {
        "asn": 1,
        "prefix": "10.0.0.0/20",
        "max_length": 24
      }
    ],
    "notheld": [
      {
        "asn": 1,
        "prefix": "10.128.0.0/9"
      }
    ],
```

(continues on next page)

(continued from previous page)


```
    "unknowns": [
      {
        "asn": 1,
        "prefix": "192.168.0.0/16"
      }
    ],
    "invalid_length": [
      {
        "asn": 1,
        "prefix": "10.0.1.0/25"
      }
    ]
  }
}
```


- Try

With RPKI ROAs you can create RPKI invalids in BGP if for example your prefix is multi homed and you authorise one ASN, but not another. Another cause of invalids might be that you authorise a covering prefix, but not more specific announcements that you do.

To help with this Krill also comes with a "try", or "feeling lucky" feature. Meaning that when `--try` is specified with an update, Krill will check the effect of the update against what it knows about BGP announcements. If the effect has no negative side-effects then it will just be applied, but if it would result in any invalid announcements then an error report will be returned instead:

```
$ krillc roas update --ca newca --add "192.168.0.0/16 => 64496" --try
Unsafe update, please review

Effect would leave the following invalids:

  Announcements from invalid ASNs:
    192.168.0.0/24 => 64497

    192.168.1.0/24 => 64497


  Announcements too specific for their ASNs:

    192.168.0.0/24 => 64496

You may want to consider this alternative:
Authorize these announcements which are currently not covered:
  192.168.0.0/24 => 64496
  192.168.0.0/24 => 64497
  192.168.1.0/24 => 64497
```

Example JSON response:


```
{
  "effect": [
    {
      "asn": 64496,
      "prefix": "192.168.0.0/16",
      "max_length": 16,
      "state": "roa_disallowing",
      "disallows": [
```


(continues on next page)

(continued from previous page)


```
        {
          "asn": 64496,
          "prefix": "192.168.0.0/24"
        },
        {
          "asn": 64497,
          "prefix": "192.168.0.0/24"
        },
        {
          "asn": 64497,
          "prefix": "192.168.1.0/24"
        }
      ]
    },
    {
      "asn": 64496,
      "prefix": "192.168.0.0/24",
      "state": "announcement_invalid_length",
      "disallowed_by": [
        {
          "asn": 64496,
          "prefix": "192.168.0.0/16",
          "max_length": 16
        }
      ]
    },
    {
      "asn": 64497,
      "prefix": "192.168.0.0/24",
      "state": "announcement_invalid_asn",
      "disallowed_by": [
        {
          "asn": 64496,
          "prefix": "192.168.0.0/16",
          "max_length": 16
        }
      ]
    },
    {
      "asn": 64497,
      "prefix": "192.168.1.0/24",
      "state": "announcement_invalid_asn",
      "disallowed_by": [
        {
          "asn": 64496,
          "prefix": "192.168.0.0/16",
          "max_length": 16
        }
      ]
    }
  ],
  "suggestion": {
    "not_found": [
      {
        "asn": 64496,
        "prefix": "192.168.0.0/24"
      },
```


(continues on next page)

(continued from previous page)

```
      {
        "asn": 64497,
        "prefix": "192.168.0.0/24"
      },
      {
        "asn": 64497,
        "prefix": "192.168.1.0/24"
      }
    ]
  }
}
```

The API call for this is the same as when posting a normal ROA delta, except that */try* is appended to the path, e.g.: *POST https://localhost:3000/api/v1/cas/newca/routes/try*

Important: Krill does this analysis based on RIPE RIS BGP information. This information may be outdated, or incomplete. More importantly it may also include erroneous or even malicious announcements that are seen in the global BGP. So **ALWAYS** review the report and suggestions returned by Krill! Note, we plan to support other ways of getting BGP information into Krill in future - e.g. by parsing a local BGP feed or table.

- Dryrun

The `dryrun` option is similar to `try`, except that, well, it doesn't even try to apply a change. It just reports the effects of a change including positive effects.. so, actually, it *is* different:

```
$ krillc roas update --ca newca --add "10.0.0.0/24 => 64496" --dryrun
Authorizations covering announcements seen:

        Definition: 10.0.0.0/24-24 => 64496

                Authorizes:
                10.0.0.0/24 => 64496

Announcements which are valid:

        Announcement: 10.0.0.0/24 => 64496
```

6.29 krillc roas bgp

Important: Krill does BGP analysis based on RIPE RIS BGP information. This information may be outdated, or incomplete. More importantly it may also include erroneous or even malicious announcements that are seen in the global BGP. So **ALWAYS** review the reports and suggestions returned by Krill! Note, we plan to support other ways of getting BGP information into Krill in future - e.g. by parsing a local BGP feed or table.

The ROA vs BGP analysis is used in the `try` and `dryrun` options when applying a ROA delta, but this can also be accessed proactively. For this the CLI has the following subcommands:

```
krillc roas bgp analyze   Show full report of ROAs vs known BGP announcements
krillc roas bgp suggest   Show ROA suggestions based on known BGP announcements
```

Example of the analyze function:

```
$ krillc roas bgp analyze --ca newca
Authorizations covering announcements seen:

        Definition: 192.168.0.0/24-24 => 64496

                Authorizes:
                192.168.0.0/24 => 64496

                Disallows:
                192.168.0.0/24 => 64497

Authorizations disallowing announcements seen. You may want to use AS0 ROAs instead:

        Definition: 192.168.0.0/16-16 => 64496


                Disallows:
                192.168.0.0/24 => 64497
                192.168.1.0/24 => 64497

Announcements which are valid:

        Announcement: 192.168.0.0/24 => 64496

Announcements from an unauthorized ASN:

        Announcement: 192.168.0.0/24 => 64497

                Disallowed by authorization(s):
                192.168.0.0/16-16 => 64496
                192.168.0.0/24-24 => 64496

        Announcement: 192.168.1.0/24 => 64497

                Disallowed by authorization(s):
                192.168.0.0/16-16 => 64496

Announcements which are 'not found' (not covered by any of your authorizations):

        Announcement: 10.0.0.0/21 => 64497
        Announcement: 10.0.0.0/22 => 64496
        Announcement: 10.0.0.0/22 => 64497
        Announcement: 10.0.0.0/24 => 64496
        Announcement: 10.0.2.0/23 => 64496
```

Example output of the “suggest” option:

```
$ krillc roas bgp suggest --ca newca
Remove the following ROAs which only disallow announcements (did you use the wrong␣
↪ASN?), if this is intended you may want to use AS0 instead:
  192.168.0.0/16-16 => 64496

Keep the following authorizations:
  192.168.0.0/24-24 => 64496

Authorize these announcements which are currently not covered:
  10.0.0.0/21 => 64497
```

(continues on next page)

(continued from previous page)

```
  10.0.0.0/22 => 64496
  10.0.0.0/22 => 64497
  10.0.0.0/24 => 64496
  10.0.2.0/23 => 64496

Authorize these announcements which are currently invalid because they are not␣
↪allowed for these ASNs:
  192.168.0.0/24 => 64497
  192.168.1.0/24 => 64497
```

6.30 krillc bulk

Manually trigger refresh/republish/resync for all CAs.

Normally there is no need to use these functions. Krill has background processes that these functions run whenever they are needed. However, they may be useful in cases where the connection between your CA(s) and their remote parents or repository may be broken for example, and you want to debug the issue.

There are three "bulk" subcommands available:

```
USAGE:
    krillc bulk [SUBCOMMAND]

SUBCOMMANDS:
    publish    Force that all CAs create new objects if needed (in which case␣
↪they will also sync)
    refresh    Force that all CAs ask their parents for updated certificates
    sync       Force that all CAs sync with their repo server
```

6.31 krillc bulk publish

Force that all CAs create new objects if needed (in which case they will also sync). Note that this function is executed when Krill starts up and then again every 10 minutes.

Example CLI:

```
$ krillc bulk publish
```

Example API call:

```
$ krillc bulk publish --api
POST:
  https://localhost:3000/api/v1/bulk/cas/publish
Headers:
  Authorization: Bearer secret
Body:
<empty>
```

6.32 krillc bulk refresh

Force that all CAs ask their parents for updated certificates. Note that this function is executed when Krill starts up and then again every 10 minutes.

Example CLI:

```
$ krillc bulk refresh
```

Example API call:

```
$ krillc bulk refresh --api
POST:
  https://localhost:3000/api/v1/bulk/cas/sync/parent
Headers:
  Authorization: Bearer secret
Body:
<empty>
```

6.33 krillc bulk sync

Force that all CAs sync with their publication server.

This function is executed when Krill starts up. When Krill is running then CAs will synchronise with their publication server whenever there is new content to publish. And if such a synchronisation fails, then Krill will schedule another attempt every 5 minutes until synchronisation succeeds.

However, if you believe that there is an issue with the publication server, or you wish to debug connection issues, then you can trigger this function manually:

```
$ krillc bulk sync --api
POST:
  https://localhost:3000/api/v1/bulk/cas/sync/repo
Headers:
  Authorization: Bearer secret
Body:
<empty>
```

6.34 krillc children

Manage children for a CA in Krill.

Most operators will not need this, but just like you can operate your Krill CA under an RIR or NIR, you can delegate your resources to so-called child CAs. This may be useful in case you need to authorise different units of your organisation or customers to manage some of your prefixes.

```
USAGE:
    krillc children [SUBCOMMAND]

SUBCOMMANDS:
    add          Add a child to a CA.
    info         Show info for a child (id and resources).
```

```
update      Update an existing child of a CA.
response    Get the RFC8183 response for a child.
remove      Remove an existing child from a CA.
```

6.35 krillc children add

Add a child to a CA. To add a child, you will need to:

1. Choose a unique local name (handle) that the parent will use for the child
2. Choose initial resources (asn, ipv4, ipv6)
3. Present the child's **RFC 8183** request

The default response is the **RFC 8183** parent response XML file. Or, if you set `--format json` you will get the plain API response.

If you need the response again, you can use the *krillc children response* command.

When you use the CLI you can provide a path to the Child Request XML and the CLI will parse this, and convert it to the JSON that Krill expects when adding a child. We chose to use a different format here because we needed to include other information not contained in the XML. I.e. just submitting the plain XML would not work here.

Example CLI:

```
$ krillc children add --ca testbed --child newca --ipv4 "10.0.0.0/8" --ipv6
 ↪"2001:db8::/32" --asn "AS65000" --request ./data/new-ca-child-request.xml
<parent_response xmlns="http://www.hactrn.net/uris/rpki/rpki-setup/" version="1"␣
 ↪service_uri="https://localhost:3000/rfc6492/testbed" child_handle="newca" parent_
 ↪handle="testbed">
  <parent_bpki_ta>
 ↪MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyhFOTBDMjE3MzRDMkMzNzBBOTFBODQ3NUNCNEYwRTc
 ↪vVYxq1F1w2yQ/
 ↪VoTr1dvEHxJ+SDayMcFVktWCObiY8tcPhvWG+OdaX9ckDJhsOEEvdVEogwiGacNs7yXJPbqDBptJtbR8/
 ↪CauF9OqMqjkB/8xkGmBoY5OI/
 ↪V2832jkp7LPsbyET0RMQN7fgSpGbewvkaZVxGU3pHh5kT1nzPTXrwjxNMXgpunSEY7zR20vYCvsYYbxnSwFNbSMSL+Jgpa+HWPU
 ↪qlqlZK+tkppI2LkSBhTV5+n7cGA8ZsCAwEAAaNTMFEwDwYDVR0TAQH/BAUwAwEB/
 ↪zAdBgNVHQ4EFgQU6Qwhc0wsNwqRqEdctPDnXaTQ8L8wHwYDVR0jBBgwFoAU6Qwhc0wsNwqRqEdctPDnXaTQ8L8wDQYJKoZIhvcN
 ↪nVmqDlK/1sWZNUXFWP4dt1wLTjDWnceyS8mI7Yx8dH/Fez60m4lp4dD45eeaXfbjP2cWnh3n/
 ↪PLGE70Nj+G0AnUhUmwiTl0H6Px1xn8fZouhv9MEheaZJA+M4NF77+Nmkp2P3WI4cvIS7Te7R/
 ↪7XpwSr29lVNtYjmRlrBDXx/bMFSgFL61mrtj/
 ↪l6G8OB40w+sAwO0XKUj1vUUpfIXc3ISCo0LNT9JSPcgy1SZWfmLb98q4HuvxekhkIPRzW7vlb/
 ↪NBXGarZmKc+HQjE2aXcIewhen2OoTSNda2jSSuEWZuWzZu0aMCKwFBNHLqs=</parent_bpki_ta>
</parent_response>
```

Example API call:

```
$ krillc children add --ca testbed --child newca --ipv4 "10.0.0.0/8" --ipv6
 ↪"2001:db8::/32" --asn "AS65000" --request ./data/new-ca-child-request.xml --api
POST:
  https://localhost:3000/api/v1/cas/testbed/children
Headers:
  content-type: application/json
  Authorization: Bearer secret
Body:
{
  "handle": "newca",
  "resources": {
```

(continues on next page)

(continued from previous page)

```
    "asn": "AS65000",
    "v4": "10.0.0.0/8",
    "v6": "2001:db8::/32"
  },
  "id_cert":
↪"MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyhFRjJENzgwRkNCRkU1QjZBMkExMjA1OUM0MDlDN0
↪X3Ho5gC7uuKf4LYqbJj+Z9ltr/236/
↪hDYJfWMXZVcEuL+wUble1zhe2NKrgnAkpReVMSdiugoqZ9ICK2Fwkj5jCGc/qHiWOba7T78zfij8OlB/
↪dGlJvkAY8b/XTNKsTrLozi1uVAC8GqDrV5MEgY/NfzUvgA024yxx/
↪rC6QBDEoBjnP7wDFiaZ2lwvL2beVYu6/
↪hVcXQzsVN+ijy7cGdkE6zi0meXJLTHPEpoA88hi3Pi+pIDBIQ3wTcpQIOqAq/
↪SZuh4dbZK7BV8MCAwEAAaNTMFEwDwYDVR0TAQH/BAUwAwEB/
↪zAdBgNVHQ4EFgQU7y14D8v+W2oqEgWcQJx8kndUGVYwHwYDVR0jBBgwFoAU7y14D8v+W2oqEgWcQJx8kndUGVYwDQYJKoZIhvcN
↪gpJtONdgIWV1EqwEzhKKA2EP6tLDF9ejsdMFNYrYr+2hVWaoLsSuarfwfLFSgKDFqR6sh3ljYq6mIz9gdkjBOJsR9JyHFEtsDsR
↪YPMPVBpmG15Z9iKantzC1tck+E1xYW5awvj+YZqGVqyFdPJOZWmaYoS83kWvg4g4IucXTH6wwy23MQ7+0gyoK4wxfXRQmWjlXpL
↪d7srQA4IxZCRGh9B+JdAIOKuXBA0nncmMJLQN8Qpxlz2bxKKAgXBLdoDqjbTDVbXTPM8YLRgc="
}
```

6.36 krillc children info

Show info for a child (id and resources).

Example CLI:

```
$ krillc children info --ca testbed --child newca
-----BEGIN CERTIFICATE-----
MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyhFRjJE
NzgwRkNCRkU1QjZBMkExMjA1OUM0MDlDN0M5Mjc3NTQxOTU2MB4XDTIxMDQwNzE0
MzUxNFoXDTM2MDQwNzE0NDAxNFowMzExMC8GA1UEAxMoRUYyRDc4MEZDQkZFNUI2
QTJBMTIwNTlDNDA5QzdDOTI3NzU0MTk1NjCCASIwDQYJKoZIhvcNAQEBBQADggEP
ADCCAQoCggEBANuBsEO4C9n7PlYcDT0PTeZntR5l778lZQDsgxiB7ofLrg8lKcf8
ugFiYI4vRqR+gDMHhR3t/X3Ho5gC7uuKf4LYqbJj+Z9ltr/236/hDYJfWMXZVcEu
L+wUble1zhe2NKrgnAkpReVMSdiugoqZ9ICK2Fwkj5jCGc/qHiWOba7T78zfij8O
lB/dGlJvkAY8b/XTNKsTrLozi1uVAC8GqDrV5MEgY/NfzUvgA024yxx/rC6QBDEo
BjnP7wDFiaZ2lwvL2beVYu6/hVcXQzsVN+ijy7cGdkE6zi0meXJLTHPEpoA88hi3
Pi+pIDBIQ3wTcpQIOqAq/SZuh4dbZK7BV8MCAwEAAaNTMFEwDwYDVR0TAQH/BAUw
AwEB/zAdBgNVHQ4EFgQU7y14D8v+W2oqEgWcQJx8kndUGVYwHwYDVR0jBBgwFoAU
7y14D8v+W2oqEgWcQJx8kndUGVYwDQYJKoZIhvcNAQELBQADggEBAArqsa/gpJtO
NdgIWV1EqwEzhKKA2EP6tLDF9ejsdMFNYrYr+2hVWaoLsSuarfwfLFSgKDFqR6sh
3ljYq6mIz9gdkjBOJsR9JyHFEtsDsRpf8Hs1WlbIb8bWb73Cp/YPMPVBpmG15Z9i
KantzC1tck+E1xYW5awvj+YZqGVqyFdPJOZWmaYoS83kWvg4g4IucXTH6wwy23MQ
7+0gyoK4wxfXRQmWjlXpLueCOsJo7ZXopsDAmXHLoFKZVEXn1ocQNc911521BEQ6
t/d7srQA4IxZCRGh9B+JdAIOKuXBA0nncmMJLQN8Qpxlz2bxKKAgXBLdoDqjbTDV
bXTPM8YLRgc=
-----END CERTIFICATE-----

SHA256 hash of PEM encoded certificate:␣
↪992ac17d85fef11d8be4aa37806586ce68b61fe9cf65c0965928dbce0c398a99
resources: asn: , v4: 10.0.0.0/8, 192.168.0.0/16, v6:
```

Example JSON response:

```
{
  "id_cert": {
```

(continues on next page)

(continued from previous page)

```
    "pem": "-----BEGIN CERTIFICATE-----\
↪nMIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyhFRjJE\
↪nNzgwRkNCRkU1QjZBMkExMjA1OUM0MDlDN0M5Mjc3NTQxOTU2MB4XDTIxMDQwNzE0\
↪nMzUxNFoXDTM2MDQwNzE0NDAxNFowMzExMC8GA1UEAxMoRUYyRDc4MEZDQkZFNUI2\
↪nQTJBMTIwNTlDNDA5QzdDOTI3NzU0MTk1NjCCASIwDQYJKoZIhvcNAQEBBQADggEP\
↪nADCCAQoCggEBANuBsEO4C9n7PlYcDT0PTeZntR5l778lZQDsgxiB7ofLrg8lKcf8\
↪nugFiYI4vRqR+gDMHhR3t/X3Ho5gC7uuKf4LYqbJj+Z9ltr/236/hDYJfWMXZVcEu\
↪nL+wUble1zhe2NKrgnAkpReVMSdiugoqZ9ICK2Fwkj5jCGc/qHiWOba7T78zfij8O\nlB/dGlJvkAY8b/
↪XTNKsTrLozi1uVAC8GqDrV5MEgY/NfzUvgA024yxx/rC6QBDEo\nBjnP7wDFiaZ2lwvL2beVYu6/
↪hVcXQzsVN+ijy7cGdkE6zi0meXJLTHPEpoA88hi3\nPi+pIDBIQ3wTcpQIOqAq/
↪SZuh4dbZK7BV8MCAwEAAaNTMFEwDwYDVR0TAQH/BAUw\nAwEB/
↪zAdBgNVHQ4EFgQU7y14D8v+W2oqEgWcQJx8kndUGVYwHwYDVR0jBBgwFoAU\
↪n7y14D8v+W2oqEgWcQJx8kndUGVYwDQYJKoZIhvcNAQELBQADggEBAArqsa/gpJtO\
↪nNdgIWV1EqwEzhKKA2EP6tLDF9ejsdMFNYrYr+2hVWaoLsSuarfwfLFSgKDFqR6sh\
↪n3ljYq6mIz9gdkjBOJsR9JyHFEtsDsRpf8Hs1WlbIb8bWb73Cp/YPMPVBpmG15Z9i\
↪nKantzC1tck+E1xYW5awvj+YZqGVqyFdPJOZWmaYoS83kWvg4g4IucXTH6wwy23MQ\
↪n7+0gyoK4wxfXRQmWjlXpLueCOsJo7ZXopsDAmXHLoFKZVEXn1ocQNc911521BEQ6\nt/
↪d7srQA4IxZCRGh9B+JdAIOKuXBA0nncmMJLQN8Qpxlz2bxKKAgXBLdoDqjbTDV\nbXTPM8YLRgc=\n-----
↪END CERTIFICATE-----\n",
    "hash": "992ac17d85fef11d8be4aa37806586ce68b61fe9cf65c0965928dbce0c398a99"
  },
  "entitled_resources": {
    "asn": "",
    "v4": "10.0.0.0/8, 192.168.0.0/16",
    "v6": ""
  }
}
```

Example API call:

```
$ krillc children info --ca testbed --child newca  --api
GET:
  https://localhost:3000/api/v1/cas/testbed/children/newca
Headers:
  Authorization: Bearer secret
```

6.37 krillc children update

Update the resource entitlements of an existing child of a CA, or update the identity certificate that they will use when sending **RFC 6492** requests.

Important: When updating resources you need to specify the full new set of resource entitlements for the child. This is not a delta. Also if you specify one resource type only like `--ipv4`, then `--ipv6` and `--asn` will be assumed to be intentionally empty:

```
$ krillc children update --ca testbed --child newca --ipv4 "10.0.0.0/8"  --api
POST:
  https://localhost:3000/api/v1/cas/testbed/children/newca
Headers:
  content-type: application/json
  Authorization: Bearer secret
Body:
```

(continues on next page)

(continued from previous page)

```
{
  "id_cert": null,
  "resources": {
    "asn": "",
    "v4": "10.0.0.0/8",
    "v6": ""
  }
}
```

When updating an ID certificate the CLI expects it to be DER encoded. It will submit it in base64 encoded form to the API and leave the "resources" as *null* then. The *null* value means that this is not updated:

```
$ krillc children update --ca testbed --child newca --idcert ./data/new-ca.cer --api
POST:
  https://localhost:3000/api/v1/cas/testbed/children/newca
Headers:
  content-type: application/json
  Authorization: Bearer secret
Body:
{
  "id_cert":
↪"MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyhFRjJENzgwRkNCRkU1QjZBMkExMjA1OUM0MDlDN0
↪X3Ho5gC7uuKf4LYqbJj+Z9ltr/236/
↪hDYJfWMXZVcEuL+wUble1zhe2NKrgnAkpReVMSdiugoqZ9ICK2Fwkj5jCGc/qHiWOba7T78zfij8OlB/
↪dGlJvkAY8b/XTNKsTrLozi1uVAC8GqDrV5MEgY/NfzUvgA024yxx/
↪rC6QBDEoBjnP7wDFiaZ2lwvL2beVYu6/
↪hVcXQzsVN+ijy7cGdkE6zi0meXJLTHPEpoA88hi3Pi+pIDBIQ3wTcpQIOqAq/
↪SZuh4dbZK7BV8MCAwEAAaNTMFEwDwYDVR0TAQH/BAUwAwEB/
↪zAdBgNVHQ4EFgQU7y14D8v+W2oqEgWcQJx8kndUGVYwHwYDVR0jBBgwFoAU7y14D8v+W2oqEgWcQJx8kndUGVYwDQYJKoZIhvcN
↪gpJtONdgIWV1EqwEzhKKA2EP6tLDF9ejsdMFNYrYr+2hVWaoLsSuarfwfLFSgKDFqR6sh3ljYq6mIz9gdkjBOJsR9JyHFEtsDsF
↪YPMPVBpmG15Z9iKantzC1tck+E1xYW5awvj+YZqGVqyFdPJOZWmaYoS83kWvg4g4IucXTH6wwy23MQ7+0gyoK4wxfXRQmWjlXpI
↪d7srQA4IxZCRGh9B+JdAIOKuXBA0nncmMJLQN8Qpxlz2bxKKAgXBLdoDqjbTDVbXTPM8YLRgc=",
  "resources": null
}
```

6.38 krillc children response

Get the **RFC 8183** Parent Response for a child. The child will need this to add your CA as their parent.

Example CLI:

```
$ krillc children response --ca testbed --child newca
<parent_response xmlns="http://www.hactrn.net/uris/rpki/rpki-setup/" version="1"␣
↪service_uri="https://localhost:3000/rfc6492/testbed" child_handle="newca" parent_
↪handle="testbed">
  <parent_bpki_ta>
↪MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyhFOTBDMjE3MzRDMkMzNzBBOTFBODQ3NUNCNEYwRTc
↪vVYxq1F1w2yQ/
↪VoTr1dvEHxJ+SDayMcFVktWCObiY8tcPhvWG+OdaX9ckDJhsOEEvdVEogwiGacNs7yXJPbqDBptJtbR8/
↪CauF9OqMqjkB/8xkGmBoY5OI/
↪V2832jkp7LPsbyET0RMQN7fgSpGbewvkaZVxGU3pHh5kT1nzPTXrwjxNMXgpunSEY7zR20vYCvsYYbxnSwFNbSMSL+Jgpa+HWPU
↪qlqlZK+tkppI2LkSBhTV5+n7cGA8ZsCAwEAAaNTMFEwDwYDVR0TAQH/BAUwAwEB/
↪zAdBgNVHQ4EFgQU6Qwhc0wsNwqRqEdctPDnXaTQ8L8wHwYDVR0jBBgwFoAU6Qwhc0wsNwqRqEdctPDnXaTQ8L8wDQYJKoZIhvcN
↪nVmqDlK/1sWZNUXFWP4dt1wLTjDWnceyS8mI7Yx8dH/Fez60m4lp4dD45eeaXfbjP2cWnh3n/
↪PLGE70Nj+G0AnUhUmwiTl0H6Px1xn8fZouhv9MEheaZJA+M4NF77+Nmkp2P3WI4cvIS7Te7R/
↪7XpwSr29lVNtYjmRlrBDXx/bMFSgFL61mrtj/
↪l6G8OB40w+sAwO0XKUj1vUUpfIXc3ISCo0LNT9JSPcgy1SZWfmLb98q4HuvxekhkIPRzW7vlb/
↪NBXGarZmKc+HQjE2aXcIewhen2OoTSNda2jSSuEWZuWzZu0aMCKwFBNHLqs=</parent_bpki_ta>
```

(continues on next page)

(continued from previous page)

```
</parent_response>
```

Example API call:

```
$ krillc children response --ca testbed --child newca --api
GET:
  https://localhost:3000/api/v1/cas/testbed/children/newca/contact
Headers:
  Authorization: Bearer secret
```

Note that the API always returns the **RFC 8183** Parent Response in JSON format, but the CLI converts it. Other API endpoints support getting such files in either JSON or RFC standard XML format. If there is desire to support this here as well, then we will add this in a future release.

6.39 krillc children remove

Remove an existing child from a CA. We suggest that you discuss this action with your child first. Removing the child will revoke their certificate(s), but they may continue to ask for their entitlements until they decide to remove your CA as a parent as well. Of course if the child had multiple parents, they may continue to maintain healthy relationships with their other parent(s).

Example CLI / API call:

```
$ krillc children remove --ca testbed --child newca --api
DELETE:
  https://localhost:3000/api/v1/cas/testbed/children/newca
Headers:
  Authorization: Bearer secret
```

6.40 krillc keyroll

Perform a key rollover for a CA.

Krill supports **RFC 6489** Key Rollovers. The process is manual for now. I.e. it's up to the operator to initiate a key rollover - there is no automation based on key age for example. We expect that this is what operators would want. More importantly though, this also means that operators should execute *both* steps in the process to start *and* finish the key rollover:

```
krillc keyroll init        Initialise roll for all keys held by this CA.
krillc keyroll activate    Finish roll for all keys held by this CA.
```

6.41 krillc keyroll init

Initialise roll for all keys held by this CA.

Example CLI/API call:

```
$ krillc keyroll init --ca newca --api
POST:
  https://localhost:3000/api/v1/cas/newca/keys/roll_init
Headers:
  Authorization: Bearer secret
Body:
<empty>
```

6.41.1 krillc keyroll activate

Finish roll for all keys held by this CA.

Note that **RFC 6489** says that you should wait 24 hours before doing this step. So, please observe this period for planned key rollovers. For emergency rollovers where the old key is compromised, or if this rollover is part of an emergency migration to a new publication server, do this step as soon as possible.

Example CLI/API:

```
$ krillc keyroll activate --ca newca --api
POST:
  https://localhost:3000/api/v1/cas/newca/keys/roll_activate
Headers:
  Authorization: Bearer secret
Body:
<empty>
```

CHAPTER

SEVEN

USING THE API

The Krill API is a primarily JSON based REST-like HTTPS API with bearer token based authentication.

7.1 Getting Help

- Consult the Interactive API documentation (courtesy of ReDoc)
- Follow the API links in the *Krill CLI documentation*, e.g. API Call: **:krill_api:`GET /v1/cas <list_cas>`**
- Check out the API hints built-in to the *Krill CLI*, e.g.

```
$ krillc list --api
GET:
  https://<your.domain>/api/v1/cas
Headers:
  Authorization: Bearer *****
```

7.2 Generating Client Code

The OpenAPI Generator can generate Krill API client code in many languages from the Krill v0.7.3 OpenAPI specification.3

7.3 Sample Application

Below is an example of how to write a small Krill client application in Python 3 using a Krill API client library produced by the OpenAPI Generator. To try out this sample you'll need Docker and Python 3.

1. Save the following as `/tmp/krill_test.py`, replacing *<YOUR XXX>* values with the correct access token and domain name for your Krill server. This example assumes that your Krill instance API endpoint is available on port 443 using a valid TLS certificate.

```
# Import the OpenAPI generated Krill client library
import krill_api
from krill_api import *

# Create a configuration for the client library telling it how to connect to
# the Krill server
krill_api_config = krill_api.Configuration()
krill_api_config.access_token = '<YOUR KRILL API TOKEN>'
```

(continues on next page)

(continued from previous page)

```
krill_api_config.host = "https://{}/api/v1".format('<YOUR KRILL FQDN>')
krill_api_config.verify_ssl = True
krill_api_config.assert_hostname = False
krill_api_config.cert_file = None

# Create a Krill API client
krill_api_client = krill_api.ApiClient(krill_api_config)

# Get the client helper for the Certificate Authority set of Krill API endpoints
krill_ca_api = CertificateAuthoritiesApi(krill_api_client)

# Query Krill for the list of configured CAs
print(krill_ca_api.list_cas())
```

2. Run the following commands in a shell to generate a Krill client library:

```
# prepare a working directory
GENDIR=/tmp/gen
VENVDIR=/tmp/venv
mkdir -p $GENDIR

# fetch the Krill OpenAPI specification document
wget -O $GENDIR/openapi.yaml https://raw.githubusercontent.com/NLnetLabs/krill/v0.7.3/
↪doc/openapi.yaml

# use the OpenAPI Generator to generate a Krill client library from the krill
# OpenAPI specification
docker run --rm -v $GENDIR:/local \
    openapitools/openapi-generator-cli generate \
    -i /local/openapi.yaml \
    -g python \
    -o /local/out \
    --additional-properties=packageName=krill_api

# install the generated library where your Python 3 can find it
python3 -m venv $VENVDIR
source $VENVDIR/bin/activate
pip3 install wheel
pip3 install $GENDIR/out/
```

3. Run the sample application:

```
$ python3 /tmp/krill_test.py
{'cas': [{'handle': 'ca'}]}
```

Tip: To learn more about using the generated client library, consult the documentation in *$GENDIR/out/README.md*.

Warning: Future improvements to the Krill OpenAPI specification may necessitate that you re-generate your client library and possibly also alter your client program to match any changed class and function names.

CHAPTER

EIGHT

MONITORING

The HTTPS server in Krill provides endpoints for monitoring the application. A data format specifically for Prometheus is available and dedicated port 9657 has been reserved.

On the `/metrics` path, Krill will expose several data points:

- A timestamp when the daemon was started
- The number of CAs Krill has configured
- The number of children for each CA
- The number of ROAs for each CA
- Timestamps when publishers were last updated
- The number of objects in the repository for each publisher
- The size of the repository, in bytes
- The RRDP serial number

This is an example of the output of the `/metrics` endpoint:

```
# HELP krill_server_start timestamp of last krill server start
# TYPE krill_server_start gauge
krill_server_start 1582189609

# HELP krill_repo_publisher number of publishers in repository
# TYPE krill_repo_publisher gauge
krill_repo_publisher 1

# HELP krill_repo_rrdp_last_update timestamp of last update by any publisher
# TYPE krill_repo_rrdp_last_update gauge
krill_repo_rrdp_last_update 1582700400

# HELP krill_repo_rrdp_serial RRDP serial
# TYPE krill_repo_rrdp_serial counter
krill_repo_rrdp_serial 128

# HELP krill_repo_objects number of objects in repository for publisher
# TYPE krill_repo_objects gauge
krill_repo_objects{publisher="acme-corp-intl"} 6

# HELP krill_repo_size size of objects in bytes in repository for publisher
# TYPE krill_repo_size gauge
krill_repo_size{publisher="acme-corp-intl"} 16468
```

(continues on next page)

(continued from previous page)

```
# HELP krill_repo_last_update timestamp of last update for publisher
# TYPE krill_repo_last_update gauge
krill_repo_last_update{publisher="acme-corp-intl"} 1582700400

# HELP krill_cas number of cas in krill
# TYPE krill_cas gauge
krill_cas 1

# HELP krill_cas_roas number of roas for CA
# TYPE krill_cas_roas gauge
krill_cas_roas{ca="acme-corp-intl"} 4

# HELP krill_cas_children number of children for CA
# TYPE krill_cas_children gauge
krill_cas_children{ca="acme-corp-intl"} 0
```

The monitoring service has several additional endpoints on the following paths:

/stats/info Returns the Krill version and timestamp when the daemon was started in a concise format

/stats/cas Returns the number of ROAs and children each CA has

/stats/repo Returns details on the internal repository, if configured

CHAPTER

NINE

FAILURE SCENARIOS

Important: The most important thing to remember about possible failure scenarios in your setup is that Krill is designed to run continuously, but there is no strict uptime requirement for the Certificate Authority (CA). If the CA is not available you just cannot create or update ROAs. As long as your existing ROAs are being served by your web server and rsync server, you have time to solve problems.

As long as Krill is running, it will automatically update the entitled resources on your certificate, as well as reissue certificates, ROAs and all other objects before they expire or become stale. If Krill does go down, you have 8 hours to bring it back up before data starts going stale.

9.1 Hardware and Software Failures

9.1.1 Failure of the Primary Krill CA Server

When the primary CA server goes down, users will not be able to create or edit ROAs until the server is back up or failover to a secondary server has been completed. Relying party software will continue to be able to download and verify existing ROAs.

With default timing settings new manifests and certificate revocation lists (CRLs) will be published eight hours before they would expire, with a validity time of 24 hours. This means that the CA should be restored within eight hours after failure. Please note that values can be changed. For example, it would be perfectly fine to republish 16 hours before expiry, or even longer if the validity time is also extended. However, we do not recommend extending the validity time beyond 24 hours because it could allow (theoretical) attacks where data is replayed to validators for longer.

9.1.2 The Disk on the Primary Krill CA Server is Full

Krill will crash, by design, if there is any failure in saving any state file to disk. If Krill cannot persist its state it should not try to carry on. It could lead to disjoints between in-memory and on-disk state that are impossible to fix. Therefore, crashing and forcing an operator to look at the system is the only sensible thing Krill can now do.

If this occurs, more space should be allocated. Krill can then try to restart properly. It will try to go back to the last possible recoverable state if:

- it cannot rebuild its state at startup due to data corruption
- the environment variable: `KRILL_FORCE_RECOVER` is set
- the configuration file contains `always_recover_data = true`

Under normal circumstances performing this recovery will not be necessary. It can also take significant time due to all the checks performed. So, we do not recommend forcing recovery when there is no data corruption. See *Recover State at Startup* for more details.

During this failure, Relying party software will continue to be able to download and verify existing ROAs.

9.1.3 Corruption of Files on the Primary Krill CA Server

If an administrator or malicious attacker with access to the server modifies files after Krill has written them, then Krill will just continue to operate. It will not re-read files as it keeps all data in memory as well. However, Krill may be unable to come up if it is restarted. By default Krill will then only read the latest snapshots of its state components and modifications since then. Krill will start if those files are unaffected.

Similar to the situation with a full disk, Krill will try to go back to the last possible recoverable state. See *Recover State at Startup* for more details.

Krill can also be restored from a backup, but it would result in losing all changes from after the backup. As described above, if Krill finds that the backup contains an incomplete transaction, it will fall back to the state prior to it.

During this failure, Relying party software will continue to be able to download and verify existing ROAs.

9.1.4 The Disk on the Krill Publication Server is Full

Krill will crash, by design, if there is any failure in saving any state file to disk. If this occurs, more space should be allocated. Krill can then try to restart properly. It will try to go back to the last possible recoverable state as described in the *Recover State at Startup* section.

If the repository is restored from back up, Relying Party software may be served outdated content from that backup for a short while. However, when the service is resumed then the Krill CA will do a full re-sync and publish its current content within 5 minutes if it had pending unpublished changes. If not, then a manual re-sync can be triggered through the user interface, the command line or the API.

During this failure, Relying Party software will continue to be able to download and verify existing ROAs.

9.1.5 Failure of one of the Repository Servers

When a repository server running the web server and/or rsyncd server goes down or the service dies, the load balancer should mitigate the issue. An error should be generated and the failed instance should be taken out of the pool. During this failure, Relying party software will continue to be able to download and verify ROAs from the other repository servers that are configured.

In the extreme case when all repository servers become unavailable, Relying Party software that has already downloaded the repository content will use cached data on disk. Depending on when data was last fetched, the RPKI validity of BGP announcements will be unaffected between 8 and 24 hours, after which ROAs and all other objects expire or become stale. Relying Party software that has never downloaded the repository contents will not affect the RPKI validity of any BGP announcements; they will have the RPKI state *Not Found*.

9.2 Usage Failures

9.2.1 A Misconfigured ROA Causes a Legitimate Route to be Considered RPKI Invalid

When an operator or a script generates a ROA that causes a legitimate route to be considered RPKI Invalid, it may (severely) affect reachability of the network, as it is increasingly likely that other networks will start dropping the BGP announcement.

Monitoring should be in place to detect misconfigured ROAs. If any legitimate BGP announcement has the state *RPKI Invalid* the operator should remove the ROA and replace it with a correct one. Note that verifying RPKI validity state does not have to be performed on actual BGP announcements seen in the DFZ. An ASN and prefix combo can be fed to several Relying Party software packages, which will return the RPKI validity state based on its current cache.

9.2.2 An Incorrect ROA is Published for the Repository Servers

When one or more ROAs are generated that cause the prefix that contains the RPKI repository server IP to be considered *RPKI Invalid*, Relying Party software will retrieve these ROAs and promptly those networks will drop the repository prefix. This means that even if/when operators fix the ROAs, the validators will not be able to retrieve the updated information until their cached manifest and CRLs go stale. This issue can persist for a minimum of 8 hours and a maximum of 24 hours.

CHAPTER

TEN

RUNNING A PUBLICATION SERVER

Important: It is highly recommended to use an RPKI publication server provided by your parent CA, if available. This relieves you of the responsibility to keep a public rsync and web server available at all times.

If you need to run your own Publication Server using Krill, then we recommend that you use a separate Krill instance acting as a repository only. This setup allows for much easier reconfiguration (more on this below), and it allows that other CAs - for example a delegated CA for one of your business units also publish at this same Publication Server.

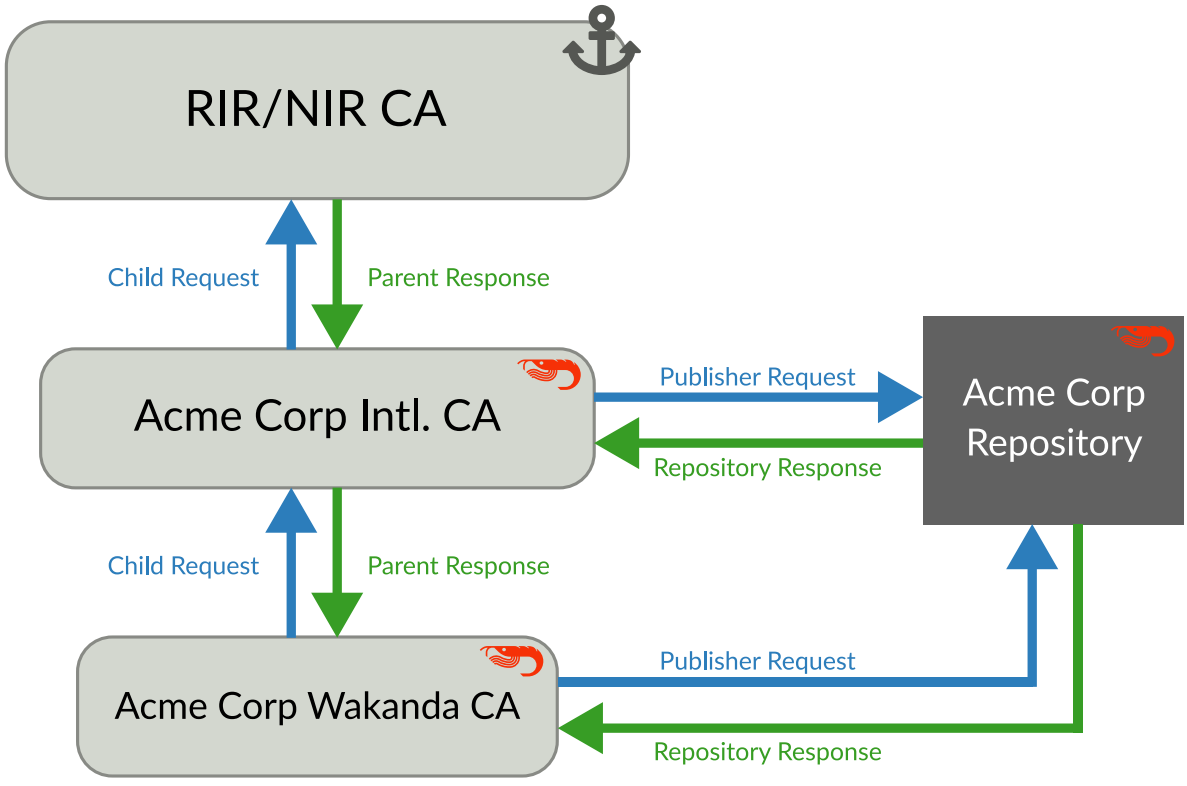


Fig. 1: Running a publication server for yourself and your children

10.1 Configuring a Krill Repository

Note: The Krill UI is not currently aimed at using Krill as a repository server. For example when visiting the UI of a Krill instance intended for use only as a repository and not as a CA, it will still prompt you on first use to create a CA. There is also no support via the UI for managing the repository, for example it is not possible via the UI to complete a child request to register with the repository.

Krill can be set up to run as a Publication Server through its configuration file. If enabled, the Publication Server is created on start-up. After this any updates to the configuration will *NOT* be reflected in the Publication Server.

For this reason you should double check the values used for the public URIs to your repository server carefully before the set-up. Using a dedicated Krill instance for the Publication Server will allow you to simply destroy and replace the instance should it have been misconfigured.

The easiest way to make a configuration file is by using *krillc config* to generate the required configuration for you. For example:

```
krillc config repo \
   --server "https://rfc8183.example.net/" \
   --token correct-horse-battery-staple \
   --data ~/data/ \
   --rrdp "https://rpki.example.net/rrdp/" \
   --rsync "rsync://rpki.example.net/repo/" > krill.conf
```

Make sure that the `--server` option reflects a base URI that your Krill CA publication clients can reach, and make sure that this URI is exposed using a proxy server that has a proper HTTPS certificate configured.

Make sure that the `--rrdp` and `--rsync` options match the configuration of your "Repository Servers" which make your repository available over HTTPS and rsync to Relying Parties.

Note: It would have been better to make the Publication Server configuration something that should be done run-time, as this would match more intuitively with the fact that the *server*, *rrdp* and *rsync* URIs cannot be changed through the configuration file.

In a future release of Krill we may do exactly that. But, even if we do it would be ill advised to allow changing these URIs at run time, as there would be no way for the Krill Publication Server to inform its publishers about any change.

So, in short, this needs to be set up correctly once. If it turns out to be wrong, then a new Publication Server should be set up and any existing publishers should be migrated as described below.

10.2 Proxy for Remote Publishers

Krill runs the RFC8181 Publication Server. Remote publishers, CAs which use your Publication Server, will need to connect to this under the */rfc8181* path under the *service_uri* that you specified in your server.

Make sure that you set up a proxy server such as NGINX, Apache, etc. which uses a valid HTTPS certificate, and which proxies */rfc8181* to Krill.

Note that you should not add any additional authentication mechanisms to this location. RFC 8181 uses cryptographically signed messages sent over HTTP and is secure. However, verifying messages and signing responses can be computationally heavy, so if you know the source IP addresses of your publisher CAs, you may wish to restrict access based on this.

10.3 Configuring Repository Servers

To actually serve the published content to Rsync and RRDP clients you will need to run your own *repository* servers using tools such as Rsyncd and NGINX.

Krill will write the repository files under the data directory specified in its configuration file:

```
$DATA_DIR/repo/rsync/current/    Contains the files for Rsync
$DATA_DIR/repo/rrdp/             Contains the files for HTTPS (RRDP)
```

You can share the contents of these directories with your repository servers in various ways. It is possible to have a redundant shared file system where the Krill Publication Server can write, and your repository servers can read. Alternatively, you can synchronise the contents of these directories in another way, such as rsyncing them over every couple of minutes.

If you are using a shared file system, please note that the rsync `/current` directory cannot be the mount point. Krill tries to write the entire repository to a new folder under `$DATA_DIR/repo/rsync` and then renames it. This is done to minimise issues with files being updated while relying party software is fetching data.

10.3.1 Rsync

The next step is to configure your rsync daemons to expose a 'module' for your files. Make sure that the Rsync URI including the 'module' matches the `rsync_base` in your Krill configuration file. Basic configuration can then be as simple as:

```
$ cat /etc/rsyncd.conf
uid = nobody
gid = nogroup
max connections = 50
socket options = SO_KEEPALIVE

[repo]
path = /var/lib/krill/data/repo/rsync/current/
comment = RPKI repository
read only = yes
```

10.3.2 RRDP

For RRDP you will need to set up a web server of your choice and ensure that it has a valid TLS certificate. Next, you can make the files found under, or copied from `$DATA_DIR/repo/rrdp` available here. Make sure that the public URI to the RRDP base directory matches the value of `rrdp_service_uri` in your `krill.conf` file, or the `--rrdp` option if you generated the configuration.

If desired, you can also use a CDN in front of this server to further reduce your load and uptime requirements. If you do, make sure that the public URI matches the directive in `krill.conf`, because this will be used in your RPKI certificate.

10.3.3 RFC 8181 (publication protocol)

Make sure that your Krill Publication Server can be reached by your Krill CA clients. The best way to do this, is by setting up a web server, similar to the RRDP set up above, which proxies access to URIs starting with `/rfc8181` under the hostname you specified with the `--server` option through to your Krill Publication Server.

10.4 Publishing in the Repository

As there is no UI support for this, you will need to use the command line interface using the krillc publisher subcommand to manage publishers.

This subcommand will allow you to add your Krill CA client's RFC8181 Publisher Request XML, and obtain a Repository Response XML for it. From the client CA's perspective this part of the process is exactly as described *here*.

To add the Krill CA client XML to your server use the following:

```
$ krillc publishers add --request <path-to-xml> [--publisher publisher]
```

If `--publisher` is not specified then the publisher identifier handle will be taken from the XML. Handles need to be unique. So, you may want or need to override this - especially if you provide your Publication Server as a service to others.

If successful this will show the response XML. But, you can also get this response XML for a configured publisher using the following:

```
$ krillc publishers response --publisher publisher
```

10.5 Migrating the Repository

If you find that there is an issue with your repository or, for example, you want to change its domain name, you can set up a new Krill instance for the new repository. When you are satisfied that the new one is correct, you can migrate your CA to it by adding them as a publisher under the new repository server, and then updating your CA to use the new repository.

Updating the repository of your Krill CAs is currently not possible using the UI, but you can archive this trough the command line interface connecting to your Krill instance that hosts your CA.

First you will need to get your CA's Publication Request XML using the following:

```
$ krillc repo request
```

You then need to give this XML to your Publication Server, be it provided by a third party or managed by yourself as described above. After receiving the Repository Response XML you can then update your CA's repository using:

```
$ krillc repo update --response <path-to-xml>
```

Krill will then make sure that objects are moved properly, and that a new certificate is requested from your parent(s) to match the new location. This scenario would also apply when your RIR starts providing a repository service. You can update your CA to start publishing there instead.

CHAPTER

ELEVEN

RUNNING WITH DOCKER

This page explains the additional features and differences compared to running Krill with Cargo that you need to be aware of when running Krill with Docker.

11.1 Get Docker

If you do not already have Docker installed, follow the platform specific installation instructions via the links in the Docker official "Supported platforms" documentation.

11.2 Fetching and Running Krill

The **`docker run`** command will automatically fetch the Krill image the first time you use it, and so there is no installation step in the traditional sense. The **`docker run`** command can take many arguments and can be a bit overwhelming at first.

The command below runs Krill in the background and shows how to configure a few extra things like log level and volume mounts (more on this below).

```
$ docker run -d --name krill -p 127.0.0.1:3000:3000 \
  -e KRILL_LOG_LEVEL=debug \
  -e KRILL_FQDN=rpki.example.net \
  -e KRILL_AUTH_TOKEN=correct-horse-battery-staple \
  -e TZ=Europe/Amsterdam \
  -v krill_data:/var/krill/data/ \
  -v /tmp/krill_rsync/:/var/krill/data/repo/rsync/ \
  nlnetlabs/krill
```

Note: The Docker container by default uses UTC time. If you need to use a different time zone you can set this using the TZ environment variable as shown in the example above.

11.3 Admin Token

By default Docker Krill secures itself with an automatically generated admin token. You will need to obtain this token from the Docker logs in order to manage Krill via the API or the **krillc** CLI tool.

```
$ docker logs krill 2>&1 | fgrep token
docker-krill: Securing Krill daemon with token <SOME_TOKEN>
```

You can pre-configure the token via the `auth_token` Krill config file setting, or if you don't want to provide a config file you can also use the Docker environment variable `KRILL_AUTH_TOKEN` as shown above.

11.4 Running the Krill CLI

11.4.1 Local

Using a Bash alias with `<SOME_TOKEN>` you can easily interact with the locally running Krill daemon via its command-line interface (CLI):

```
$ alias krillc='docker exec \
  -e KRILL_CLI_SERVER=https://127.0.0.1:3000/ \
  -e KRILL_CLI_TOKEN=correct-horse-battery-staple \
  nlnetlabs/krill krillc'

$ krillc list -f json
{
  "cas": []
}
```

11.4.2 Remote

The Docker image can also be used to run **krillc** to manage remote Krill servers. Using a shell alias simplifies this considerably:

```
 $ alias krillc='docker run --rm \
   -e KRILL_CLI_SERVER=https://rpki.example.net/ \
   -e KRILL_CLI_TOKEN=correct-horse-battery-staple \
   -v /tmp/ka:/tmp/ka nlnetlabs/krill krillc'

$ krillc list -f json
{
   "cas": []
}
```

Note: The `-v` volume mount is optional, but without it you will not be able to pass files to **krillc** which some subcommands require, e.g.

```
$ krillc roas update --ca my_ca --delta /tmp/delta.in
```

11.5 Service and Certificate URIs

The Krill `service_uri` and `rsync_base` config file settings can be configured via the Docker environment variable `KRILL_FQDN` as shown in the example above. Providing `KRILL_FQDN` will set **both** `service_uri` and `rsync_base`.

11.6 Data

Krill writes state and data files to a data directory which in Docker Krill is hidden inside the Docker container and is lost when the Docker container is destroyed.

11.6.1 Persistence

To protect the data you can write it to a persistent Docker volume which is preserved even if the Krill Docker container is destroyed. The following fragment from the example above shows how to configure this:

```
docker run -v krill_data:/var/krill/data/
```

11.6.2 Access

Some of the data files written by Krill to its data directory are intended to be shared with external clients via the rsync protocol. To make this possible with Docker Krill you can either:

- Mount the rsync data directory in the host and run rsyncd on the host, *OR*
- Share the rsync data with another Docker container which runs rsyncd

Mounting the data in a host directory:

```
docker run -v /tmp/krill_rsync:/var/krill/data/repo/rsync
```

Sharing via a named volume:

```
docker run -v krill_rsync:/var/krill/data/repo/rsync
```

11.7 Logging

Krill logs to a file by default. Docker Krill however logs by default to stderr so that you can see the output using the **`docker logs`** command.

At the default `warn` log level Krill doesn't output anything unless there is something to warn about. Docker Krill however comes with some additional logging which appears with the prefix `docker-krill:`. On startup you will see something like the following in the logs:

```
docker-krill: Securing Krill daemon with token ba473bac-021c-4fc9-9946-6ec109befec3
docker-krill: Configuring /var/krill/data/krill.conf ..
docker-krill: Dumping /var/krill/data/krill.conf config file
...
docker-krill: End of dump
```

11.8 Environment Variables

The Krill Docker image supports the following Docker environment variables which map to the following `krill.conf` settings:

Environment variable	Equivalent Krill config setting
`KRILL_AUTH_TOKEN`	`auth_token`
`KRILL_FQDN`	`service_uri` and `rsync_base`
`KRILL_LOG_LEVEL`	`log_level`
`KRILL_USE_TA`	`use_ta`

To set these environment variables use `-e` when invoking **`docker`**, e.g.:

```
docker run -e KRILL_FQDN=https://rpki.example.net/
```

11.9 Using a Config File

Via a volume mount you can replace the Docker Krill config file with your own and take complete control:

```
docker run -v /tmp/krill.conf:/var/krill/data/krill.conf
```

This will instruct Docker to replace the default config file used by Docker Krill with the file `/tmp/krill.conf` on your host computer.

INDEX

R