Krill User Manual

NLnet Labs

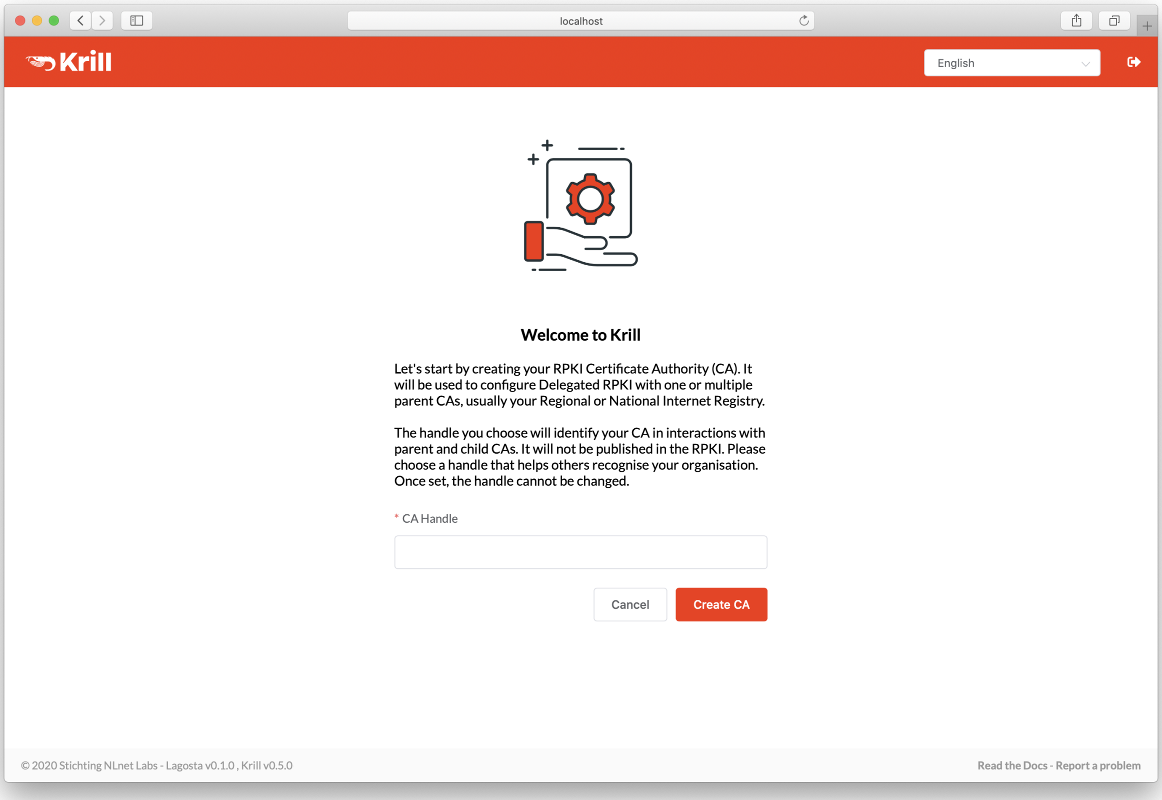
Sep 20, 2022

GETTING STARTED

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

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.

Note: For a quick summary of what's new and changed in the latest version see the release notes. If upgrading consult the *upgrade guide*.



Welcome to Krill

Let's start by creating your RPKI 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 choose will identify your CA in interactions with parent and child CAs. It will not be published in the RPKI. Please choose a handle that helps others recognise your organisation. Once set, the handle cannot be changed.

* CA Handle

© 2020 Stichting NLnet Labs - Lagosta v0.1.0 , Krill v0.5.0

Read the Docs - Report a problem

Krill is intended for:

- **Organisations who hold address space from multiple Regional Internet Registries (RIRs). Using Krill, ROAs can be managed seamlessly for all resources within one system.**
- **Organisations that need to be able to delegate RPKI to their customers or different business units, so that that they can run their own CA and manage ROAs themselves.**
- **Organisations who do not wish to rely on the web interface of the hosted systems that the RIRs offer, but require RPKI management that is integrated with their own systems using a common UI or API.**

Using Krill, you can run your own RPKI Certificate Authority as a child of one or more parent CAs, usually a Regional Internet Registry (RIR) or National Internet Registry (NIR). With Krill you can run under multiple parent CAs seamlessly and transparently. This is especially convenient if your organisation holds address space in several RIR regions, as it can all be managed as a single pool.

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

Lastly, Krill features a publication server so you can either publish your certificate and ROAs with a third party, such as your NIR or RIR, or you publish them yourself. Krill can be managed with a web user interface, from the command line and through an API.

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.

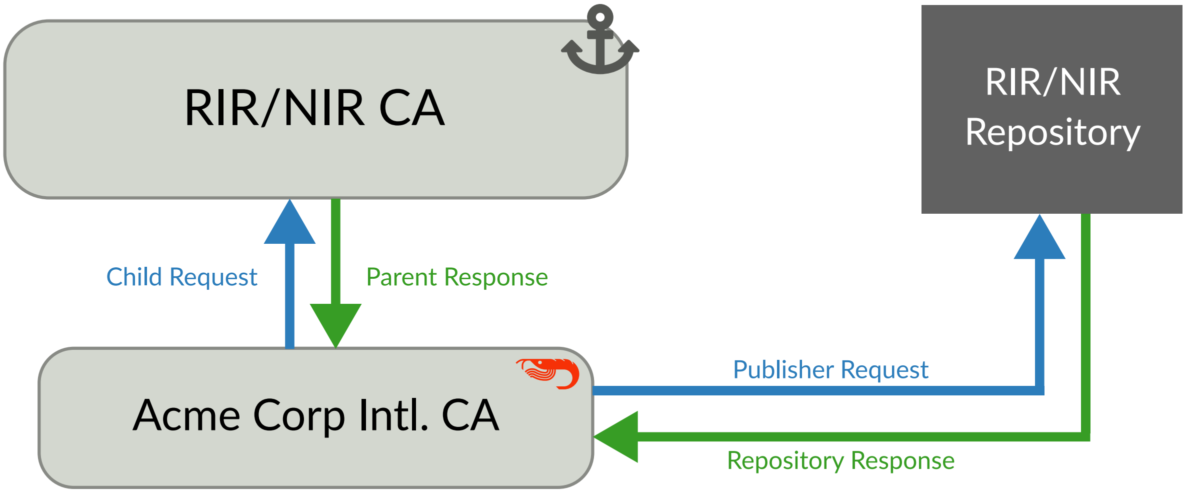


Fig. 1: A repository hosted by the parent CA, in this case the RIR or NIR.

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, APNIC, ARIN and Brazilian NIR NIC.br offer a publication server for their members. Several other RIRs have this functionality on their roadmap. This means that in some 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`. 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 save space - potentially quite a bit of 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 the gory details. However, understanding this section will help 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 (your CA etc.) 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 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.

> **Warning:** 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, then 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, i.e. when there is no data corruption, 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 this.

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 / 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 contain an incomplete transaction, it will just 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.

2.7 Krill Downgrades

Downgrading Krill data is not supported. So, downgrading can only be achieved by installing a previous version of Krill and restoring a backup from before your upgrade.

2.8 Proxy and HTTPS

2.8.1 HTTPS Mode

Krill uses HTTPS by default, and will generate a key pair and create a self-signed certificate if no previous key pair or certificate is found. Files are stored under the data directory as `ssl/key.pem` and `ssl/cert.pem` respectively.

Alternatively you make Krill configure krill to not generate these files but use existing files at the same file locations. This should work, but has not been tested extensively. To use this mode you can use `` `https_mode = "existing"` `` in your krill configuration file.

It also possible to force Krill to disable HTTPS and use plain HTTP. We do not recommend this set up, but it may be useful in certain setups. Arguably, as long as Krill listens on 127.0.0.1 only (as is the default), and an HTTPS enabled proxy server is used for public access, then having plain HTTP traffic between the proxy and Krill over the loopback interface is not necessarily problematic. To use this mode set `` `https_mode = "disable"` `` in your configuration file.

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

2.8.2 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.

2.8.3 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.

2.8.4 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

Before you can start to use Krill you will need to install, configure and run the Krill application somewhere. Please follow the steps below and you will be ready to *get started*.

3.1 Quick Start

Getting started with Krill is really easy by either installing a binary package for Debian and Ubuntu or for Red Hat Enterprise Linux and CentOS. You can also run with *Docker* or build from Cargo, Rust's build system and package manager.

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.

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.

Debian

Ubuntu

RHEL/CentOS

Cargo

If you have a machine with an amd64/x86_64 architecture running Debian 9, 10 or 11, you can install Krill from our software package repository.

If your machine uses an ARM architecture we also provide (via the same repository) ARMv6 & ARM64 packages for Debian 10 and an ARMv7 package for Debian 11, intended to support Raspberry Pi 1b, Rock64 and Raspberry Pi 4b respectively.

First update the apt package index:

```
sudo apt update
```

Then install packages to allow apt to use a repository over HTTPS:

```
sudo apt install \
  ca-certificates \
  curl \
  gnupg \
  lsb-release
```

Add the GPG key from NLnet Labs:

```
curl -fsSL https://packages.nlnetlabs.nl/aptkey.asc | sudo gpg --dearmor -o /usr/share/
↪keyrings/nlnetlabs-archive-keyring.gpg
```

Now, use the following command to set up the *main* repository:

```
echo \
"deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/nlnetlabs-archive-
↪keyring.gpg] https://packages.nlnetlabs.nl/linux/debian \
$(lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/nlnetlabs.list > /dev/null
```

After updating the `apt` package index you can install Krill:

```
sudo apt update
sudo apt install krill
```

Review the generated configuration file at `/etc/krill.conf`. **Pay particular attention** to the `service_uri` and `admin_token` settings. Tip: The configuration file was generated for you using the `krillc config simple` command.

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. The krill daemon runs as user `krill` and stores its data in `/var/lib/krill`.

You can check the status of Krill with:

```
sudo systemctl status krill
```

You can view the logs with:

```
sudo journalctl --unit=krill
```

If you have a machine with an amd64/x86_64 architecture running Ubuntu 16.x, 18.x, 20.x or 22.x, you can install Krill from our software package repository.

First update the `apt` package index:

```
sudo apt update
```

Then install packages to allow `apt` to use a repository over HTTPS:

```
sudo apt install \
  ca-certificates \
  curl \
  gnupg \
  lsb-release
```

Add the GPG key from NLnet Labs:

```
curl -fsSL https://packages.nlnetlabs.nl/aptkey.asc | sudo gpg --dearmor -o /usr/share/
↪keyrings/nlnetlabs-archive-keyring.gpg
```

Now, use the following command to set up the *main* repository:

```
echo \
"deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/nlnetlabs-archive-keyring.gpg] https://packages.nlnetlabs.nl/linux/ubuntu \
$(lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/nlnetlabs.list > /dev/null
```

After updating the `apt` package index you can install Krill:

```
sudo apt update
sudo apt install krill
```

Review the generated configuration file at `/etc/krill.conf`. **Pay particular attention** to the `service_uri` and `admin_token` settings. Tip: The configuration file was generated for you using the `krillc config simple` command.

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. The krill daemon runs as user `krill` and stores its data in `/var/lib/krill`.

You can check the status of Krill with:

```
sudo systemctl status krill
```

You can view the logs with:

```
sudo journalctl --unit=krill
```

If you have a machine with an amd64/x86_64 architecture running a RHEL (Red Hat Enterprise Linux)/CentOS 7 or 8 distribution, or a compatible OS such as Rocky Linux, you can install Krill from our software package repository.

To use this repository, create a file named `/etc/yum.repos.d/nlnetlabs.repo`, enter this configuration and save it:

```
[nlnetlabs]
name=NLnet Labs
baseurl=https://packages.nlnetlabs.nl/linux/centos/$releasever/main/$basearch
enabled=1
```

Then run the following command to add the public key:

```
sudo rpm --import https://packages.nlnetlabs.nl/aptkey.asc
```

You can then install Krill by running:

```
sudo yum install -y krill
```

Review the generated configuration file at `/etc/krill.conf`. **Pay particular attention** to the `service_uri` and `admin_token` settings. Tip: The configuration file was generated for you using the `krillc config simple` command.

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. The krill daemon runs as user `krill` and stores its data in `/var/lib/krill`.

You can check the status of Krill with:

```
sudo systemctl status krill
```

You can view the logs with:

```
sudo journalctl --unit=krill
```

Assuming you have a newly installed Debian or Ubuntu machine, you will need to install the C toolchain, OpenSSL and Rust. You can then install Krill using:

```
sudo apt install curl build-essential libssl-dev openssl pkg-config
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source ~/.cargo/env
cargo install --locked krill
```

3.2 Updating

Debian

Ubuntu

RHEL/CentOS

Cargo

To update an existing Krill installation, first update the repository using:

```
sudo apt update
```

You can use this command to get an overview of the available versions:

```
sudo apt policy krill
```

You can upgrade an existing Krill installation to the latest version using:

```
sudo apt --only-upgrade install krill
```

To update an existing Krill installation, first update the repository using:

```
sudo apt update
```

You can use this command to get an overview of the available versions:

```
sudo apt policy krill
```

You can upgrade an existing Krill installation to the latest version using:

```
sudo apt --only-upgrade install krill
```

To update an existing Krill installation, you can use this command to get an overview of the available versions:

```
sudo yum --showduplicates list krill
```

You can update to the latest version using:

```
sudo yum update -y krill
```

If you want to install the latest version of Krill using Cargo, it's recommended to also update Rust to the latest version first. Use the `--force` option to overwrite an existing version with the latest release:

```
rustup update
cargo install --locked --force krill
```

3.3 Installing Specific Versions

Before every new release of Krill, one or more release candidates are provided for testing through every installation method. You can also install a specific version, if needed.

Debian

Ubuntu

RHEL/CentOS

Cargo

If you would like to try out release candidates of Routinator you can add the *proposed* repository to the existing *main* repository described earlier.

Assuming you already have followed the steps to install regular releases, run this command to add the additional repository:

```
echo \
"deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/nlnetlabs-archive-
↪keyring.gpg] https://packages.nlnetlabs.nl/linux/debian \
$(lsb_release -cs)-proposed main" | sudo tee /etc/apt/sources.list.d/nlnetlabs-proposed.
↪list > /dev/null
```

Make sure to update the apt package index:

```
sudo apt update
```

You can now use this command to get an overview of the available versions:

```
sudo apt policy krill
```

You can install a specific version using <package name>=<version>, e.g.:

```
sudo apt install krill=0.9.0~rc2-1buster
```

If you would like to try out release candidates of Krill you can add the *proposed* repository to the existing *main* repository described earlier.

Assuming you already have followed the steps to install regular releases, run this command to add the additional repository:

```
echo \
"deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/nlnetlabs-archive-
↪keyring.gpg] https://packages.nlnetlabs.nl/linux/ubuntu \
$(lsb_release -cs)-proposed main" | sudo tee /etc/apt/sources.list.d/nlnetlabs-proposed.
↪list > /dev/null
```

Make sure to update the apt package index:

```
sudo apt update
```

You can now use this command to get an overview of the available versions:

```
sudo apt policy krill
```

You can install a specific version using `<package name>=<version>`, e.g.:

```
sudo apt install krill=0.9.0~rc2-1bionic
```

To install release candidates of Routinator, create an additional repo file named `/etc/yum.repos.d/nlnetlabs-testing.repo`, enter this configuration and save it:

```
[nlnetlabs-testing]
name=NLnet Labs Testing
baseurl=https://packages.nlnetlabs.nl/linux/centos/$releasever/proposed/$basearch
enabled=1
```

You can use this command to get an overview of the available versions:

```
sudo yum --showduplicates list krill
```

You can install a specific version using `<package name>-<version info>`, e.g.:

```
sudo yum install -y krill-0.9.0~rc2
```

All release versions of Krill, as well as release candidates, are available on crates.io, the Rust package registry. If you want to install a specific version of Krill using Cargo, explicitly use the `--version` option. If needed, use the `--force` option to overwrite an existing version:

```
cargo install --locked --force krill --version 0.9.0-rc2
```

All new features of Krill are built on a branch and merged via a pull request, allowing you to easily try them out using Cargo. If you want to try the a specific branch from the repository you can use the `--git` and `--branch` options:

```
cargo install --git https://github.com/NLnetLabs/krill.git --branch main
```

For more installation options refer to the Cargo book.

3.4 Installing with Cargo

There are three things you need for Krill: Rust, a 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.

3.4.1 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 1.45 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
```

3.4.2 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.

3.4.3 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
```

3.4.4 Building

The easiest way to get Krill v0.9.0 RC1 is to leave it to cargo by saying:

```
cargo install krill --git https://github.com/NLnetLabs/krill \
                    --tag v0.9.0-rc1 \
                    --locked
```

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.

3.5 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.

3.6 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 ROAs you need to do a one time setup where you:

- create your CA
- connect to Publication Server
- connect to 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* 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
```

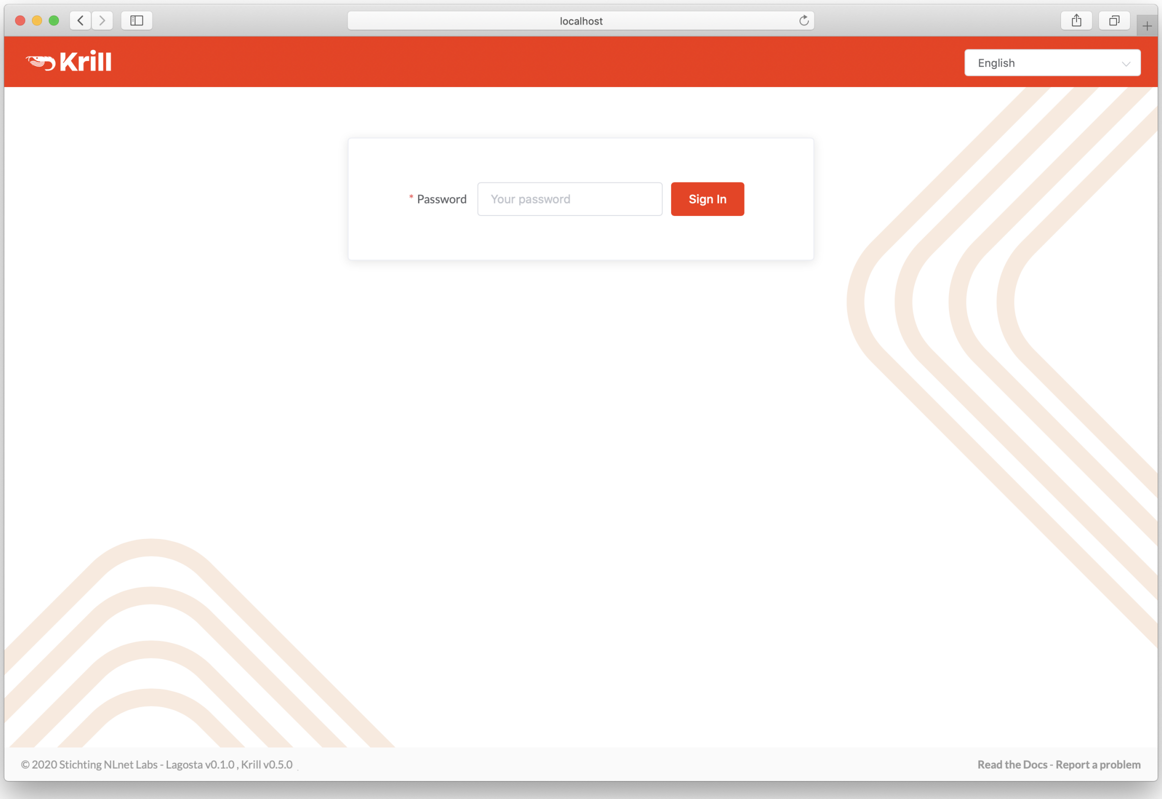
Here we will guide you through the set up process using the UI, but we will also link to the relevant subcommands of the *command line interface (CLI)*

4.1 Login

Tip: To login to the web user interface using named users instead of the secret token, see *Login with Named Users*.

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, or you set the the *KRILL_CLI_TOKEN* environment variable and save yourself the trouble of repeating it.



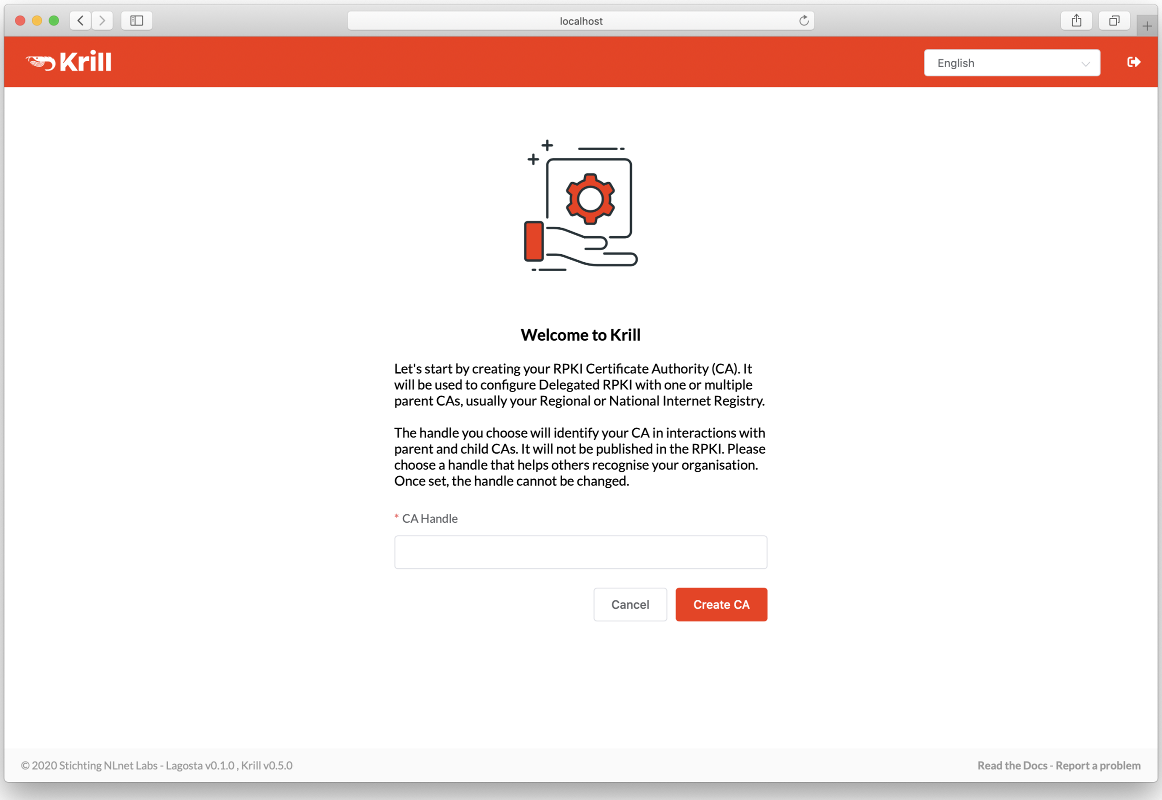
© 2020 Stichting NLnet Labs - Lagosta v0.1.0 , Krill v0.5.0

Fig. 1: Enter your secret token to access Krill

4.2 Create your Certification Authority

Next, you will see the Welcome screen where you can create your Certification 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.



Welcome to Krill

Let's start by creating your RPKI 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 choose will identify your CA in interactions with parent and child CAs. It will not be published in the RPKI. Please choose a handle that helps others recognise your organisation. Once set, the handle cannot be changed.

* CA Handle

© 2020 Stichting NLnet Labs - Lagosta v0.1.0 , Krill v0.5.0

Read the Docs - Report a problem

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 Repository Setup

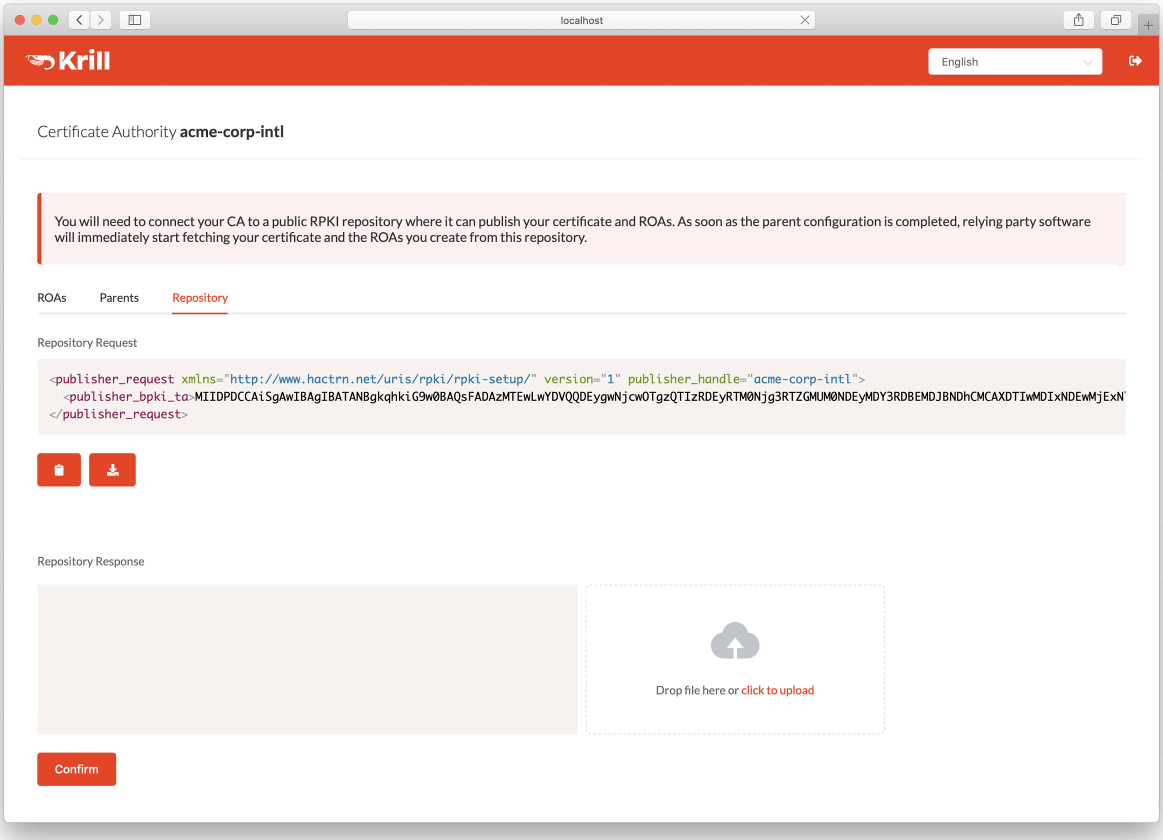
Note: If you are a member of NIC.BR, ARIN or APNIC, then you're in luck. These organisations provide an RPKI Publication Server as a service to their members, so you can configure your Krill CA publish there.

If you need to run your own Publication Server then please have a look *here* to see how you can use Krill to achieve this.

In either case the same process described below applies from your Krill CA's perspective.

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:



Certificate Authority **acme-corp-intl**

You will need to connect your CA to a public RPKI repository where it can publish your certificate and ROAs. As soon as the parent configuration is completed, relying party software will immediately start fetching your certificate and the ROAs you create from this repository.

Repository Request

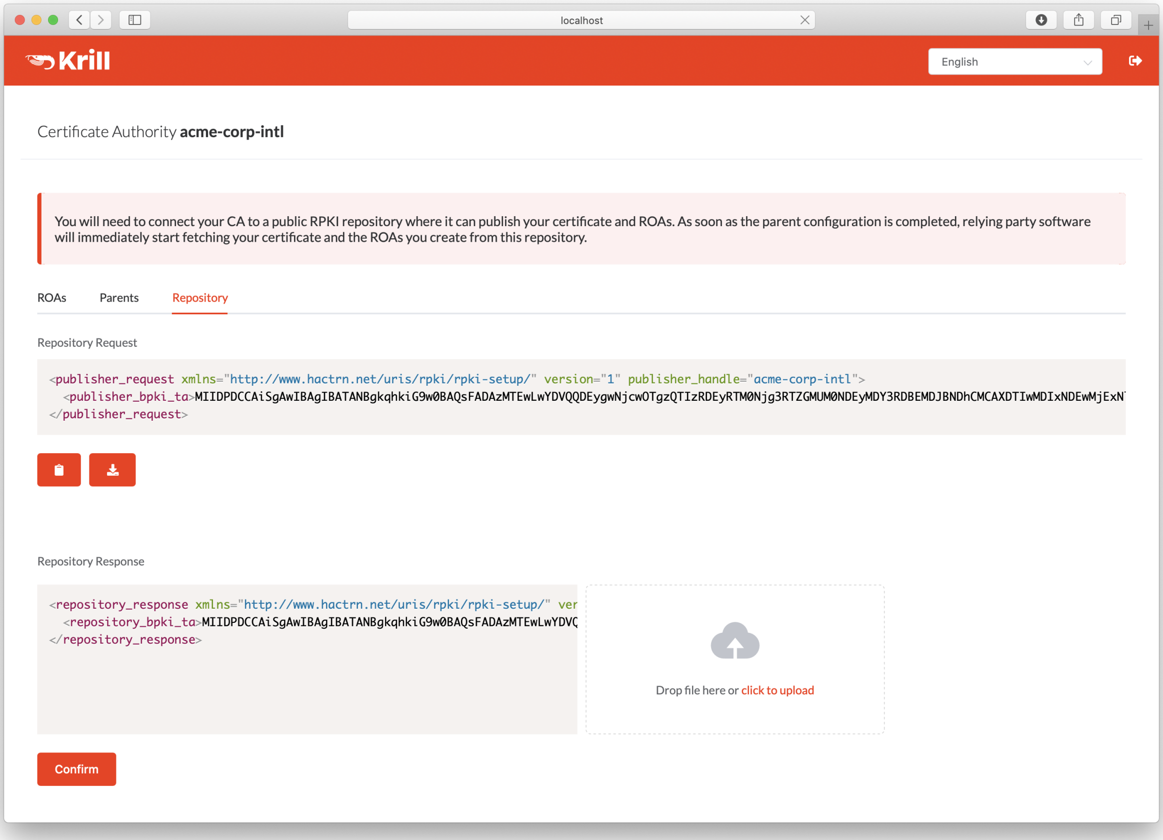
```
<publisher_request xmlns="http://www.hactrn.net/uris/rpki/rpki-setup/" version="1" publisher_handle="acme-corp-intl">
  <publisher_bpki_ta>MIIDPDCCAiSgAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEygwNjcwOTgzQTIzRDEyRTM0Njg3RTZGMUM0NDEyMDY3RDBEMDJBNDhCMCAXDTIwMDIxNDEwMjExN
</publisher_request>
```

Repository Response

Fig. 3: Copy the publisher request XML or download the file

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:

Note: Migrating to a new Repository later is not supported through the web UI, but *you can use the CLI to do this*.



Certificate Authority **acme-corp-intl**

You will need to connect your CA to a public RPKI repository where it can publish your certificate and ROAs. As soon as the parent configuration is completed, relying party software will immediately start fetching your certificate and the ROAs you create from this repository.

Repository Request

```
<publisher_request xmlns="http://www.hactrn.net/uris/rpki/rpki-setup/" version="1" publisher_handle="acme-corp-intl">
  <publisher_bpki_ta>MIIDPDCCAiSgAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEygwNjcwOTgzQTIzRDEyRTM0Njg3RTZGMUM0NDEyMDY3RDBEMDJBNDhCMCAXDTIwMDIxNDEwMjExN
</publisher_request>
```

Repository Response

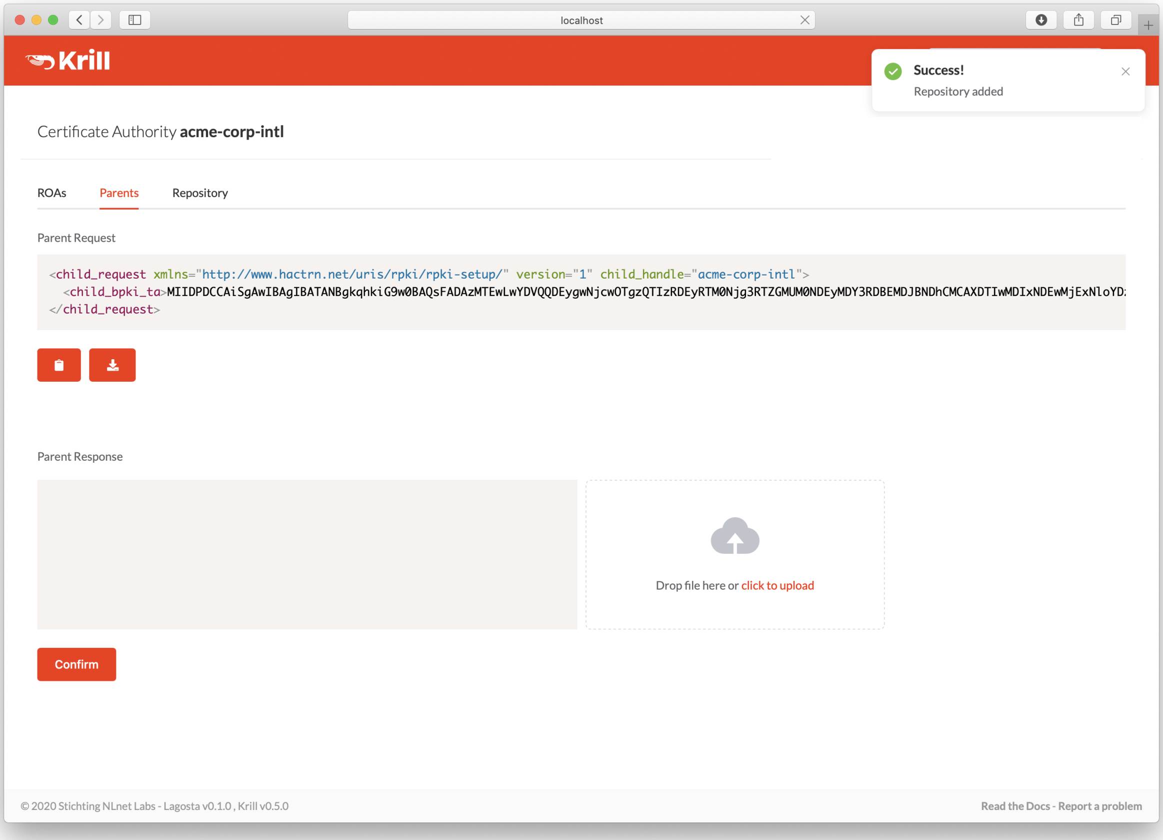
```
<repository_response xmlns="http://www.hactrn.net/uris/rpki/rpki-setup/" ver
  <repository_bpki_ta>MIIDPDCCAiSgAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQ
</repository_response>
```

Drop file here or click to upload

Fig. 4: Paste or upload the repository response XML

4.4 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:



Success!
Repository added

Certificate Authority **acme-corp-intl**

Parent Request

```
<child_request xmlns="http://www.hactrn.net/uris/rpki/rpki-setup/" version="1" child_handle="acme-corp-intl">
  <child_bpki_ta>MIIDPDCCAiSgAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEygwNjcwOTgzQTIzRDEyRTM0Njg3RTZGMUM0NDEyMDY3RDBEMDJBNDhCMCAXDTIwMDIxNDEwMjExNloYD:
</child_request>
```

Parent Response

Drop file here or click to upload

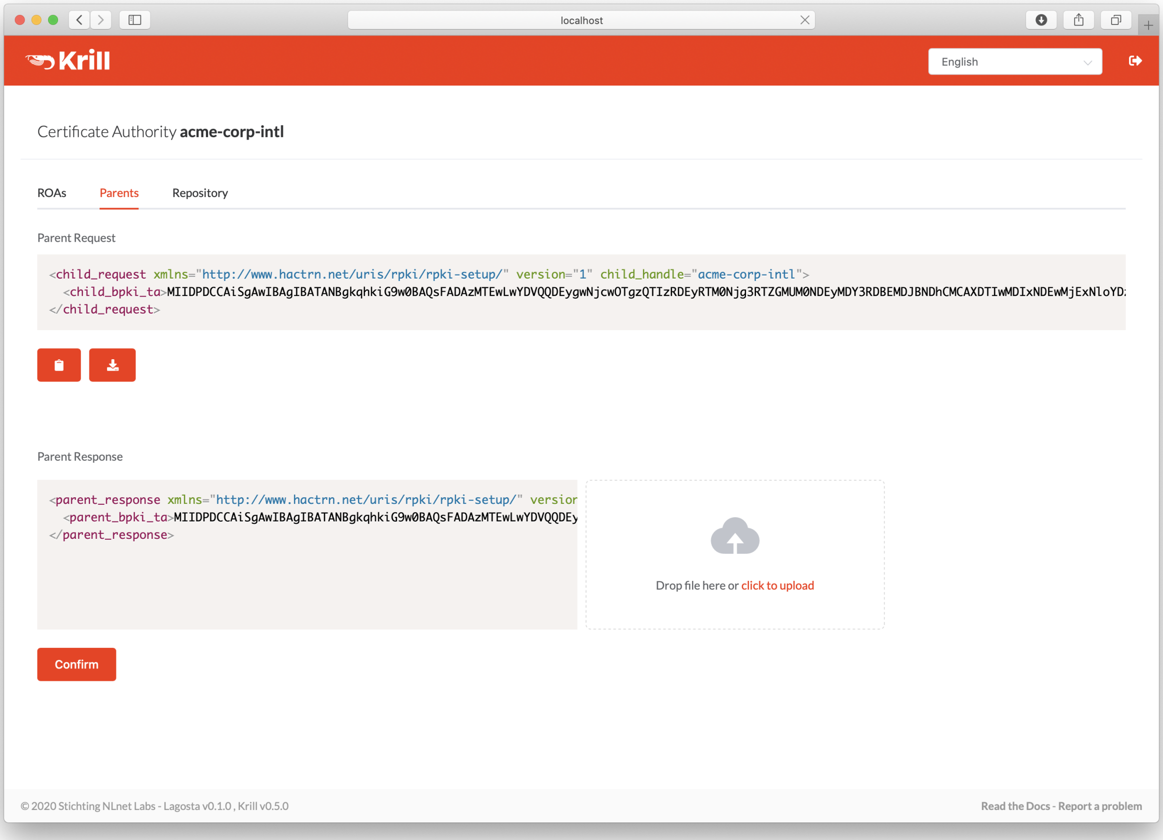
Confirm

© 2020 Stichting NLnet Labs - Lagosta v0.1.0 , Krill v0.5.0

Read the Docs - Report a problem

Fig. 5: Copy the child request XML or download the file

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:



Certificate Authority **acme-corp-intl**

Parent Request

```
<child_request xmlns="http://www.hactrn.net/uris/rpki/rpki-setup/" version="1" child_handle="acme-corp-intl">
  <child_bpki_ta>MIIDPDCCAiSgAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEygwNjcwOTgzQTIzRDEyRTM0Njg3RTZGMUM0NDEyMDY3RDBEMDJBNDhCMCAXDTIwMDIxNDEwMjExNloYD:
</child_request>
```

Parent Response

```
<parent_response xmlns="http://www.hactrn.net/uris/rpki/rpki-setup/" versior
  <parent_bpki_ta>MIIDPDCCAiSgAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEy
</parent_response>
```

© 2020 Stichting NLnet Labs - Lagosta v0.1.0 , Krill v0.5.0

Fig. 6: Paste or upload the parent response XML

CHAPTER

FIVE

RIR AND NIR INTERACTIONS

In almost all cases, you will interact with one or more Regional Internet Registries (RIRs) or National Internet Registries (NIRs) when setting up delegated RPKI.

The fundamental principle is the same with each of them: the RIR or NIR needs to establish who you are, which resources you are entitled to and where your RPKI certificate and ROAs will be published.

Your identity, permissions and entitlements are all managed by the registry and exposed via their respective member portals. The rest of the information is exchanged in two XML files. You will need to provide a child request generated by Krill, and in return you will receive a parent response that you need to give back to Krill. See *Parent Setup* for more details.

5.1 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. To use this service you will need to do an additional exchange. You need to generate and provide a publisher request file and in return you will receive a repository response.

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, APNIC and ARIN currently offer 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. If you need to publish your certificate and ROAs yourself, follow the steps described in the *Running a Publication Server* section.

5.2 Member Portals

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.

5.2.1 AFRINIC

AFRINIC have delegated RPKI available in their test environment, but it's not operational yet.

5.2.2 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. It is highly recommended to make use of this, as it relieves you of the need to run a highly available repository yourself. We provide a step-by-step guide to set this up.

5.2.3 ARIN

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

Please note that ARIN offers RPKI publication as a service. It is highly recommended to make use of this, as it relieves you of the need to run a highly available repository yourself. We provide a step-by-step guide to set this up.

5.2.4 LACNIC

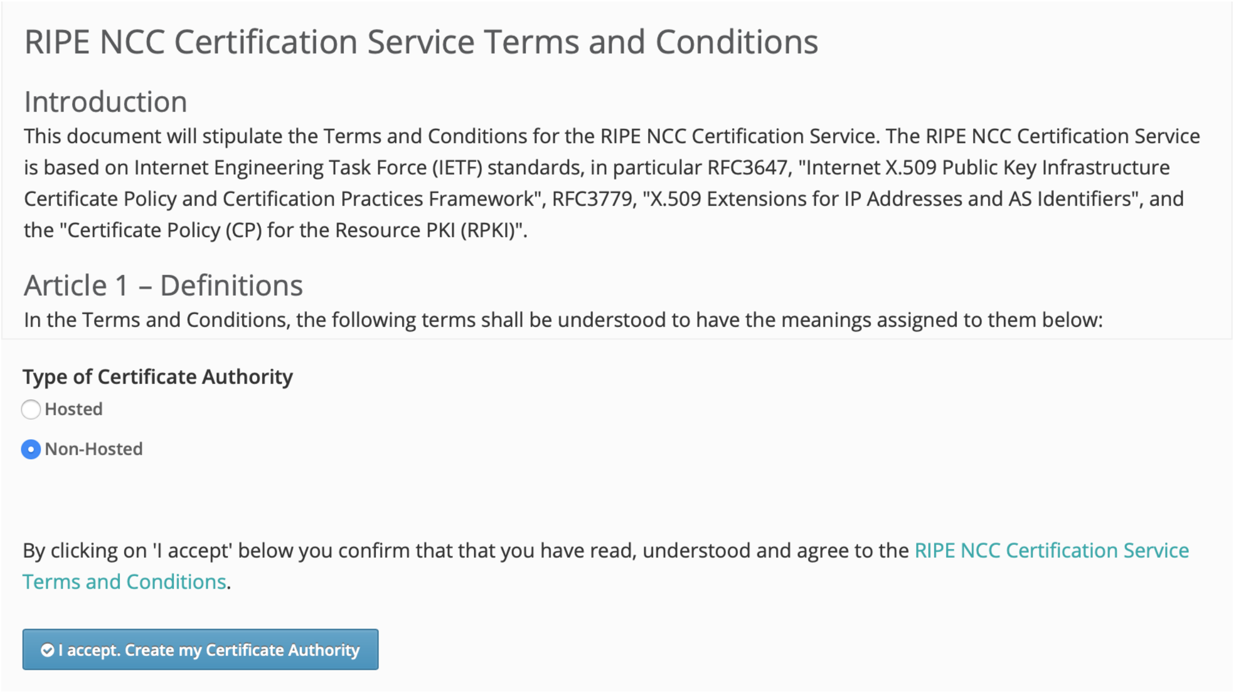
Although LACNIC offers delegated RPKI, it is not possible to configure this in their member portal yet. While the procedures are still being defined, please open a ticket via hostmaster@lacnic.net to get started.

5.2.5 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.

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.



RIPE NCC Certification Service Terms and Conditions

Introduction

This document will stipulate the Terms and Conditions for the RIPE NCC Certification Service. The RIPE NCC Certification Service is based on Internet Engineering Task Force (IETF) standards, in particular RFC3647, "Internet X.509 Public Key Infrastructure Certificate Policy and Certification Practices Framework", RFC3779, "X.509 Extensions for IP Addresses and AS Identifiers", and the "Certificate Policy (CP) for the Resource PKI (RPKI)".

Article 1 – Definitions

In the Terms and Conditions, the following terms shall be understood to have the meanings assigned to them below:

Type of Certificate Authority

○ Hosted

● Non-Hosted

By clicking on 'I accept' below you confirm that that you have read, understood and agree to the RIPE NCC Certification Service Terms and Conditions.

Fig. 1: RIPE NCC RPKI setup screen

CHAPTER

SIX

MANAGE ROAS

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

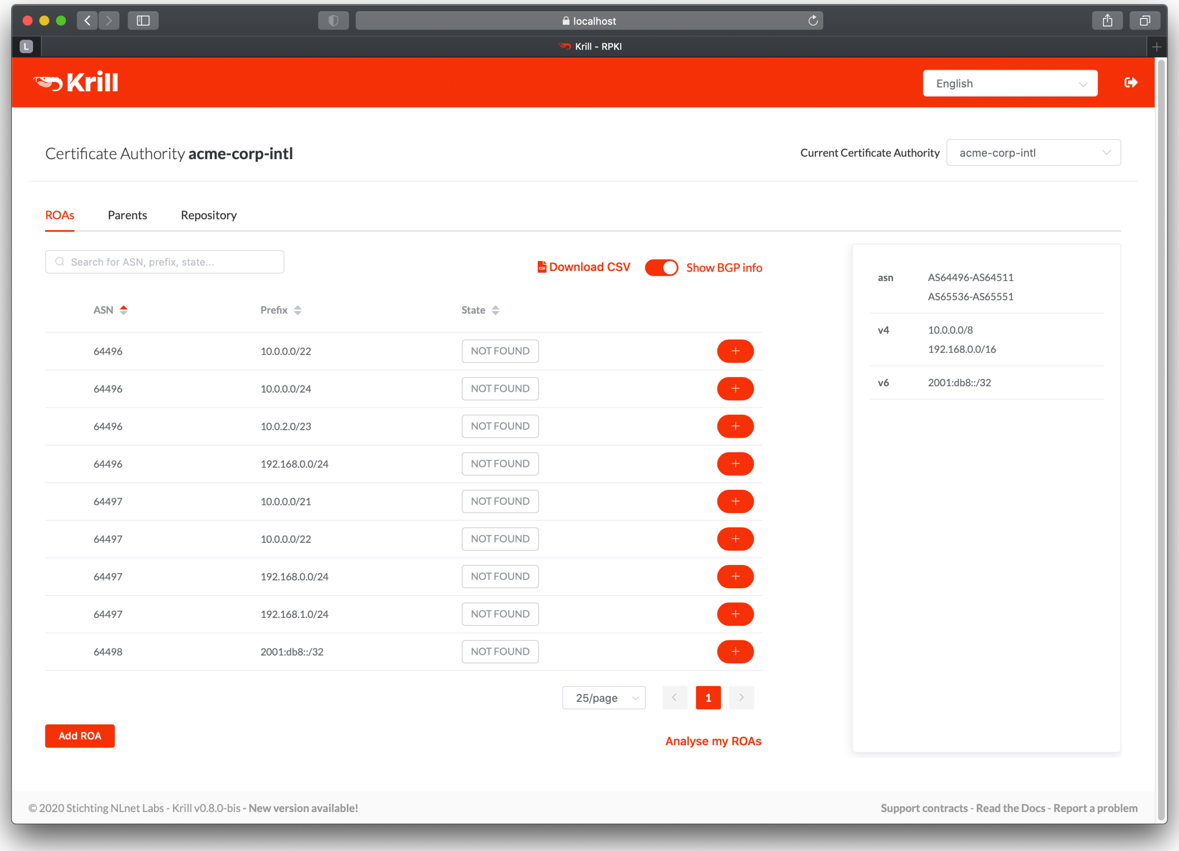
6.1 Show BGP Info

Krill automatically downloads BGP announcement information from RIPE 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 do some suggestions on ROAs you may want to create or remove.

Krill recognises the following ‘States’ in its analysis:

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 just set up your Krill instance you will see that your announcements all have the status `NOT FOUND`, meaning that you have not created any ROAs covering them yet.



Certificate Authority **acme-corp-intl**

Current Certificate Authority acme-corp-intl

ASN	Prefix	State	
64496	10.0.0.0/22	NOT FOUND	+
64496	10.0.0.0/24	NOT FOUND	+
64496	10.0.2.0/23	NOT FOUND	+
64496	192.168.0.0/24	NOT FOUND	+
64497	10.0.0.0/21	NOT FOUND	+
64497	10.0.0.0/22	NOT FOUND	+
64497	192.168.0.0/24	NOT FOUND	+
64497	192.168.1.0/24	NOT FOUND	+
64498	2001:db8::/32	NOT FOUND	+

asn	AS64496-AS64511 AS65536-AS65551
v4	10.0.0.0/8 192.168.0.0/16
v6	2001:db8::/32

© 2020 Stichting NLnet Labs - Krill v0.8.0-bis - **New version available!**

Support contracts - Read the Docs - Report a problem

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

6.2 ROA Suggestions

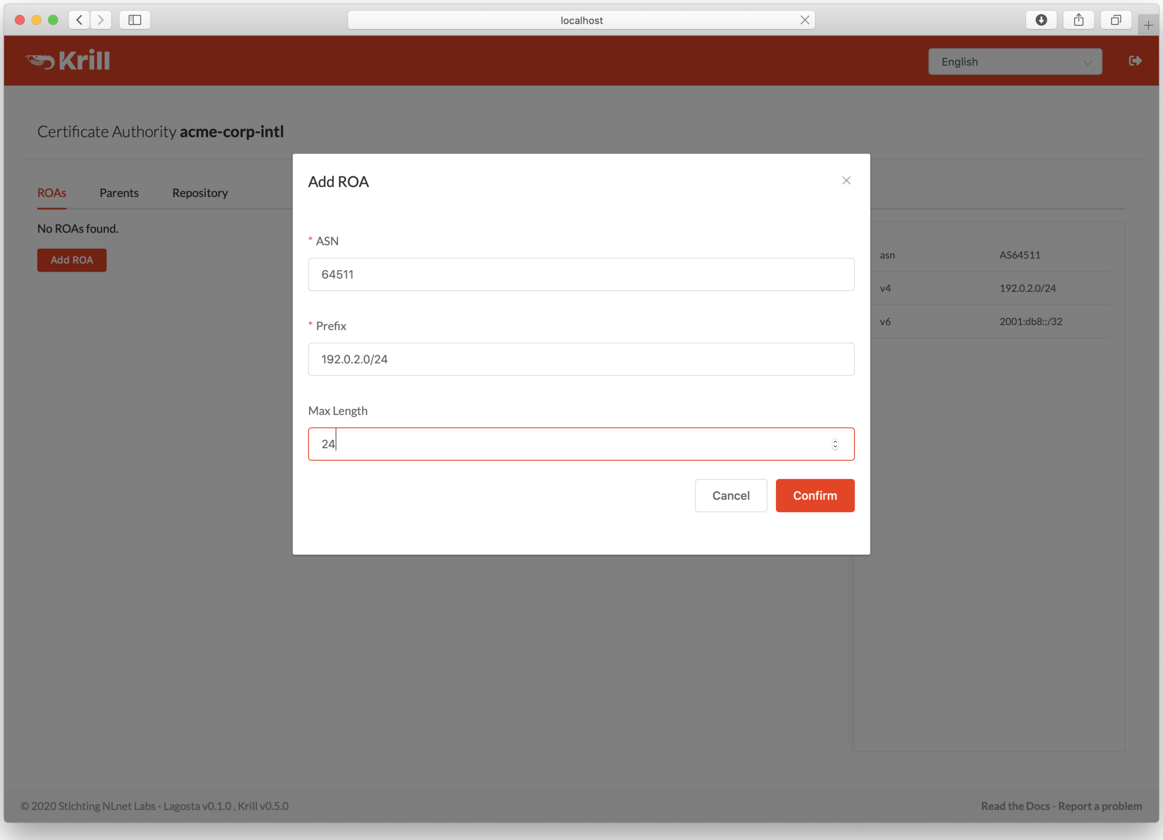
> **Warning:** You should **always** verify the suggestions done by Krill. Krill bases its analysis on information collected by the RIPE NCC Routing Information Service (RIS) and saved in aggregated dumps every 8 hours. So, the announcements that Krill sees may be outdated. More importantly they may include announcements by others that you do **NOT** wish to allow. And 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	
PREFIX REMOVED	Remove	Keep the ROA if you believe that

6.3 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.



Certificate Authority **acme-corp-intl**

No ROAs found.

Add ROA

* ASN: 64511

* Prefix: 192.0.2.0/24

Max Length: 24

asn	AS64511
v4	192.0.2.0/24
v6	2001:db8::/32

© 2020 Stichting NLnet Labs - Lagosta v0.1.0 , Krill v0.5.0

Fig. 2: Adding a new ROA

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

6.4 Disable BGP Info

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
```

CHAPTER

SEVEN

MONITORING

7.1 Prometheus

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 a lot of details. Generating these metrics is not particularly hard on Krill, but in case you have many CAs, children or publishers under your Krill instance you may still want to disable certain metrics to reduce the amount of data fetched and stored by Prometheus.

7.1.1 General Metrics

The following are always enabled:

```
# HELP krill_server_start unix timestamp in seconds of last krill server start
# TYPE krill_server_start gauge
krill_server_start 1631542209

# HELP krill_version_major krill server major version number
# TYPE krill_version_major gauge
krill_version_major 0

# HELP krill_version_minor krill server minor version number
# TYPE krill_version_minor gauge
krill_version_minor 9

# HELP krill_version_patch krill server patch version number
# TYPE krill_version_patch gauge
krill_version_patch 2

# HELP krill_auth_session_cache_size total number of cached login session tokens
# TYPE krill_auth_session_cache_size gauge
krill_auth_session_cache_size 0

# HELP krill_cas number of cas in krill
# TYPE krill_cas gauge
krill_cas 6
```

7.1.2 CA Metrics

There are a number of metrics which use a label like {ca="ca_name"}. You can disable all of them by setting the following in your configuration file:

```
metrics_hide_ca_details = true
```

Example:

```
# HELP krill_ca_parent_success status of last CA to parent connection (0=issue,␣
↪1=success)
# TYPE krill_ca_parent_success gauge
krill_ca_parent_success{ca="CA1", parent="testbed"} 1
krill_ca_parent_success{ca="ca", parent="testbed"} 1
krill_ca_parent_success{ca="CA2", parent="testbed"} 1
krill_ca_parent_success{ca="testbed", parent="ta"} 1
krill_ca_parent_success{ca="dummy_ca", parent="testbed"} 1

# HELP krill_ca_parent_last_success_time unix timestamp in seconds of last successful CA␣
↪to parent connection
# TYPE krill_ca_parent_last_success_time gauge
krill_ca_parent_last_success_time{ca="CA1", parent="testbed"} 1631542800
krill_ca_parent_last_success_time{ca="ca", parent="testbed"} 1631542800
krill_ca_parent_last_success_time{ca="CA2", parent="testbed"} 1631542800
krill_ca_parent_last_success_time{ca="testbed", parent="ta"} 1631542800

# HELP krill_ca_ps_success status of last CA to Publication Server connection (0=issue,␣
↪1=success)
# TYPE krill_ca_ps_success gauge
krill_ca_ps_success{ca="CA1"} 1
krill_ca_ps_success{ca="ca"} 1
krill_ca_ps_success{ca="CA2"} 1
krill_ca_ps_success{ca="ta"} 1
krill_ca_ps_success{ca="testbed"} 1
krill_ca_ps_success{ca="dummy_ca"} 0

# HELP krill_ca_ps_last_success_time unix timestamp in seconds of last successful CA to␣
↪Publication Server connection
# TYPE krill_ca_ps_last_success_time gauge
krill_ca_ps_last_success_time{ca="CA1"} 1631542801
krill_ca_ps_last_success_time{ca="ca"} 1631542802
krill_ca_ps_last_success_time{ca="CA2"} 1631542802
krill_ca_ps_last_success_time{ca="ta"} 1631542801
krill_ca_ps_last_success_time{ca="testbed"} 1631542802

# HELP krill_ca_ps_next_planned_time unix timestamp in seconds of next planned CA to␣
↪Publication Server connection (unless e.g. ROAs are changed)
# TYPE krill_ca_ps_next_planned_time gauge
krill_ca_ps_next_planned_time{ca="CA1"} 1631600401
krill_ca_ps_next_planned_time{ca="ca"} 1631600402
krill_ca_ps_next_planned_time{ca="CA2"} 1631600402
krill_ca_ps_next_planned_time{ca="ta"} 1631600401
krill_ca_ps_next_planned_time{ca="testbed"} 1631600402
krill_ca_ps_next_planned_time{ca="dummy_ca"} 1631543137
```

7.1.3 Child metrics

NOTE: These metrics are only shown if you have any child CAs under your CA(s) in Krill.

By default Krill will also show metrics on child CAs for each CA. If you left the showing CA details enabled, but you wish to hide these details then you can do so by setting the following directive in your configuration file:

```
metrics_hide_child_details = true
```

Example:

```
# HELP krill_ca_child_success status of last child to CA connection (0=issue, 1=success)
# TYPE krill_ca_child_success gauge
krill_ca_child_success{ca="ta", child="testbed"} 1
krill_ca_child_success{ca="testbed", child="ca"} 1
krill_ca_child_success{ca="testbed", child="CA1"} 1
krill_ca_child_success{ca="testbed", child="CA2"} 1

# HELP krill_ca_child_state child state (see 'suspend_child_after_inactive_hours'␣
↪config) (0=suspended, 1=active)
# TYPE krill_ca_child_state gauge
krill_ca_child_state{ca="ta", child="testbed"} 0
krill_ca_child_state{ca="testbed", child="ca"} 0
krill_ca_child_state{ca="testbed", child="CA1"} 0
krill_ca_child_state{ca="testbed", child="CA2"} 0

# HELP krill_ca_child_last_connection unix timestamp in seconds of last child to CA␣
↪connection
# TYPE krill_ca_child_last_connection gauge
krill_ca_child_last_connection{ca="ta", child="testbed"} 1631542800
krill_ca_child_last_connection{ca="testbed", child="ca"} 1631542800
krill_ca_child_last_connection{ca="testbed", child="CA1"} 1631542800
krill_ca_child_last_connection{ca="testbed", child="CA2"} 1631542800

# HELP krill_ca_child_last_success unix timestamp in seconds of last successful child to␣
↪CA connection
# TYPE krill_ca_child_last_success gauge
krill_ca_child_last_success{ca="ta", child="testbed"} 1631542800
krill_ca_child_last_success{ca="testbed", child="ca"} 1631542800
krill_ca_child_last_success{ca="testbed", child="CA1"} 1631542800
krill_ca_child_last_success{ca="testbed", child="CA2"} 1631542800

# HELP krill_ca_child_agent_total total children per user agent based on their last␣
↪connection
# TYPE krill_ca_child_agent_total gauge
krill_ca_child_agent_total{ca="ta", user_agent="krill/0.9.2-rc1"} 1
krill_ca_child_agent_total{ca="testbed", user_agent="krill/0.9.2-rc1"} 3
```

7.1.4 ROA Metrics

By default Krill will also show metrics on ROAs in relation to known BGP announcements for each CA. If you left the showing CA details enabled, but you wish to hide these details then you can do so by setting the following directive in your configuration file:

```
metrics_hide_roa_details = true
```

Example:

```
# HELP krill_cas_bgp_announcements_valid number of announcements seen for CA resources␣
↪with RPKI state VALID
# TYPE krill_cas_bgp_announcements_valid gauge
krill_cas_bgp_announcements_valid{ca="CA1"} 0
krill_cas_bgp_announcements_valid{ca="ca"} 2
krill_cas_bgp_announcements_valid{ca="CA2"} 0
krill_cas_bgp_announcements_valid{ca="testbed"} 0
krill_cas_bgp_announcements_valid{ca="ta"} 0
krill_cas_bgp_announcements_valid{ca="dummy_ca"} 0

# HELP krill_cas_bgp_announcements_invalid_asn number of announcements seen for CA␣
↪resources with RPKI state INVALID (ASN mismatch)
# TYPE krill_cas_bgp_announcements_invalid_asn gauge
krill_cas_bgp_announcements_invalid_asn{ca="dummy_ca"} 0
krill_cas_bgp_announcements_invalid_asn{ca="testbed"} 0
krill_cas_bgp_announcements_invalid_asn{ca="CA2"} 0
krill_cas_bgp_announcements_invalid_asn{ca="CA1"} 0
krill_cas_bgp_announcements_invalid_asn{ca="ta"} 0
krill_cas_bgp_announcements_invalid_asn{ca="ca"} 1

# HELP krill_cas_bgp_announcements_invalid_length number of announcements seen for CA␣
↪resources with RPKI state INVALID (prefix exceeds max length)
# TYPE krill_cas_bgp_announcements_invalid_length gauge
krill_cas_bgp_announcements_invalid_length{ca="testbed"} 0
krill_cas_bgp_announcements_invalid_length{ca="dummy_ca"} 0
krill_cas_bgp_announcements_invalid_length{ca="ta"} 0
krill_cas_bgp_announcements_invalid_length{ca="CA2"} 0
krill_cas_bgp_announcements_invalid_length{ca="ca"} 0
krill_cas_bgp_announcements_invalid_length{ca="CA1"} 0

# HELP krill_cas_bgp_announcements_not_found number of announcements seen for CA␣
↪resources with RPKI state NOT FOUND (none of the CA's ROAs cover this)
# TYPE krill_cas_bgp_announcements_not_found gauge
krill_cas_bgp_announcements_not_found{ca="CA2"} 0
krill_cas_bgp_announcements_not_found{ca="ta"} 0
krill_cas_bgp_announcements_not_found{ca="ca"} 0
krill_cas_bgp_announcements_not_found{ca="dummy_ca"} 0
krill_cas_bgp_announcements_not_found{ca="CA1"} 5
krill_cas_bgp_announcements_not_found{ca="testbed"} 0

# HELP krill_cas_bgp_roas_too_permissive number of ROAs for this CA which allow excess␣
↪announcements (0 may also indicate that no BGP info is available)
# TYPE krill_cas_bgp_roas_too_permissive gauge
krill_cas_bgp_roas_too_permissive{ca="ca"} 0
```

(continues on next page)

(continued from previous page)

```
krill_cas_bgp_roas_too_permissive{ca="testbed"} 0
krill_cas_bgp_roas_too_permissive{ca="CA1"} 0
krill_cas_bgp_roas_too_permissive{ca="dummy_ca"} 0
krill_cas_bgp_roas_too_permissive{ca="ta"} 0
krill_cas_bgp_roas_too_permissive{ca="CA2"} 0

# HELP krill_cas_bgp_roas_redundant number of ROAs for this CA which are redundant (0␣
↪may also indicate that no BGP info is available)
# TYPE krill_cas_bgp_roas_redundant gauge
krill_cas_bgp_roas_redundant{ca="ta"} 0
krill_cas_bgp_roas_redundant{ca="testbed"} 0
krill_cas_bgp_roas_redundant{ca="ca"} 0
krill_cas_bgp_roas_redundant{ca="dummy_ca"} 0
krill_cas_bgp_roas_redundant{ca="CA1"} 0
krill_cas_bgp_roas_redundant{ca="CA2"} 0

# HELP krill_cas_bgp_roas_stale number of ROAs for this CA for which no announcements␣
↪are seen (0 may also indicate that no BGP info is available)
# TYPE krill_cas_bgp_roas_stale gauge
krill_cas_bgp_roas_stale{ca="CA1"} 0
krill_cas_bgp_roas_stale{ca="CA2"} 0
krill_cas_bgp_roas_stale{ca="ca"} 0
krill_cas_bgp_roas_stale{ca="ta"} 0
krill_cas_bgp_roas_stale{ca="testbed"} 0
krill_cas_bgp_roas_stale{ca="dummy_ca"} 0

# HELP krill_cas_bgp_roas_total total number of ROAs for this CA
# TYPE krill_cas_bgp_roas_stale gauge
krill_cas_bgp_roas_total{ca="dummy_ca"} 0
krill_cas_bgp_roas_total{ca="ca"} 3
krill_cas_bgp_roas_total{ca="CA1"} 0
krill_cas_bgp_roas_total{ca="ta"} 0
krill_cas_bgp_roas_total{ca="testbed"} 0
krill_cas_bgp_roas_total{ca="CA2"} 0
```

7.1.5 Publication Server Metrics

The following metrics are always enabled if you have an active Publication Server:

```
# HELP krill_repo_publisher number of publishers in repository
# TYPE krill_repo_publisher gauge
krill_repo_publisher 6

# HELP krill_repo_rrdp_last_update unix timestamp in seconds of last update by any␣
↪publisher
# TYPE krill_repo_rrdp_last_update gauge
krill_repo_rrdp_last_update 1631542802

# HELP krill_repo_rrdp_serial RRDP serial
# TYPE krill_repo_rrdp_serial counter
krill_repo_rrdp_serial 5
```

By default per publisher (publishing CA) metrics are also included, this can be disabled by setting the following directive in your configuration file:

```
metrics_hide_publisher_details = true
```

Example:

```
# HELP krill_repo_objects number of objects in repository for publisher
# TYPE krill_repo_objects gauge
krill_repo_objects{publisher="ta"} 3
krill_repo_objects{publisher="mos-eisley"} 4
krill_repo_objects{publisher="testbed"} 5
krill_repo_objects{publisher="CA1"} 2
krill_repo_objects{publisher="CA2"} 2
krill_repo_objects{publisher="dummy_ca"} 0

# HELP krill_repo_size size of objects in bytes in repository for publisher
# TYPE krill_repo_size gauge
krill_repo_size{publisher="ta"} 7592
krill_repo_size{publisher="mos-eisley"} 10056
krill_repo_size{publisher="testbed"} 9988
krill_repo_size{publisher="CA1"} 3744
krill_repo_size{publisher="CA2"} 3744
krill_repo_size{publisher="dummy_ca"} 0

# HELP krill_repo_last_update unix timestamp in seconds of last update for publisher
# TYPE krill_repo_last_update gauge
krill_repo_last_update{publisher="ta"} 1631542801
krill_repo_last_update{publisher="mos-eisley"} 1631542802
krill_repo_last_update{publisher="testbed"} 1631542802
krill_repo_last_update{publisher="CA1"} 1631542801
krill_repo_last_update{publisher="CA2"} 1631542802
krill_repo_last_update{publisher="dummy_ca"} 1628062124
```

7.2 Stats Endpoints

The monitoring service has a number of additional endpoints which can be accessed without the need for authentication on the following paths:

/stats/info Returns the Krill version and timestamp when the daemon was started.

/stats/cas Returns stats on your CAs, including an analysis of ROA configurations based on known BGP announcements.

/stats/repo Returns stats on the repository, if enabled. This includes publisher stats: number and size of objects and last connection time.

CHAPTER

EIGHT

USING THE CLI OR API

8.1 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
```

8.2 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.

8.3 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 *Running a Krill Test Environment*.

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

8.4 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.

8.5 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
```

8.6 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
}
```

8.7 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:
{
  "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"}}
```

8.8 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"}}
```

8.9 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"
    }
  ]
}
```

8.10 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

8.11 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>
↪MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyhFRjJENzgwRkNCRkU1QjZBMkExMjA1OUM0MDlDN0M5M
↪X3Ho5gC7uuKf4LYqbJj+Z9ltr/236/
↪hDYJfWMXZVcEuL+wUble1zhe2NKrgnAkpReVMSdiugoqZ9ICK2Fwkj5jCGc/qHiWOba7T78zfij8OlB/
↪dGlJvkAY8b/XTNKsTrLozi1uVAC8GqDrV5MEgY/NfzUvgA024yxx/rC6QBDEoBjnP7wDFiaZ2lwvL2beVYu6/
↪hVcXQzsVN+ijy7cGdkE6zi0meXJLTHPEpoA88hi3Pi+pIDBIQ3wTcpQIOqAq/
↪SZuh4dbZK7BV8MCAwEAAaNTMFEwDwYDVR0TAQH/BAUwAwEB/
↪zAdBgNVHQ4EFgQU7y14D8v+W2oqEgWcQJx8kndUGVYwHwYDVR0jBBgwFoAU7y14D8v+W2oqEgWcQJx8kndUGVYwDQYJKoZIhvcNAQ
↪gpJtONdgIWV1EqwEzhKKA2EP6tLDF9ejsdMFNYrYr+2hVWaoLsSuarfwfLFSgKDFqR6sh3ljYq6mIz9gdkjBOJsR9JyHFEtsDsRpf
↪YPMPVBpmG15Z9iKantzC1tck+E1xYW5awvj+YZqGVqyFdPJOZWmaYoS83kWvg4g4IucXTH6wwy23MQ7+0gyoK4wxfXRQmWjlXpLue
↪d7srQA4IxZCRGh9B+JdAIOKuXBA0nncmMJLQN8Qpxlz2bxKKAgXBLdoDqjbTDVbXTPM8YLRgc=</child_bpki_
↪ta>
</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
```

8.12 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":
↪"MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyhFOTBDMjE3MzRDMkMzNzBBOTFBODQ3NUNCNEYwRTc1
↪vVYxq1F1w2yQ/
↪VoTr1dvEHxJ+SDayMcFVktWCObiY8tcPhvWG+OdaX9ckDJhsOEEvdVEogwiGacNs7yXJPbqDBptJtbR8/
↪CauF9OqMqjkB/8xkGmBoY5OI/
↪V2832jkp7LPsbyET0RMQN7fgSpGbewvkaZVxGU3pHh5kT1nzPTXrwjxNMXgpunSEY7zR20vYCvsYYbxnSwFNbSMSL+Jgpa+HWPUc0
↪qlqlZK+tkppI2LkSBhTV5+n7cGA8ZsCAwEAAaNTMFEwDwYDVR0TAQH/BAUwAwEB/
↪zAdBgNVHQ4EFgQU6Qwhc0wsNwqRqEdctPDnXaTQ8L8wHwYDVR0jBBgwFoAU6Qwhc0wsNwqRqEdctPDnXaTQ8L8wDQYJKoZIhvcNAQ
↪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"
  }
}
```

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.

8.13 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
Resource Entitlements: asn: AS65000, ipv4: 10.0.0.0/8, ipv6: 2001:db8::/32
  resource class: 0
  entitled resources: asn: 'AS65000', ipv4: '10.0.0.0/8', ipv6: '2001:db8::/32'
  entitled not after: 2023-03-15T14:23:57+00:00
  issuing cert uri: rsync://localhost/repo/ta/0/0BA5C132B94891CB2D3A89EDE12F01ACA4BCD3DC.
↪cer
  issuing cert PEM:

-----BEGIN CERTIFICATE-----
MIIFKzCCBBOgAwIBA...
-----END CERTIFICATE-----

  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-----
MIIFYDCCBEigAwIBAgIUN5PzATTKVrjgual4CpJMaggW2EIwDQYJKoZIhvcNAQELBQAwMzExMC8GA1UEAxMoMEJBNUMxMzJCOTQ4OTF
↪7OHN8TU6crIu1/
↪w1gkf6UCXFrv+poW9EJHnLonMa4ZFLSFsvQACIGUpxIuiQjaSYFltTbb+o2c9KWoKsX0kZqt5zOrgAP8cke8SFGHdqqenPInXKTgy
↪ngql4OV0bLkbb63J/26c8FZOThZAgMBAAGjggJqMIICZjAPBgNVHRMBAf8EBTADAQH/
↪MB0GA1UdDgQWBBQWsxyS6xFrxgAmxQlErUQgXdmsvTAfBgNVHSMEGDAWgBQLpcEyuUiRyy06ie3hLwGspLzT3DAOBgNVHQ8BAf8EB
↪BA4wDDAKBggrBgEFBQcOAjAsBggrBgEFBQcBBwEB/
↪wQdMBswCgQCAAEwBAMCAAowDQQCAAIwBwMFACABDbgwGgYIKwYBBQUHAQgBAf8ECzAJoAcwBQIDAP3oMA0GCSqGSIb3DQEBCwUAA4
↪Yja+sYhyg/pG1/
↪ZTvhOLIxGWap8JmqOnYa9XgX8uUlsV8LgJoEH3Gde3txcGtfLO99ugvbnKKGOcPxB8AX5hAhhfdiSnt3V06dEz3HUoTYdUKTV0bZr
↪K8349vN0QJUZGm3gAUsM5PlnAqbkM7VFIyu8g2Yp9g+M/
↪iwaHar8CqABKxLBThYgqrPLLv6CsZD3mjk5BkXVZh6R9dBcR7sPbSfGBWPWCv8SwLknyQDOvsWTho1Ga6AibjUQp
-----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"
    },
    "last_success": 1617881400,
    "all_resources": {
      "asn": "AS65000",
      "ipv4": "10.0.0.0/8",
      "ipv6": "2001:db8::/32"
    },
    "classes": [
      {
        "class_name": "0",
        "resource_set": {
          "asn": "AS65000",
          "ipv4": "10.0.0.0/8",
          "ipv6": "2001:db8::/32"
        },
        "not_after": "2023-03-15T14:23:57Z",
        "issued_certs": [
          {
            "uri": "rsync://localhost/repo/testbed/0/
↪16B31C92EB116BC60026C50944AD44205DD9ACBD.cer",
            "req_limit": {},
            "cert": "MII..."
          }
        ],
        "signing_cert": {
          "url": "rsync://localhost/repo/ta/0/0BA5C132B94891CB2D3A89EDE12F01ACA4BCD3DC.
↪cer",
          "cert": "MII..."
        }
      }
    ]
  }
}
```

Example API:

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

8.14 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":
↪"MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyhFOTBDMjE3MzRDMkMzNzBBOTFBODQ3NUNCNEYwRTc1
↪vVYxq1F1w2yQ/
↪VoTr1dvEHxJ+SDayMcFVktWCObiY8tcPhvWG+OdaX9ckDJhsOEEvdVEogwiGacNs7yXJPbqDBptJtbR8/
↪CauF9OqMqjkB/8xkGmBoY5OI/
↪V2832jkp7LPsbyET0RMQN7fgSpGbewvkaZVxGU3pHh5kT1nzPTXrwjxNMXgpunSEY7zR20vYCvsYYbxnSwFNbSMSL+Jgpa+HWPUc0
↪qlqlZK+tkppI2LkSBhTV5+n7cGA8ZsCAwEAAaNTMFEwDwYDVR0TAQH/BAUwAwEB/
↪zAdBgNVHQ4EFgQU6Qwhc0wsNwqRqEdctPDnXaTQ8L8wHwYDVR0jBBgwFoAU6Qwhc0wsNwqRqEdctPDnXaTQ8L8wDQYJKoZIhvcNAQ
↪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:
  https://localhost:3000/api/v1/cas/newca/parents/my_parent
Headers:
  Authorization: Bearer secret
```

8.15 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
```

8.16 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
```

8.17 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>
↪MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyhFRjJENzgwRkNCRkU1QjZBMkExMjA1OUM0MDlDN0M5M
↪X3Ho5gC7uuKf4LYqbJj+Z9ltr/236/
↪hDYJfWMXZVcEuL+wUble1zhe2NKrgnAkpReVMSdiugoqZ9ICK2Fwkj5jCGc/qHiWOba7T78zfij8OlB/
↪dGlJvkAY8b/XTNKsTrLozi1uVAC8GqDrV5MEgY/NfzUvgA024yxx/rC6QBDEoBjnP7wDFiaZ2lwvL2beVYu6/
↪hVcXQzsVN+ijy7cGdkE6zi0meXJLTHPEpoA88hi3Pi+pIDBIQ3wTcpQIOqAq/
↪SZuh4dbZK7BV8MCAwEAAaNTMFEwDwYDVR0TAQH/BAUwAwEB/
↪zAdBgNVHQ4EFgQU7y14D8v+W2oqEgWcQJx8kndUGVYwHwYDVR0jBBgwFoAU7y14D8v+W2oqEgWcQJx8kndUGVYwDQYJKoZIhvcNAQ
↪gpJtONdgIWV1EqwEzhKKA2EP6tLDF9ejsdMFNYrYr+2hVWaoLsSuarfwfLFSgKDFqR6sh3ljYq6mIz9gdkjBOJsR9JyHFEtsDsRpf
↪YPMPVBpmG15Z9iKantzC1tck+E1xYW5awvj+YZqGVqyFdPJOZWmaYoS83kWvg4g4IucXTH6wwy23MQ7+0gyoK4wxfXRQmWjlXpLue
↪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
```

8.18 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":
↪"MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyg4OEJBMzA2QkMzMUVFRkU3NzRDNzYzRUY1N0VBNUZE
↪c7x5sy3JbmUWaQHtkl6N9l8vcRlQQfhk0vwlVCHcQQrcMViJ5GmGtEjo7+Uf9e0TDA+rrkdqOkpOLcGRKjs1SZNqCRktubQU7Ndc0
↪gYAnYssX26kObXan0fD9rgv4aWK0Xzp5hwz1ECAwEAAaNTMFEwDwYDVR0TAQH/BAUwAwEB/
↪zAdBgNVHQ4EFgQUiLowa8Me7+d0x2PvV+pf3HoZ3iUwHwYDVR0jBBgwFoAUiLowa8Me7+d0x2PvV+pf3HoZ3iUwDQYJKoZIhvcNAQ
↪Yz1VkQUTjLn2x7DKwuL9A8+IrYELSth4aCNSgPkhZfDL238MflAxptNRAoIeRGn8l3oSg4AUzBuSCErwvBbHWSh066nV0wzVFb+mL
↪GMiZHI/MwGZpj86Q/8wvyyw2C0b0ddWaoXwDyJjuxja0nHPDHVriJ8/
↪xsOfBk144n1zyP++apQXmXorCy4hs9GPyr+HGeoL6kNydDxdwzJLCqWW7u3wSnxjCJk+hfGq82qNm90ALv5PaQb58fDgWwBwuvTP0
↪",
```

(continues on next page)

(continued from previous page)

```
    "service_uri": "https://localhost:3000/rfc8181/localname/",
    "repo_info": {
      "sia_base": "rsync://localhost/repo/localname/",
      "rrdp_notification_uri": "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.

8.19 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":
↪"MIIJTQYJKoZIhvcNAQcCoIIJPjCCCToCAQMxDTALBglghkgBZQMEAgEwgZsGCyqGSIb3DQEJEAEaoIGLBIGIMIGFAgEBGA8yMDIx
↪NWi2U6R7ffaCCBtgwggbUMIIFvKADAgECAhQfMMPbsoNAMCZM8zHTPf4QMZ06vjANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEygx
↪q+Cnm2o/yJFjBfZSYNpE2xo4+nwz4HT6ShXoeSvcQAa6xjeP5z9QwbShfsoGWG0PTert08L5nMpnN4WB/
↪uyfT2l6cHua+M36HNpd23Bx57aaOwRhugMph47pkADkiq5HICZIt/a1/
↪UeFqtsx4D5LMXE6P4XnSPG69K2mQhroGLIIrr6x0LAXlE0uNAkYEcHnmdJwWssG0RoMRGf4myWPDk9pSrkLAgMBAAGjggHiMIIB3j
↪6TJwtHRc735eady5RAOIwHwYDVR0jBBgwFoAUFrMckusRa8YAJsUJRK1EIF3ZrL0wDgYDVR0PAQH/
↪BAQDAgeAMGAGA1UdHwRZMFcwVaBToFGGT3JzeW5jOi8vbG9jYWxob3N0L3JlcG8vbG9jYWxuYW1lLzAvMTZCMzFDOTJFQjExNkJDN
↪BA4wDDAKBggrBgEFBQcOAjAhBggrBgEFBQcBBwEB/
↪wQSMBAwBgQCAAEFADAGBAIAAgUAMBUGCCsGAQUFBwEIAQH/
↪BAYwBKACBQAwDQYJKoZIhvcNAQELBQADggEBAJQiHZ91d7/
↪a52qM0DyXp7jbkygm2MkT5tc6pp6sxHv6pDfXxAzJS8OtgcFCTDKC57pKvVvw8THE079nbMSxfaA5nP8egedxeuTzrj8iOh9nHk/
↪X4pWhIWsAvNgiTebYj+Eax97MmRWAkDgxWpDQ+CWQBl2gBstLmBKCBTw6cFlkGrBCLVe+gSDTnHpy4ltza6pD+EawTNrGLBnFn+/
↪+dgzx/GA2qbRXiBLm2/R4HR7zI/QYy+wWDoaZraCu6dUZEF4WomS99aihEyNp8tzyuEntmmMfw0z/
```

(continues on next page)

(continued from previous page)

```
      "uri": "rsync://localhost/repo/localname/0/
↪16B31C92EB116BC60026C50944AD44205DD9ACBD.mft"
    },
    {
      "base64":
↪"MIIBrzCBmAIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEygxNkIzMUM5MkVCMTE2QkM2MDAyNkM1MDk0NEFENDQyMDVERDlB
↪0Qx341/
↪wWYrBrEAQ56NE6AVN+r0qjmO2mhNgVNQ1VdCLjo67ilQufmxGhtUQxBS625f1hr69cYw1l15wHDP4SFpXO96ysTxBhxpLGL215nT0
↪",
      "uri": "rsync://localhost/repo/localname/0/
↪16B31C92EB116BC60026C50944AD44205DD9ACBD.crl"
    }
  ]
}
```

Example API:

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

8.20 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
```

8.21 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
7y14D8v+W2oqEgWcQJx8kndUGVYwDQYJKoZIhvcNAQELBQADggEBAArqsa/gpJtO
NdgIWV1EqwEzhKKA2EP6tLDF9ejsdMFNYrYr+2hVWaoLsSuarfwfLFSgKDFqR6sh
3ljYq6mIz9gdkjBOJsR9JyHFEtsDsRpf8Hs1WlbIb8bWb73Cp/YPMPVBpmG15Z9i
KantzC1tck+E1xYW5awvj+YZqGVqyFdPJOZWmaYoS83kWvg4g4IucXTH6wwy23MQ
7+0gyoK4wxfXRQmWjlXpLueCOsJo7ZXopsDAmXHLoFKZVEXn1ocQNc91l521BEQ6
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:
```

(continues on next page)

(continued from previous page)

```
<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+0gyoK4wxfXRQmWjlXpLueCOsJo7ZXopsDAmXHLoFKZVEXn1ocQNc91l521BEQ6\nt/
↪d7srQA4IxZCRGh9B+JdAIOKuXBA0nncmMJLQN8Qpxlz2bxKKAgXBLdoDqjbTDV\nbXTPM8YLRgc=\n-----END␣
↪CERTIFICATE-----\n",
    "hash": "992ac17d85fef11d8be4aa37806586ce68b61fe9cf65c0965928dbce0c398a99"
  },
  "repo_info": {
    "sia_base": "rsync://localhost/repo/localname/",
    "rrdp_notification_uri": "https://localhost:3000/rrdp/notification.xml"
  },
  "parents": [
    {
      "handle": "my_parent",
      "kind": "rfc6492"
    }
  ],
  "resources": {
    "asn": "AS65000",
    "ipv4": "10.0.0.0/8",
    "ipv6": "2001:db8::/32"
  },
  "resource_classes": {
    "0": {
      "name_space": "0",
      "parent_handle": "my_parent",
      "keys": {
        "active": {
          "active_key": {
            "key_id": "16B31C92EB116BC60026C50944AD44205DD9ACBD",
            "incoming_cert": {
```

(continues on next page)

(continued from previous page)

```
              "cert":
↪"MIIFYDCCBEigAwIBAgIUN5PzATTKVrjgual4CpJMaggW2EIwDQYJKoZIhvcNAQELBQAwMzExMC8GA1UEAxMoMEJBNUMxMzJCOTQ4
↪7OHN8TU6crIu1/
↪w1gkf6UCXFrv+poW9EJHnLonMa4ZFLSFsvQACIGUpxIuiQjaSYFltTbb+o2c9KWoKsX0kZqt5zOrgAP8cke8SFGHdqqenPInXKTgy
↪ngql4OV0bLkbb63J/26c8FZOThZAgMBAAGjggJqMIICZjAPBgNVHRMBAf8EBTADAQH/
↪MB0GA1UdDgQWBBQWsxyS6xFrxgAmxQlErUQgXdmsvTAfBgNVHSMEGDAWgBQLpcEyuUiRyy06ie3hLwGspLzT3DAOBgNVHQ8BAf8EB
↪BA4wDDAKBggrBgEFBQcOAjAsBggrBgEFBQcBBwEB/
↪wQdMBswCgQCAAEwBAMCAAowDQQCAAIwBwMFACABDbgwGgYIKwYBBQUHAQgBAf8ECzAJoAcwBQIDAP3oMA0GCSqGSIb3DQEBCwUAA4
↪Yja+sYhyg/pG1/
↪ZTvhOLIxGWap8JmqOnYa9XgX8uUlsV8LgJoEH3Gde3txcGtfLO99ugvbnKKGOcPxB8AX5hAhhfdiSnt3V06dEz3HUoTYdUKTV0bZr
↪K8349vN0QJUZGm3gAUsM5PlnAqbkM7VFIyu8g2Yp9g+M/
↪iwaHar8CqABKxLBThYgqrPLLv6CsZD3mjk5BkXVZh6R9dBcR7sPbSfGBWPWCv8SwLknyQDOvsWTho1Ga6AibjUQp
↪",
              "uri": "rsync://localhost/repo/testbed/0/
↪16B31C92EB116BC60026C50944AD44205DD9ACBD.cer",
              "resources": {
                "asn": "AS65000",
                "ipv4": "10.0.0.0/8",
                "ipv6": "2001:db8::/32"
              }
            },
            "request": null
          }
        }
      }
    }
  },
  "children": [],
  "suspended_children": []
}
```

Example API call:

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

8.22 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
```

8.23 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
```

8.24 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",
        "events": [
          1
        ]
      }
    },
    {
      "key": "command--1617875603--2--cmd-ca-repo-update",
      "actor": "master-token",
      "timestamp": 1617875603613,
      "handle": "newca",
      "version": 2,
```

(continues on next page)

(continued from previous page)

```
      "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": {
        "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",
```

(continues on next page)

(continued from previous page)

```
          "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
```

8.25 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
Changes:
  added resource class with name '0'
  requested certificate for key (hash) '16B31C92EB116BC60026C50944AD44205DD9ACBD' under␣
↪resource class '0'
```

(continues on next page)

(continued from previous page)

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",
            "ipv4": "10.0.0.0/8",
            "ipv6": "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": {
```

(continues on next page)

(continued from previous page)

```
      "class_name": "0",
      "limit": {
       "asn": "none",
       "ipv4": "none",
       "ipv6": "none"
      },
      "csr":
↪"MIIDjzCCAncCAQAwMzExMC8GA1UEAxMoMTZCMzFDOTJFQjExNkJDNjAwMjZDNTA5NDRBRDQ0MjA1REQ5QUNCRDCCASIwDQYJKoZI
↪6EvY4Uo0ICel3vWE5cwE1db/s4c3xNTpysi7X/DWCR/pQJcWu/
↪6mhb0QkecuicxrhkUtIWy9AAIgZSnEi6JCNpJgWW1Ntv6jZz0pagqxfSRmq3nM6uAA/
↪xyR7xIUYd2qp6c8idcpODKyz2QKz2kW0yToEhroqNG+oVVkh/
↪rEbnfK0ncmqwaO8SjyqgdjGS+Qy1uOGOWZbT3uLoN4LXbjVfdIbrgNIyTUI2/
↪XG3kxJe5svNY4P7aUsh3+eCqXg5XRsuRtvrcn/
↪bpzwVk5OFkCAwEAAaCCARUwggERBgkqhkiG9w0BCQ4xggECMIH/MA8GA1UdEwEB/wQFMAMBAf8wDgYDVR0PAQH/
↪BAQDAgEGMIHbBggrBgEFBQcBCwEBAASByzCByDAvBggrBgEFBQcwBYYjcnN5bmM6Ly9sb2NhbGhvc3QvcmVwby9sb2NhbG5hbWUvM
↪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
```

8.26 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
```

8.27 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
```

8.28 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
```

(continues on next page)

(continued from previous page)

```
Body:
{
  "added": [
    {
      "asn": 64496,
      "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
```

(continues on next page)

(continued from previous page)

```
Headers:
 content-type: application/json
 Authorization: Bearer secret
Body:
{
 "added": [
   {
     "asn": 64496,
     "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": [
      {
```

(continues on next page)

(continued from previous page)

```
        "asn": 1,
        "prefix": "10.0.0.0/20",
        "max_length": 24
      }
    ],
    "notheld": [
      {
        "asn": 1,
        "prefix": "10.128.0.0/9"
      }
    ],
    "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:
```

(continues on next page)

(continued from previous page)

```
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": [
        {
          "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
        }
      ]
```

(continues on next page)

(continued from previous page)

```
  },
  {
   "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"
   },
   {
    "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
```

(continues on next page)

(continued from previous page)

```

Announcements which are valid:

        Announcement: 10.0.0.0/24 => 64496
```

8.29 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):
```

(continues on next page)

(continued from previous page)

```
                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
  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
```

8.30 krillc bgpsec

Manage BGPSec Router Certificates for your CA.

Krill lets users create **RFC 8209** BGPSec Router Certificates. These certificates are used in BGPSec to authorise a router key for an ASN in the RPKI.

At the moment BGPSec deployment is virtually non-existent, so you are unlikely to need this. However, this functionality is provided in the hope that it will help the community gain operational experience that may help BGPSec deployment.

Currently BGPSec Router Certificates can only be managed through the API. If there is popular demand we will add this to the UI in future.

```
USAGE:
    krillc bgpsec [SUBCOMMAND]

SUBCOMMANDS:
    list      Show current BGPSec configurations
    add      Add BGPSec configurations
    remove      Remove a BGPSec definition
```

8.31 krillc bgpsec list

Show the current BGPSec configurations.

Example CLI:

```
$ krillc bgpsec list
ASN, key identifier, CSR base64
AS211321, 17316903F0671229E8808BA8E8AB0105FA915A07, MIH.....
```

Example JSON response:

8.32 krillc bgpsec add

Add a new BGPSec configurations. I.e. choose an ASN you hold and a Certificate Sign Request (CSR) you got from your router so that Krill can create a BGPSec Router Certificate for it.

Example CLI:

```
$ krillc bgpsec add --asn AS65000 --csr ./router-csr.der
```

This will submit the following JSON to the API:

```
$ krillc bgpsec add --asn AS65000 --csr ./router-csr.der --api
POST:
  https://localhost:3000/api/v1/cas/local-testbed-child/bgpsec
Headers:
  content-type: application/json
  Authorization: Bearer secret
Body:
{
  "add": [
    {
      "asn": 65000,
      "csr":
↪"MIH7MIGiAgEAMBoxGDAWBgNVBAMMD1JPVVRFUi0wMDAwM0NDQTBZMBMGByqGSM49AgEGCCqGSM49AwEHA0IABE9dBTAcT+j96+mh
↪DBs2N3Fzb2euM+AqWdtoH+LXsmxqvKOgJjAkBgkqhkiG9w0BCQ4xFzAVMBMGA1UdJQQMMAoGCCsGAQUFBwMeMAoGCCqGSM49BAMCA
↪5SMaUJ2hp3r8joKFq3hA="
    }
  ],
  "remove": []
```

}

8.33 krillc bgpsec remove

Note that Krill may actually create multiple BGPSec Router Certificates based on the CSR *if* you hold the ASN multiple times. E.g. under mutliple parents. In practice this is unlikely to happen, but this is conceptually important when it comes to removal. You can remove any and all BGPSec Router Certificate by asking Krill to remove the configuration for a given ASN and router key identifier (as shown in the list command).

Example CLI:

```
$ krillc bgpsec remove --asn AS65000 --key 17316903F0671229E8808BA8E8AB0105FA915A07
```

This submits the following JSON to the API:

```
$ krillc bgpsec remove --asn AS65000 --key 17316903F0671229E8808BA8E8AB0105FA915A07 --api
POST:
  https://localhost:3000/api/v1/cas/local-testbed-child/bgpsec
Headers:
  content-type: application/json
  Authorization: Bearer secret
Body:
{
  "add": [],
  "remove": [
    "ROUTER-00033979-17316903F0671229E8808BA8E8AB0105FA915A07"
  ]
}
```

Careful observers may have noticed that the API supports mutliple additions and removals in a single update. However, such bulk changes are not yet supported in the CLI.

8.34 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
```

8.35 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>
```

8.36 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>
```

8.37 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:
```

(continues on next page)

(continued from previous page)

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

8.38 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        Show the RFC8183 Parent Response XML
    connections     Show connections stats for children of a CA
    suspend         Suspend a child CA: hide certificate(s) issued to child
    unsuspend       Suspend a child CA: republish certificate(s) issued to child
    remove          Remove an existing child from a CA
```

8.39 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">
```

(continues on next page)

(continued from previous page)

```
  <parent_bpki_ta>
↪MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyhFOTBDMjE3MzRDMkMzNzBBOTFBODQ3NUNCNEYwRTc1R
↪vVYxq1F1w2yQ/
↪VoTr1dvEHxJ+SDayMcFVktWCObiY8tcPhvWG+OdaX9ckDJhsOEEvdVEogwiGacNs7yXJPbqDBptJtbR8/
↪CauF9OqMqjkB/8xkGmBoY5OI/
↪V2832jkp7LPsbyET0RMQN7fgSpGbewvkaZVxGU3pHh5kT1nzPTXrwjxNMXgpunSEY7zR20vYCvsYYbxnSwFNbSMSL+Jgpa+HWPUc0
↪qlqlZK+tkppI2LkSBhTV5+n7cGA8ZsCAwEAAaNTMFEwDwYDVR0TAQH/BAUwAwEB/
↪zAdBgNVHQ4EFgQU6Qwhc0wsNwqRqEdctPDnXaTQ8L8wHwYDVR0jBBgwFoAU6Qwhc0wsNwqRqEdctPDnXaTQ8L8wDQYJKoZIhvcNAQ
↪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": {
    "asn": "AS65000",
    "ipv4": "10.0.0.0/8",
    "ipv6": "2001:db8::/32"
  },
  "id_cert":
↪"MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyhFRjJENzgwRkNCRkU1QjZBMkExMjA1OUM0MDlDN0M5
↪X3Ho5gC7uuKf4LYqbJj+Z9ltr/236/
↪hDYJfWMXZVcEuL+wUble1zhe2NKrgnAkpReVMSdiugoqZ9ICK2Fwkj5jCGc/qHiWOba7T78zfij8OlB/
↪dGlJvkAY8b/XTNKsTrLozi1uVAC8GqDrV5MEgY/NfzUvgA024yxx/rC6QBDEoBjnP7wDFiaZ2lwvL2beVYu6/
↪hVcXQzsVN+ijy7cGdkE6zi0meXJLTHPEpoA88hi3Pi+pIDBIQ3wTcpQIOqAq/
↪SZuh4dbZK7BV8MCAwEAAaNTMFEwDwYDVR0TAQH/BAUwAwEB/
↪zAdBgNVHQ4EFgQU7y14D8v+W2oqEgWcQJx8kndUGVYwHwYDVR0jBBgwFoAU7y14D8v+W2oqEgWcQJx8kndUGVYwDQYJKoZIhvcNAQ
↪gpJtONdgIWV1EqwEzhKKA2EP6tLDF9ejsdMFNYrYr+2hVWaoLsSuarfwfLFSgKDFqR6sh3ljYq6mIz9gdkjBOJsR9JyHFEtsDsRpf
↪YPMPVBpmG15Z9iKantzC1tck+E1xYW5awvj+YZqGVqyFdPJOZWmaYoS83kWvg4g4IucXTH6wwy23MQ7+0gyoK4wxfXRQmWjlXpLue
↪d7srQA4IxZCRGh9B+JdAIOKuXBA0nncmMJLQN8Qpxlz2bxKKAgXBLdoDqjbTDVbXTPM8YLRgc="
}
```

8.40 krillc children info

Show info for a child: state, id certificate info and resources. The “state” can either be “active”, or “suspended”.

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+0gyoK4wxfXRQmWjlXpLueCOsJo7ZXopsDAmXHLoFKZVEXn1ocQNc91l521BEQ6
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:
state: active
```

Example JSON response:

```
{
  "state": "active",
  "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+0gyoK4wxfXRQmWjlXpLueCOsJo7ZXopsDAmXHLoFKZVEXn1ocQNc91l521BEQ6\nt/
↪d7srQA4IxZCRGh9B+JdAIOKuXBA0nncmMJLQN8Qpxlz2bxKKAgXBLdoDqjbTDV\nbXTPM8YLRgc=\n-----END␣
↪CERTIFICATE-----\n",
```

(continues on next page)

(continued from previous page)

```
    "hash": "992ac17d85fef11d8be4aa37806586ce68b61fe9cf65c0965928dbce0c398a99"
  },
  "entitled_resources": {
    "asn": "",
    "ipv4": "10.0.0.0/8, 192.168.0.0/16",
    "ipv6": ""
  }
}
```

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
```

8.41 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:
{
  "id_cert": null,
  "resources": {
    "asn": "",
    "ipv4": "10.0.0.0/8",
    "ipv6": ""
  }
}
```

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:
```

(continues on next page)

(continued from previous page)

```
  content-type: application/json
  Authorization: Bearer secret
Body:
{
  "id_cert":
↪"MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyhFRjJENzgwRkNCRkU1QjZBMkExMjA1OUM0MDlDN0M5
↪X3Ho5gC7uuKf4LYqbJj+Z9ltr/236/
↪hDYJfWMXZVcEuL+wUble1zhe2NKrgnAkpReVMSdiugoqZ9ICK2Fwkj5jCGc/qHiWOba7T78zfij8OlB/
↪dGlJvkAY8b/XTNKsTrLozi1uVAC8GqDrV5MEgY/NfzUvgA024yxx/rC6QBDEoBjnP7wDFiaZ2lwvL2beVYu6/
↪hVcXQzsVN+ijy7cGdkE6zi0meXJLTHPEpoA88hi3Pi+pIDBIQ3wTcpQIOqAq/
↪SZuh4dbZK7BV8MCAwEAAaNTMFEwDwYDVR0TAQH/BAUwAwEB/
↪zAdBgNVHQ4EFgQU7y14D8v+W2oqEgWcQJx8kndUGVYwHwYDVR0jBBgwFoAU7y14D8v+W2oqEgWcQJx8kndUGVYwDQYJKoZIhvcNAQ
↪gpJtONdgIWV1EqwEzhKKA2EP6tLDF9ejsdMFNYrYr+2hVWaoLsSuarfwfLFSgKDFqR6sh3ljYq6mIz9gdkjBOJsR9JyHFEtsDsRpf
↪YPMPVBpmG15Z9iKantzC1tck+E1xYW5awvj+YZqGVqyFdPJOZWmaYoS83kWvg4g4IucXTH6wwy23MQ7+0gyoK4wxfXRQmWjlXpLue
↪d7srQA4IxZCRGh9B+JdAIOKuXBA0nncmMJLQN8Qpxlz2bxKKAgXBLdoDqjbTDVbXTPM8YLRgc=",
  "resources": null
}
```

8.42 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>
↪MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyhFOTBDMjE3MzRDMkMzNzBBOTFBODQ3NUNCNEYwRTc1R
↪vVYxq1F1w2yQ/
↪VoTr1dvEHxJ+SDayMcFVktWCObiY8tcPhvWG+OdaX9ckDJhsOEEvdVEogwiGacNs7yXJPbqDBptJtbR8/
↪CauF9OqMqjkB/8xkGmBoY5OI/
↪V2832jkp7LPsbyET0RMQN7fgSpGbewvkaZVxGU3pHh5kT1nzPTXrwjxNMXgpunSEY7zR20vYCvsYYbxnSwFNbSMSL+Jgpa+HWPUc0
↪qlqlZK+tkppI2LkSBhTV5+n7cGA8ZsCAwEAAaNTMFEwDwYDVR0TAQH/BAUwAwEB/
↪zAdBgNVHQ4EFgQU6Qwhc0wsNwqRqEdctPDnXaTQ8L8wHwYDVR0jBBgwFoAU6Qwhc0wsNwqRqEdctPDnXaTQ8L8wDQYJKoZIhvcNAQ
↪nVmqDlK/1sWZNUXFWP4dt1wLTjDWnceyS8mI7Yx8dH/Fez60m4lp4dD45eeaXfbjP2cWnh3n/
↪PLGE70Nj+G0AnUhUmwiTl0H6Px1xn8fZouhv9MEheaZJA+M4NF77+Nmkp2P3WI4cvIS7Te7R/
↪7XpwSr29lVNtYjmRlrBDXx/bMFSgFL61mrtj/
↪l6G8OB40w+sAwO0XKUj1vUUpfIXc3ISCo0LNT9JSPcgy1SZWfmLb98q4HuvxekhkIPRzW7vlb/
↪NBXGarZmKc+HQjE2aXcIewhen2OoTSNda2jSSuEWZuWzZu0aMCKwFBNHLqs=</parent_bpki_ta>
</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.

8.43 krillc children connections

Show the connections stats for children of a CA. This can be useful for monitoring for potentially deactivated child CAs. Furthermore the user-agent for the last known connection from each child is shown. This can help to monitor for children running potentially outdated RPKI CA implementations (old krill versions or other implementations).

Example CLI:

```
$ krillc children connections --ca testbed
handle,user_agent,last_exchange,result,state
CA2,krill/0.9.2-rc3,2021-09-24T10:00:00+00:00,success,active
ca,krill/0.9.2-rc3,2021-09-24T10:00:00+00:00,success,active
CA1,krill/0.9.2-rc1,2021-09-13T14:30:00+00:00,success,active
dummy_ca,n/a,never,n/a,active
```

Example API call:

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

Example JSON response:

```
{
  "children": [
    {
      "handle": "newca",
      "last_exchange": {
        "timestamp": 1632477600,
        "result": "Success",
        "user_agent": "krill/0.9.2"
      },
      "state": "active"
    },
    {
      "handle": "oldca",
      "last_exchange": {
        "timestamp": 1632477600,
        "result": "Success",
        "user_agent": "krill"
      },
      "state": "active"
    },
    {
      "handle": "brandnewca",
      "last_exchange": null,
      "state": "active"
```

(continues on next page)

(continued from previous page)

```
    },
  ]
}
```

Note that krill 0.9.1 and below use the user-agent “krill”, while krill 0.9.2 and above include the version, e.g.: “krill/0.9.2”. Other RPKI CA implementation may or may not include user-agents strings in their requests.

Furthermore note that the “last_exchange” may be “null” in case a CA was just added by the parent, but the child CA did not import the parent XML response yet - or was otherwise unable to connect.

The “last_exchange” field will also be “null” after upgrading to krill 0.9.2. This information was not kept prior to krill 0.9.2 so, after upgrading, this will only be set when your existing child CAs connect for the first time.

8.44 krillc children suspend

If you believe that a child CA has been deactivated then you may wish to “suspend” it, rather than remove it altogether. If you suspend a child CA, then any certificate(s) issued to it by your CA will be withdrawn, and they will no longer be processed by RPKI validation software. This is particularly useful if the manifest and CRL of the child CA have expired, and presumably their ROAs are no longer maintained either.

If and when a “suspended” child CA connects to your CA again, it will automatically be “un-suspended”. Meaning that any certificate(s) previously issued to this child will be published again.

The main goal of this is to facilitate an easier recovery path in cases where a child CA suffers a long outage. By “suspending” them until the child CA is reactivated you suppress RPKI validation errors for their expired publication point, while ensuring that the delegation to this CA will be re-enabled as soon as it is successfully started.

Example CLI/API:

```
$ krillc children suspend --ca testbed --child newca --api

POST:
  https://localhost:3000/api/v1/cas/testbed/children/newca
Headers:
  content-type: application/json
  Authorization: Bearer secret
Body:
{
  "suspend": true
}
```

Important: It is not always trivial to figure out if a child CA has been deactivated. The expiry of the child CA’s manifest and CRL is a strong indication of this, but this information is not available to the krill CA parent. What it *does* have is the knowledge of when a child CA connected for the last time.

If the child CA did not connect for a long time, then the parent may be inclined to think that they have been deactived. This is true for child CAs running Krill 0.9.2 or above, because here the maximum configurable ‘refresh’ rate is one hour. So, if you have not seen any connection attempts for such child CAs for, say 8 hours, then you can safely suspend them.

However, earlier krill versions, while using a default of 10 minutes, would allow overriding this value without any upper bound. Other RPKI CA implementations may also use longer cycles.

In short: be careful before deciding that a child CA is truly deactived.

8.45 krillc children unsuspend

If needed you can manually "un-suspend" a "suspended" child CA. Generally speaking there is no need do this, because a child will be un-suspended automatically whenever it re-connects with your CA.

Example CLI/API:

```
$ krillc children unsuspend --ca testbed --child newca --api
POST:
  https://localhost:3000/api/v1/cas/testbed/children/newca
Headers:
  content-type: application/json
  Authorization: Bearer secret
Body:
{
  "suspend": false
}
```

8.46 krillc children remove

Remove an existing child from a CA. This removes and revokes any certificate(s) issued to this child CA. Furthermore this child CA, if still active or re-activated, will no longer be allowed to connect to your CA. They will have to remove you as a parent first and then re-do the XML exchange with you in order to be re-added as a child.

if you think that the child CA may be temporarily disabled, then you may wish to "suspend" them instead.

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
```

8.47 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.

8.48 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>
```

8.48.1 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:

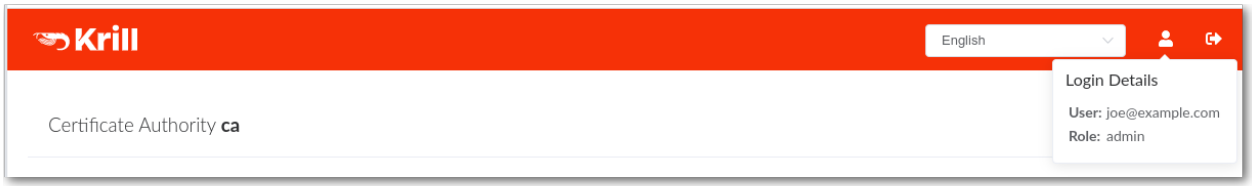
```
$ krillc keyroll activate --ca newca --api
POST:
  https://localhost:3000/api/v1/cas/newca/keys/roll_activate
Headers:
  Authorization: Bearer secret
Body:
<empty>
```

CHAPTER

NINE

LOGIN WITH NAMED USERS

New in version v0.9.0.



User: joe@example.com

Role: admin

Certificate Authority **ca**

Fig. 1: Checking the currently logged in user and user attributes

By default Krill requires users to authenticate using the configured secret token, and actions in the event history are attributed to a client using the secret token or to Krill itself.

Krill also supports authenticating users **of the web user interface** with their own username and credentials. Actions taken by such logged in users are attributed in the event history to their username.

To login users by username Krill must first be configured either with locally defined user details and credentials, or with the details necessary to interact with a separate OpenID Connect compliant identity provider system.

Further reading:

9.1 Permissions, Roles & Attributes

New in version v0.9.0.

This page summarizes the different ways that Krill supports for restricting access to *named users* that login to Krill. For backward compatibility, users that authenticate with the secret token are given unrestricted access to Krill.

9.1.1 Permissions

Internally within Krill each REST API endpoint requires the logged in user to have a specific Krill permission in order to execute the request.

9.1.2 User Attributes

User attributes are assigned by the identity provider, either in the `krill.conf` file for locally defined users, or in the management interface of the OpenID Connect provider that manages your users.

Warning: By default, user attributes and their values are shown in the Krill web user interface and the web user interface stores these attributes in browser local storage. To prevent sensitive attributes being revealed in the browser you can mark them as private. One possible use for this is to restrict access using the `exc_cas` attribute but not reveal the name of the restricted CA by doing so. See `auth_private_attributes` in `krill.conf` file for more information.

9.1.3 Role Based Access Control

At the highest level Krill can restrict access based on user roles. A role is a named collection of internal Krill permissions.

By default Krill supports three roles which can be assigned to users. A user can only have one role at a time. A role is assigned to a user via the `role` user attribute (see below for more on attributes).

The default roles are:

- `admin` : Grants users unrestricted access.
- `readwrite`: Grants users the right to list, view and modify *existing* CAs.
- `readonly` : Grants users the right to list and view CAs only.

9.1.4 Attribute Based Access Control

Krill supports `inc_cas` and `exc_cas` user attributes which can be used to permit or deny access to one or more Certificate Authorities in Krill. User attributes can also be used to make decisions in *custom authorization policies*.

9.2 Config File Users

New in version v0.9.0.

9.2.1 Introduction

By setting `auth_type = "config-file"` in `krill.conf` you can configure Krill to require users to enter a username and password in the web user interface when logging in, rather than the secret token that is usually required:

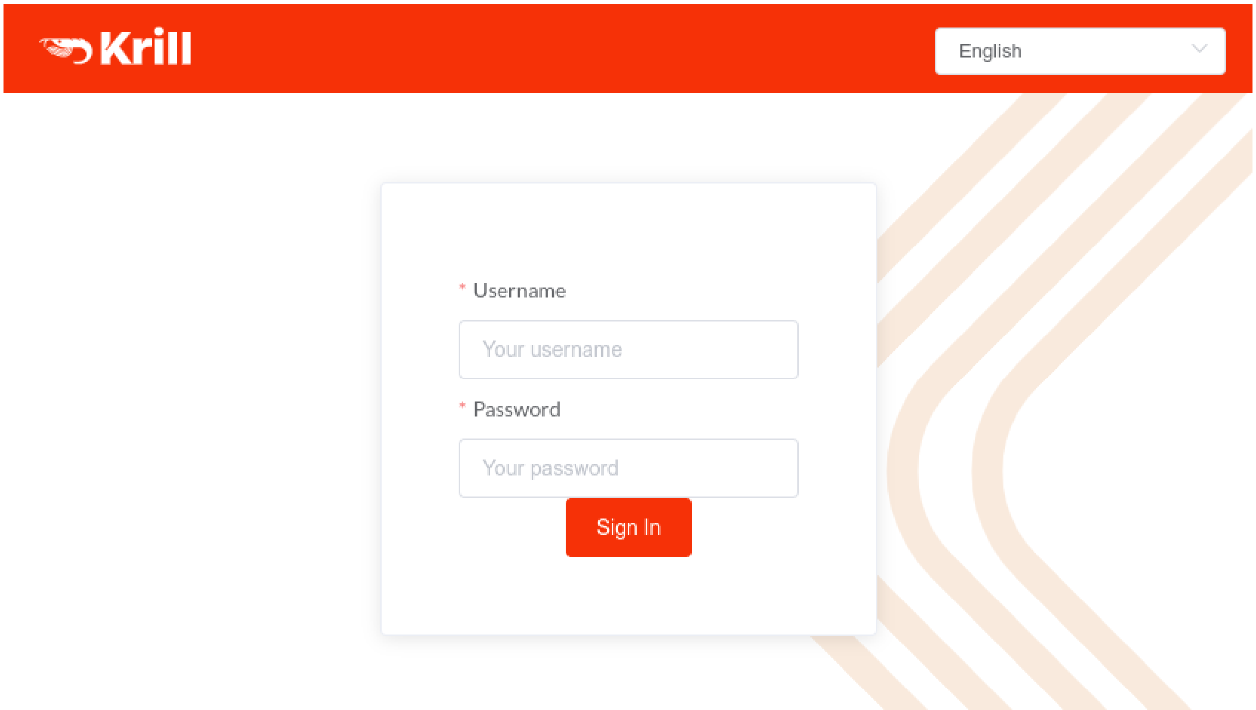


Fig. 2: Using config file user credentials to login to Krill

Note: It is important to realize that Krill is not a complete user management system and that Config File Users therefore have some *limitations*.

While Config File Users are useful as a quick way to test named user support in Krill and may suffice for simple situations, in larger more critical settings you are strongly advised to consider using *OpenID Connect Users* instead.

9.2.2 How does it work?

To add a user to the `krill.conf` file an administrator uses the `krillc` command to compute a password *hash* for the user and then adds an entry to the `[auth_users]` section including their username, password *hash*, salt and any *attributes* that are relevant for that user.

When a user enters their username and password into the web user interface a hash of the password is computed and sent with the username to the Krill server.

The Krill server will verify that the user logging in provided a correct password and has the `LOGIN` permission. On success Krill will respond with a token which the web user interface should send on subsequent requests to authenticate itself with Krill. The web user interface will keep a copy of this token in browser local storage until the user logs out or is timed out due to inactivity.

Tip: The actual user password is **NEVER** stored on either the Krill server nor the client browser and is **NEVER** sent by the client browser to the Krill server. Only password *hashes* are stored and transmitted.

Warning: Do **NOT** serve the Krill web user interface over unencrypted HTTP. While the password is never transmitted, the authentication token that the user is subsequently issued is subject to interception by malicious parties if sent unencrypted from the Krill server to the web user interface. Note that this is equally true when using any credential to authenticate with Krill, whether secret token or password hash or when Krill is configured to interact with an OpenID Connect provider.

9.2.3 Known limitations

Config File Users are easy to define and give you complete control over who has access to your Krill instance and what level of access is granted. However, Krill is not a complete user management system and so there are some things to remember when using Config File Users:

- Krill has no feature for requiring a user to change their password on first login. As such, by issuing users with passwords you become responsible for delivering the new password to them securely.
- OpenID Connect providers often have support for one-time passwords (OTP) or other secondary lines of defence to protect an account than just a username and password. Krill does not have this capability.
- Krill has no feature for generating cryptographically strong passwords. You are responsible for choosing sufficiently strong passwords for your users.
- Usernames, password hashes and user attributes are sensitive information. By adding them to your `krill.conf` file you become responsible for protecting them.
- If you lose your `krill.conf` file you will also lose the password hashes and will have to reset your users passwords unless you have a (**secure**) copy elsewhere.
- If a user forgets their password you will need to issue them with a new one. Krill does not offer a forgotten password or password reset feature.
- Adding or changing users requires a restart of Krill. There is no support in Krill at present for reloading the user details while Krill is running. While Krill is restarting the web user interface will be unavailable for your users.

9.2.4 Setting it up

The following steps are required to use Config File Users in your Krill setup.

1. Decide on the settings to be configured.

Decide which usernames you are going to configure, and what *role* and password they should have. For this example let's assume we want to configure the following users:

Username	Password	Role
joe@example.com	dFdsapE5	admin
sally	wdGypnx5	readonly
dave_the_octopus	qnky8Zuj	readwrite

2. Configure Krill

For each user generate a password hash and salt using the following command:

```
$ krillc config user --id joe@example.com
Enter the password to hash: ********

[auth_users]
"joe@example.com" = { password_hash="521e....0529", salt="d539....115e" }
```

Then add the `auth_type`, `[auth_users]` and individual user lines to `krill.conf`. The end result should look something like this:

```
auth_type = "config-file"

[auth_users]
"joe@example.com"  = { attributes={ role="admin" },     password_hash="521e....0529",␣
↪salt="d539....115e" }
"sally"            = { attributes={ role="readonly" },  password_hash="...", salt="..." }
"dave_the_octopus" = { attributes={ role="readwrite" }, password_hash="...", salt="..." }
```

3. Go!

Restart Krill and deliver the chosen passwords to the respective users to whom they belong. The users should now be able to login to your Krill instance.

Warning: Take whatever steps you think are necessary to ensure that the passwords are delivered **securely** to your users.

9.2.5 Advanced configuration

The information above gives you the basic structure for the configuration file syntax needed to configure local users in Krill.

See *Permissions, Roles & Attributes* for information about other user attributes and configuration settings that you might want to use.

See *Custom Authorization Policies* for information about customizing the configuration even further.

Below is a slightly modified version of the example above that also uses the `inc_cas`, `exc_cas` and `auth_private_attributes` features and adds a user that has custom team attributes as well. Notice how the team user does **NOT** have a `role` attribute!

```
auth_type = "config-file"
auth_private_attributes = [ "exc_cas" ]

[auth_users]
"joe@example.com"   = { attributes={ role="admin" }, password_hash="f45d...b25f", salt=".
↪.." }
"sally"             = { attributes={ role="readonly", inc_cas="ca1,ca3" },  password_
↪hash="...", salt="..." }
```

(continues on next page)

(continued from previous page)

```
"dave_the_octopus"  = { attributes={ role="readwrite" }, exc_cas="some_private_ca" },␣
↪password_hash="...", salt="..." }
"rob_from_team_one" = { attributes={ team="t1", teamrole="readwrite" }, password_hash="..
↪.", salt="..." }
```

9.2.6 Additional sources of information

The `krill.conf` file is the definitive guide to the possible values that can be used in the Krill configuration file. If in doubt, consult the `krill.conf` file that came with your copy of Krill.

Login related events will be reported in the Krill logs:

- Login failures are reported at error level.
- Login successes are reported at info level.
- Additional diagnostics may be reported at debug or trace level.

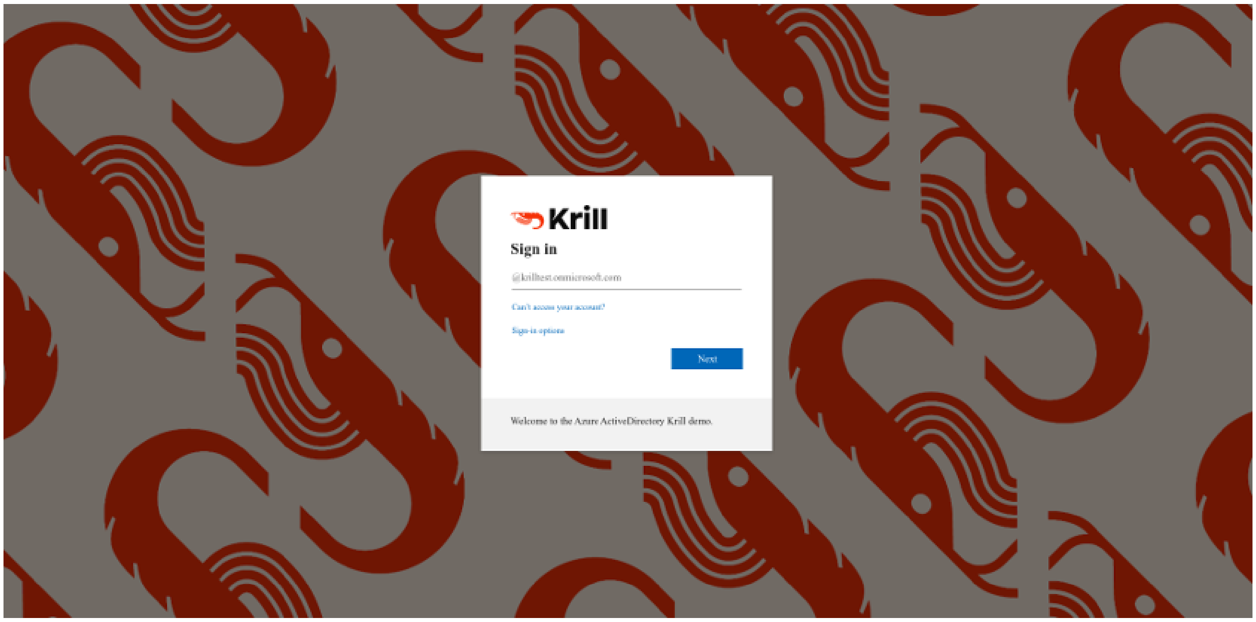
9.3 OpenID Connect Users

New in version v0.9.0.

9.3.1 Introduction

OpenID Connect is a widely supported standard that builds on the OAuth 2.0 standard to authenticate users and provide basic profile information about those users.

The user visible part of the login experience when using OpenID Connect is handled by the OpenID Connect provider and may look quite different to the Krill web user interface:



Welcome to the Azure ActiveDirectory Krill demo.

Fig. 3: Using Azure Active Directory as an OpenID Connect provider with Krill

To use OpenID Connect Users in Krill you will either need to run your own OpenID Connect provider or use one provided by a 3rd party service provider.

Why OpenID Connect?

From the OpenID Connect FAQ:

> **What problem does OpenID Connect solve?**
>
> *It lets app and site developers authenticate users without taking on the responsibility of storing and managing passwords in the face of an Internet that is well-populated with people trying to compromise your users' accounts for their own gain.*

OpenID Connect takes the lessons learned from earlier identity protocols and improves on them. It is widely implemented and deployed, and for situations where the primary identity provider does not implement OpenID Connect there are OpenID Connect providers that can act as a bridge to systems that implement other identity protocols.

As a modern, tried & tested and widely implemented protocol it is therefore quite likely that it is either already in use by (potential) Krill operators or viable for them to adopt.

Why not OAuth 2.0?

From https://oauth.net/articles/authentication/:

> **OAuth 2.0 is not an authentication protocol.**
>
> *Much of the confusion comes from the fact that OAuth is used inside of authentication protocols, and developers will see the OAuth components and interact with the OAuth flow and assume that by simply using OAuth, they can accomplish user authentication. This turns out to be not only untrue, but also dangerous for service providers, developers, and end users.*

9.3.2 How does it work?

Let's assume that the OpenID Connect provider is compatible with Krill and that Krill has been registered with the provider (see below for more on these topics).

The user experience

When an end user visits the Krill website in their browser they will be redirected to the login page of the OpenID Connect provider. This is **NOT** part of Krill.

For example, when logging in to a Krill instance connected to the OpenID Connect provider in a large company, the end user might see a very familiar login page. That's because it is probably a page they have to login to in order to use many other services in their company. Often this login page will even be themed to match the corporate branding.

The user enters **their** credentials into the OpenID Connect provider login page. At this point Krill knows nothing about who is logging in at the provider login form.

Tip: Krill **NEVER** receives the username or password that the user enters in to the OpenID Connect provider login page and Krill has no control over the appearance and/or behaviour of the OpenID Connect provider login page.

If the login is successful, from the users perspective their browser is then directed back to Krill where they see the Krill web user interface as if they are logged in. Krill will provide the web user interface with a token which the web user interface should send on subsequent requests to authenticate itself with Krill. The web user interface will keep a copy of this token in browser local storage until the user logs out or is timed out due to inactivity.

Krill will honour any session expiration time communicated to it by the OpenID Connect provider. When using OpenID Connect Users it is therefore possible that the user will be informed that they cannot perform the requested action because their login session has timed out and they need to login again. Where possible Krill will automatically extend the login session to avoid this happening.

In the background

What the user doesn't see, except perhaps if their network connection is very slow, is that there are "hidden" intermediate steps occuring in the login flow, between the browser and Krill and between Krill and the OpenID Connect provider. These steps implement the OpenID Connect "Authorizaton Code Flow".

If the user logged in correctly at the OpenID Connect provider login page and Krill was correctly registered with the provider and the provider was correctly setup for Krill, then Krill will receive a temporary Authorization Code which it exchanges for an OAuth 2.0 Access Token (and maybe also an OAuth 2.0 Refresh Token) and an OpenID Connect ID Token.

The ID Token includes so-called OAuth 2.0 **claims**, metadata about the user logging in. These claims are the key to whether or not Krill is able to determine which rights, if any, to grant to the user that is attempting to login.

9.3.3 Known limitations

OpenID Connect Users avoid the problems with *Config File Users* but require more effort to setup and maintain:

- Requires operating another service or using a 3rd party service.
- Confguring Krill and the OpenID Connect provider is more involved than setting up *Config File Users*.
- If Krill cannot contact the OpenID Connect provider, users will be unable to login to Krill with their OpenID Connect credentials. It will however still be possible to authenticate with Krill using its secret token.

> **Warning:** If you encounter HTTP 502 Bad Gateway errors from your HTTP proxy in front of Krill when logging in, or login loops where you are taken back to the OpenID Connect provider login page but the Krill logs show a successful login, you may need to increase the HTTP request and/or response header buffer sizes used by your proxy.
>
> With NGINX this can be done by increasing settings such as proxy_buffer_size, proxy_buffers, large_client_header_buffers (or http2_max_field_size and http2_max_header_size before NGINX v1.19.7). Thanks to GitHub user racompton for the `large_client_header_buffers` tip! If using Kubernetes use the equivalent NGINX ingress controller ConfigMap settings, e.g. http2-max-field-size. Thanks to GitHub user TheEnbyperor for the HTTP/2 and Kubernetes tips!
>
> These issues occur because the size of the HTTP request & response headers on login to Krill when using OpenID Connect can be quite large.

9.3.4 Choosing a provider

There are many identity providers that support OpenID Connect to choose from. Some are software products that you can host yourself, others are online services that you can create an account with.

Any OpenID Connect provider that you choose must implement the following standards:

- OpenID Connect Core 1.0
- OpenID Connect Discovery 1.0
- OpenID Connect RP-Initiated Logout 1.0 *(optional)*
- RFC 7009 OAuth 2.0 Token Revocation *(optional)*

Krill has been tested with the following OpenID Connect providers (in alphabetical order):

- Amazon Cognito
- Keycloak
- Microsoft Azure Active Directory
- Micro Focus NetIQ Access Manager 4.5

> **Warning:** Krill has been verified to be able to login and logout with Google Cloud accounts. However, it is not advisable to grant access to Google accounts in general. Instead you should use a Google product that permits you to manage your own pool of users so that you can restrict access to just these users. Additionally, if you wish to assign different Krill rights to different users you will need some way to mark the users to indicate which role they should receive, e.g. by grouping them or configuring custom claims.

Tip: If your provider does not offer sufficient control over the claim values it exposes to Krill, or if you do not have the ability to change these values to meet your needs, Krill supports a *hybrid mode* whereby authentication is handled by the OpenID Connect provider, but authorization can be based in whole or in part on config file defined user attributes.

9.3.5 Setting it up

Overview

The process for setting up Krill to support login by users of an OpenID Connect provider follows the same basic pattern for all providers but differs greatly in the details from one provider to the next.

In short, to setup any OpenID Connect provider with Krill the following steps must be taken:

1. **Decide on the settings to be configured**

   Ensure you have the basic pieces of information that you need. For example:

   - Which URL will Krill be available at?
   - Which user(s) will have admin rights in Krill?
   - Is there some property of these users that distinguishes them from other users (for example they may already be members of some internal Active Directory group) or will you need to mark them out in some way so that Krill can spot that they should be admins?
   - Is this property available by default as part of the standard claims sent by the provider to the client, or is it a provider specific claim or will it need to be configured in the provider as a custom claim?[1]
   - If no suitable claim values can be arranged with the provider, consider using *hybrid mode* instead.

2. **Gain access to the provider**

   This could be installing and operating provider software yourself, or signing up to a cloud service, or arranging for support from your internal IT department to have changes made to your in-house provider on your behalf.

3. **Register Krill with the provider**

   You will need to supply the Krill redirect URLs:[2]

   - https://yourdomain/auth/callback
   - https://yourdomain/ *(if the provider supports Connect RP-Initiated Logout 1.0)*

   You should receive back from the registration process three pieces of information that will be needed to configure Krill:

[1] Some provider specific information regarding claims can be found at the following links: Microsoft Azure Active Directory (here and here), Amazon Cognito (here)

[2] Alternatively your provider may support wildcard redirect URLs in which case you can supply https://yourdomain/*. However wildcard URLs are not advised as they could potentially be abused to redirect requests to other locations.

- The provider OpenID Connect Discovery 1.0 issuer URL[3]
- A client ID
- A client secret

4. **Create users, groups and/or claims in the provider**

   If all of your users will have admin rights in Krill you can ignore groups and claims and just create users.

   If however you want some users to have different rights than other users you will need to configure your provider to include some hint about the role that a user should have in the claims data that it sends to Krill.

   The manner in which this is setup varies greatly by provider. With Keycloak for example you have direct control over the claim data that is exposed to the OpenID Connect client and have multiple different ways to tell Krill via the claims data which role each user should have in Krill.

   With Azure Active Directory however you are by default limited to only being able to expose claims that it defines or to add users to groups. The group memberships can be exposed as claim data and Krill can parse the group data and match against it.

5. **Configure additional provider features**

   - How long are the tokens issued by the provider valid for? Can the provider issue refresh tokens? These properties affect how long a user can remain logged in to Krill.
   - Ensure that the provider has a real TLS certificate, or for in-house certificates you will need a copy of the Certificate Authority root certificate so that you can configure Krill to trust it. If neither are possible you can configure Krill to trust the insecure certificate anyway, but this is not advised.
   - Do you need to configure the provider to ensure that the claims you want to use will be sent to Krill?

6. **Configure Krill**

   Lastly, add the issuer URL, client ID and client secret to `krill.conf` and if necessary configure any claim mapping rules to instruct Krill how to obtain role information from the claims data that it will be sent.

   You may also need to use some of the other OpenID Connect specific configuration settings that Krill offers. For example to use the Amazon Cognito logout endpoint you have to configure that manually.

   Tip: The `krill.conf` file contains example configurations for providers that Krill has been tested with.

[3] A correct URL will either end in /.well-known/openid-configuration or should have that appended to it, e.g. the Google issuer URL is: https://accounts.google.com/.well-known/openid-configuration

Using Keycloak

In this section you will see how to setup Keycloak as an OpenID Connect provider for Krill.

The following steps are required to use OpenID Connect Users in your Krill setup.

1. Decide on the settings to be configured.

For this example let's assume we want to configure the following users:

Username	Email	Password	Role
joe@example.com	joe@example.com	dFdsapE5	admin
sally	sally@example.com	wdGypnx5	readonly
dave_the_octopus	dave@example.com	qnky8Zuj	readwrite

And let's assume that we are going to use a local Docker Keycloak container as our OpenID Connect provider which will be running at https://localhost:8443/.

---

2. Configure the provider

Let's walk through configuring the provider step by step:

Download and run Keycloak

```
$ sudo docker run \
    --detach \
    --name keycloak \
    --publish 8443:8443 \
    --env KEYCLOAK_USER=admin \
    --env KEYCLOAK_PASSWORD=password \
    --env DB_VENDOR=h2 quay.io/keycloak/keycloak:12.0.4
```

Warning: Do **NOT** run Keycloak like this in production. This command instructs Keycloak to use an in-memory H2 database which is convenient for demonstration and testing purposes but should not be used in a production setting.

Follow the logs until Keycloak is ready:

```
$ docker logs --follow keycloak
...
14:31:20,766 INFO  [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: Keycloak 12.0.4 (WildFly Core 13.0.3.Final) started in 23954ms - Started 687 of 972 services (687 services are lazy, passive or on-demand)
14:31:20,768 INFO  [org.jboss.as] (Controller Boot Thread) WFLYSRV0060: Http management interface listening on http://127.0.0.1:9990/management
14:31:20,769 INFO  [org.jboss.as] (Controller Boot Thread) WFLYSRV0051: Admin console listening on http://127.0.0.1:9990
```

Login to the Keycloak admin UI

- Browse to https://localhost:8443/.
- Accept the self-signed TLS certificate.
- Click on *Administration Console*.
- Login as user *admin* password *password*.

Create a realm

Note: A realm is a Keycloak concept and is a good example of how providers differ in what needs to be done to set them up.

- Hover over *Master* in the top left and click on the *Add Realm* button that appears.
- Set the field values as follows then click *Create*:

Field	Value
Name	*krill*

Create a client application

Tip: This is where we register Krill with the OpenID Connect provider.

Continuing in the KeyCloak web UI with realm set to *krill*:

- Click *Clients* (top left) then *Create* (top right).
- Set the field values as follows then click *Save*:

Field	Value
Client ID	*krill*

- On the *Settings* tab that is shown next set the field values as follows then click *Save* at the bottom.

Field	Value
Access Type	*confidential*[4]
Valid Redirect URIs	*https://localhost:3000/**[5]

- Generate credentials for Krill to use:
  - Open the *Credentials* tab (at the top).
  - Copy the *Secret* value somewhere safe, we'll need it later.

Configure a role mapper

Tip: This is where we create custom claims that Krill can detect and use to determine which rights in Krill to assign to the user.

- Open the *Mappers* tab (at the top) and then click *Create*.
- Set field values as follows then click *Save* at the bottom:

Field	Value
Name	*krill_role*
Mapper Type	*User Attribute*
User Attribute	*role*
Token Claim Name	*role*
Claim JSON Type	*String*

[4] Krill is an OAuth 2.0 "Confidential Client" as defined in RFC 6749 Section 2.1.

[5] We could configure this explicitly as two separate redirect URLs: https://localhost:3000/auth/callback (for post-login) and https://localhost:3000/ (for post-logout). However, as this is a localhost demo and Keycloak supports wildcard redirect URLs we can keep it simple in this case.

Create the users

- Click *Users* (on the left) then click *Add User* (top right).
- Set field values as follows then click *Save* at the bottom:

Field	Value
Username	*<THE USERS NAME>*
Email[6]	*<THE USERS EMAIL ADDRESS>*

- Open the *Credentials* tab and set the field values as follows:

Field	Value
Password	*<THE USERS PASSWORD>*
Password Confirmation	*<THE USERS PASSWORD>*

- Leave *Temporary* set to *ON*.[7]
- Click *Set Password*.
- When asked *"Are you sure you want to set a password for this user?"* click *Set password*.
- Open the *Attributes* tab.
  - Enter Key *role* with value *<THE USERS ROLE>* and press *Add*.
  - Click *Save* at the bottom.

Repeat the above adding the other users.

---

3. Configure Krill

Add the following to your `krill.conf` file: (remove or comment out any existing `auth_type` line)

```
auth_type = "openid-connect"

[auth_openidconnect]
issuer_url = "https://localhost:8443/auth/realms/krill"
client_id = "krill"
client_secret = "<SECRET VALUE SAVED EARLIER>"
insecure = true  [8]
```

---

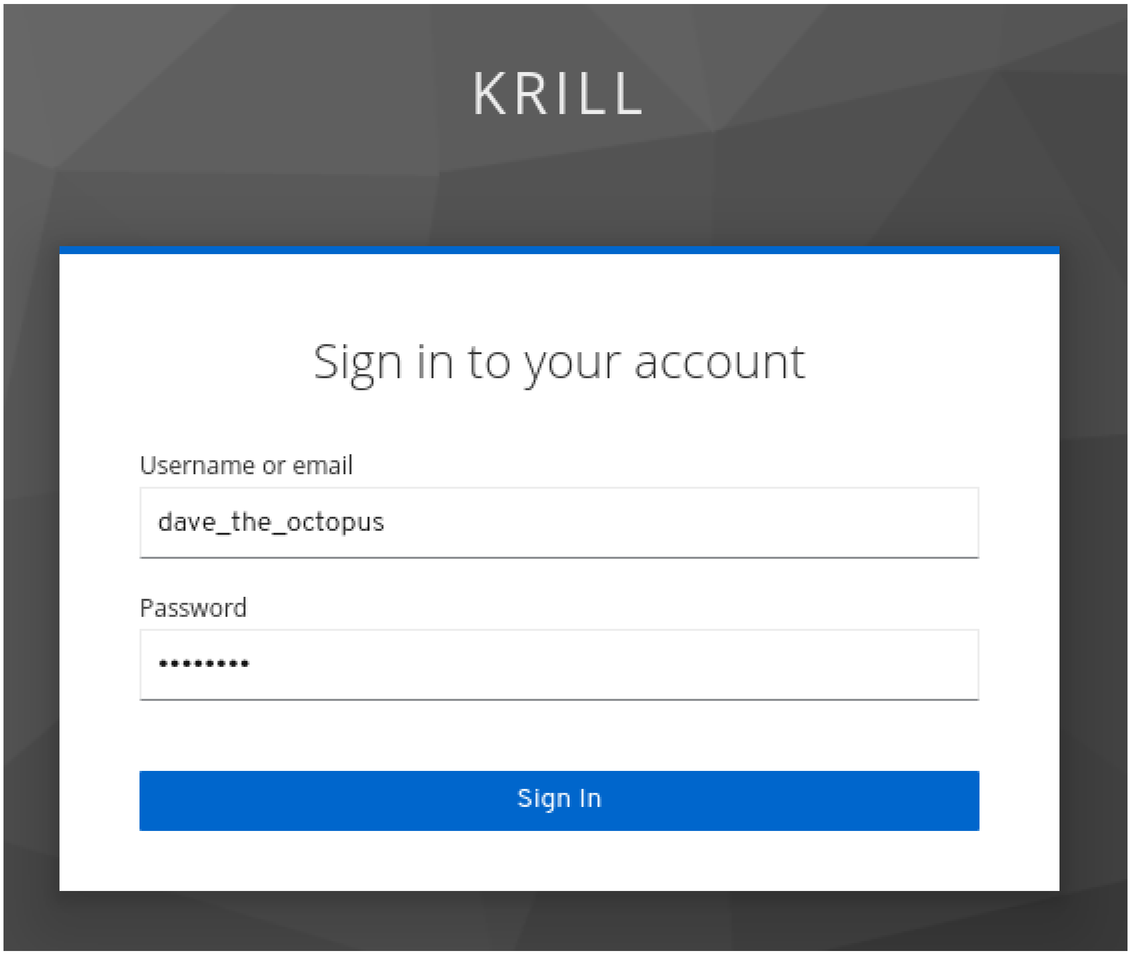
[6] By default Krill expects there to be an "email" claim in the ID Token response from the provider. If we didn't setup an email here we would need to define a claim mapping so that Krill could extract the *Username* value that we provide from some other claim field. In the case of Keycloak that would be the *preferred_username* field. We'll revisit this topic later.

[7] This is a good example of where using an OpenID Connect provider has benefits over using *Config File Users*. By leaving *Temporary* set to *ON*, Keycloak will require the user to change their password on first login. Krill doesn't have this functionality itself. This doesn't remove the need to communicate an initial unique password securely to the user, but the opportunity for abuse is more limited and passwords are not so readily visible to the Krill operator.

[8] % Do **NOT** use this in a production setting. We have to set *insecure* to *true* in this demonstration because our Keycloak instance does not have a real TLS certificate. Without *insecure* set to *true* Krill would reject the insecure self-signed TLS certificate.

4. Go!

Restart Krill and browse to the Krill web user interface. Your users should now be able to login with the Keycloak login form.



KRILL

Sign in to your account

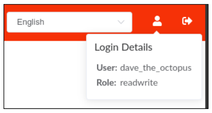
Username or email

dave_the_octopus

Password

•••••••••

Once logged in your users should have the role that you assigned to them:



Login Details

User: dave_the_octopus

Role: readwrite

With other providers

The OpenID Connect Users support within Krill is intended to be able to connect to and work with as many OpenID Connect providers as possible.

As such there are quite a few extra configuration options listed in `krill.conf` each of which is accompanied by documentation explaining what it does and how to use it.

Rather than duplicate that documentation here, instead we will focus on a few of the more difficult features to use and problems to overcome.

Understanding claims

Before we look at how to match claims let’s first take a look at what claims actually are and what it is that Krill has to match against.

Claims are part of the JSON data sent by the provider to the client. Krill must first extract this JSON data from the encoded, signed JWT data. The resulting claims look something like this:

```
{
  "iss": "http://server.example.com",
  "sub": "248289761001",
  "aud": "s6BhdRkqt3",
  "nonce": "n-0S6_WzA2Mj",
  "exp": 1311281970,
  "iat": 1311280970,
  "name": "Jane Doe",
  "given_name": "Jane",
  "family_name": "Doe",
  "gender": "female",
  "birthdate": "0000-10-31",
  "email": "janedoe@example.com",
  "picture": "http://example.com/janedoe/me.jpg"
}
```

Source: https://openid.net/specs/openid-connect-core-1_0.html#id_tokenExample

Thus if you were to configure Krill to use the “given_name” claim as the ID of the user in Krill, like so:

```
[auth_openidconnect.claims]
id = { jmespath="given_name" }
```

Then in this example Krill would use the value “Jane” as the ID of the user logged in to Krill.

Matching claims by name

Imagine that you want to show users by their name in the Krill web user interface and not by their email address, and that you know that the full name is available in a claim called *name*.

This can be achieved using a config section that looks like this in `krill.conf`:

```
[auth_openidconnect.claims]
id = { jmespath="name" }
```

This tells Krill to search all of the claim data it receives for a field called *name* and use that as the ID for the user in Krill. This ID will also be logged in the Krill event history as the actor responsible for any events that they caused.h

What is JMESPath? According to https://jmespath.org/:

> *"JMESPath is a query language for JSON."*

JSON is the format that OpenID Connect claim data is provided in by the provider. JMESPath can therefore be used to tell Krill which particular part from within the JSON it should use.

This is a very trivial example of the power of JMESPath. You can find out more about it at the https://jmespath.org/ website and in `krill.conf`. Krill comes with a couple of extensions to JMESPath syntax which are also documented in `krill.conf`.

Matching claims by value

Imagine that your users already exist in an OpenID Connect compatible identity provider and that the only distinguishing feature that you can use to assign them admin or some other role within Krill is their group membership. Now imagine that these groups do not have nice friendly names but instead are identified by an array of UUIDs!

How do you tell Krill which users should have readonly access and which users should be have readwrite access?

This is actually a real situation you can encounter with Azure Active Directory. JMESPath can also be used to handle this scenario, albeit with a much more complicated expression:

```
[auth_openidconnect.claims]
ro_role = { jmespath="resub(groups[?@ == 'gggggggg-gggg-gggg-gggg-gggggggggggg'] | [0],
↪'^.+$', 'readonly')", dest="role" }
rw_role = { jmespath="resub(groups[?@ == 'hhhhhhhh-hhhh-hhhh-hhhh-hhhhhhhhhhhh'] | [0],
↪'^.+$', 'readwrite')", dest="role" }
```

Let's break the *ro_role* claim mapping rule down:

- *gggg* and *hhhh* values represent the UUIDs of the groups to find in a claim array called *groups*.
- The *resub* JMESPath function is a Krill extension to JMESPath that performs regular expression based substitution.
- *groups[?@ == '… ']* finds all entries in the *groups* array that match the specified UUID.
- We then assume that there is only ever zero or one matches and just use the first match | *[0]* found.
- Then we instruct Krill to take the entire value with ^.+$.
- And to replace it with the value *readonly*.
- Finally, instead of assigning the value *readonly* to the user attribute *ro_role*, *dest* is used to instead store *readonly* in a user attribute called *role*.

As *role* is the user attribute that the Krill authorization policy engine looks at by default this will cause the user to be assigned the readonly role if their user is a member of the group with the UUID value that represents the “readonly” group!

If we had only one rule we could write *role* on the left, but as we have two rules that both try to provide a value for the same user attribute and the keys on the left of the = must be unique, we use the *dest* trick to map any value found to the *role* user attribute.

Matching claims by partial value

Now imagine that the group membership is instead expressed not as array elements that each exactly match some group name or UUID that we can look for, but that each array element is a long string composed of *key=value* comma separated pairs.

This can happen when the identity provider expresses group memberships in LDAP X.500 format (see RFC 2253 Lightweight Directory Access Protocol (v3): UTF-8 String Representation of Distinguished Names).

For example you might see something like `CN=Joe Bloggs,OU=NetworkTeam-Admins,DC=mycorp.com`, representing a user called Joe who is in the administrators group of the networking team of a company called mycorp.com.

Hopefully you’ll only need simple rules but also equally hopefully if you need more powerful matching Krill will be up to the task. For example, here’s a more complicated rule:

```
dynamic_role = { jmespath="resub(memberof[?starts_with(@, 'CN=DL-Krill-')] | [0], '^
↪CN=DL-Krill-(?P<role>[^-,]+).+', '$role')" }
```

This rule will match elements of an array called *memberof* whose value starts with `CN=DL-Krill-`, and wlll then extract just the part after that upto a comma or dash, and will use that captured value as the Krill `role` user attribute!

Matching claims to config values (aka ‘hybrid’ mode)

Usually when defining a claim mapping there is no need to define the source of the claim. Krill will search all of the different OpenID Connect provider claim sources that it supports (standard and additional claims in both the ID Token and User Info responses) for a matching claim.

However, if needed you can specify the claim source explicitly on a per claim basis. Possible uses for this include:

- Selecting the right claim when the same claim name exists in more than one claim source but with different values.
- Defining user attributes in the Krill configuration when the claim values cannot be configured in the provider (perhaps due to lack of support by or access to the provider). This is known as hybrid mode because it causes Krill to use a hybrid of OpenID Connect provider for authentication and config file defined user attributes for authorization.

When defining a claim mapping we have so far seen `jmespath` and `dest` settings, but there is also a `source` setting. The source can be set to one of the following values:

- `config-file`
- `id-token-standard-claim`
- `id-token-additional-claim`
- `user-info-standard-claim`
- `user-info-additional-claim`

The first one is the really interesting one. The rest should hopefully never be needed as by default Krill searches all of the possible OpenID Connect provider claim sources that it supports.

When using the `config-file` source there are two changes in the way that Krill looks up the claim value:

1. The `jmespath` setting is not used. Instead an attribute with the same name as the TOML key of the claim mapping is looked for on the user.
2. The user attributes are taken from a config file entry with the `id` of the current user is looked up in the `[auth_users]` config file section.

Note that the `id` of the current user is still determined by a normal OpenID Connect claim lookup, i.e. by default the `email` value reported by the provider for the user is used unless you define a claim mapping for `id` explicitly.

For example, to identify users by the given name reported by the OpenID Connect provider, and to set their role using entries in `krill.conf` instead of basing the role on provider claim values, you could do something like this:

```
[auth_users]
"Joe Bloggs"  = { attributes={ role="admin" } }
"Sally Alley" = { attributes={ role="readonly" } }

[auth_openidconnect.claims]
id   = { jmespath="given_name" }
role = { source="config-file" }
```

This will cause a user that logs in via the OpenID Connect provider who has a `given_name` claim value of `Joe Bloggs` to be granted the `admin` role in Krill.

Requesting missing claims

If you find that expected claim data is indeed not being sent by the provider this may not be an issue with the provider, rather it may be that the provider requires that Krill ask to be sent those claims.

Look at the `extra_login_scopes` setting in `krill.conf`, at OpenID Connect Core 1.0 section 5.4 Requesting Claims using Scope Values and at the documentation for your provider. Try and determine if there is a particular "scope" value that should be sent by Krill that is not currently being sent.

Diagnosing login problems

If you think your OpenID Connect provider should be providing certain claims about your users but are not sure, or if you are not redirected properly to the OpenID Connect provider login page or are not redirected post-login back to Krill, you can increase the log level.

- `log_level = "debug"` will cause Krill to log more about what it is doing.
- `log_level = "trace"` will cause Krill to log OpenID Connect requests and responses.

Note however that some of the communication will be between your browser and the OpenID Connect provider and that will not be visible in the Krill logs. To monitor that you will need to use the network inspector tool of your browser to see the requests and responses being exchanged.

> **Warning:** Trace level logging is VERY verbose and can reveal sensitive information such as OAuth 2.0 Access Tokens and users profile data. Only enable trace level logging while investigating a problem. Normally it should be sufficient to use `log_level = "warn"`.

9.4 Custom Authorization Policies

New in version v0.9.0.

9.4.1 Introduction

Note: This is an advanced topic, you don't need this feature to get started with Named Users. If you are considering implementing a custom authorization policy we'd love to hear from you!

Custom authorization policies are a way of extending Krill by supplying one or more files containing rules that will be added to those used by Krill when deciding if a given action by a user should be permitted or denied.

9.4.2 Examples

Some examples showing the power of this can be seen in doc/policies directory in the Krill source code repository.

role-per-ca-demo

By default Krill lets you assign a role to a user that will be enforced for all of the actions that they take irrespective of the CA being worked with. The *role-per-ca-demo* example extends Krill so that a user can be given different roles for different CAs.

The demo also shows how to use new user attributes to influence authorization decisions, in this case by looking for a user attribute by the same name as the CA being worked with, and if found it uses the attribute value as the role that the user should have when working with that CA.

Finally, the demo demonstrates how to add new roles to Krill by adding two new roles that are more limited in power than the default roles in Krill:

- A *readonly*-like role that also has the right to update ROAs.
- A role that only permits a user to login and list CAs.

team-based-access-demo

The *team-based-access-demo* shows how one can define teams in the policy:

- Users can optionally belong to a team.
- Users can have a different role in the team than outside of it.
- Being a member of a team grants access to the CAs that the team works with.

The example works by defining the team names in the policy file. Each team is given a name and a list of CAs it works with. Krill is then extended to understand two new user attributes:

- *team* - which team a user belongs to
- *teamrole* - which role the user has in the team

9.4.3 Using custom policies

To use a custom policies there must be an `auth_policies` setting in `krill.conf` specifying the path to one ore more custom policy files to load on startup.

```
auth_type = "..."
auth_policies = [ "doc/policies/role-per-ca-demo.polar" ]
```

Warning: Krill will fail to start if a custom authorization policy file is syntactically invalid or if one of the self-checks in the policy fails.

Warning: Policy files should only be readable by Krill and trusted operating system user accounts.

Krill performs some basic sanity checks on startup to verify that its authorization policies are working as expected, but a malicious actor could make more subtle changes to the policy logic which may go undetected, like granting their own user elevated rights in Krill.

If a malicious user is able to write to the policy file they may however already be able to do much more significant damage than editing a policy file!

Note: Policy files are not reloaded if changed on disk while Krill is running.

For policies that only contain rules this is not a problem as they would not be expected to change very often, if ever.

However, for policies that define configuration in the policy file, such as the *team-based-access-demo*, changes to the policy configuration will not take effect until Krill is restarted.

9.4.4 Writing custom policies

Policies are written in the Polar language. The following articles from the Oso website can help you get started with Polar:

- The Polar Language
- Write Oso Policies (30 min)
- Polar Syntax Reference
- Rust Types in Polar

The core policies and permissions that Krill uses are embedded into Krill itself and cannot be changed. It is however possible to add new roles and to add new logic based around the value of custom user attributes.

Defining new roles

Krill roles are defined by `role_allow("rolename", action:  Permission)` Polar rules. The rule is tested if the role of the current user is "rolename". The current role definitions test if the requested action is in a set defined to be valid for that role.

Tip: You can see the built-in role and permission definitions in the Krill GitHub repository.

To define a new role that grants read only rights plus the right to update ROAs one could write the following Polar rule:

```
role_allow("roawrite", action: Permission)
    role_allow("readonly", action) or
    action = ROUTES_UPDATE;
```

This example is actually taken from the *role-per-ca-demo.polar* policy.

Defining new rules

Let's write a rule that completely prevents the update of ROAs.

When Oso does a permission check the search for a matching rule starts by matching rules of the form `allow(actor, action, resource)`.

Tip: "resource" in this context is a Polar term and should not be confused with the RPKI term "resource".

The Krill policy delegates from its *allow* rules immediately to a special `disallow(actor, action, resource)` rule. The only definition of the `disallow()` rule in Krill by default says `if false`, i.e. nothing is disallowed.

While technically you can prevent an action by `cut` -ing out of an `allow()` rule that is more specific than any other `allow()` rules, it's not always possible to ensure that your rule is the most specific match. That's where `disallow()` comes in handy.

Let's use `disallow()` to implement our rule.

Create a file called `no_roa_updates.polar` containing the following content:

```
# define our new rule: disallow all ROA updates
disallow(_, ROUTES_UPDATE, _);

# we could also write this more explicitly like so:
# disallow(_, ROUTES_UPDATE, _) if true;

# add a test to check that our new rule works by
# showing that an admin user can no longer update
# ROAs!
?= not allow(new Actor("test", { role: "admin" }), ROUTES_UPDATE, new Handle("some_ca"));
```

Let’s break this down:

- The _ character is Polar syntax for “match any”.
- Lines starting with # are comments.
- Lines starting with ?= defines self-test inline queries that will be executed when Krill starts. If a self-test inline query fails Krill will exit with an error.

The rule that we have created says that for any actor trying to update a ROA on any “resource” (i.e. Certificate Authority), succeed (i.e. disallow the attempt).

If we now set `auth_policies = [ "path/to/no_roa_updates.polar" ]` in our `krill.conf` file and restart Krill it will no longer be possible for anyone to update ROAs.

This is obviously not the most useful policy, but it demonstrates the idea :-)

Diagnosing issues

If a rule doesn’t work as expected a good way to investigate is to add more self-test inline queries.

If that fails you can set `log_level = "debug"` and set O/S environment variable `POLAR_LOG=1` when runnng Krill. This will cause a huge amount of internal Polar diagnostic logging which will show exactly which rules Polar evaluated in which order with which parameters and what the results were.

Note: Clients using the Krill REST API directly or via `krillc` cannot authenticate using named users, they can only authenticate using the secret token. If you need this capability please let us know.

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.

NIC.br, ARIN and APNIC provide publication as a service to their members.

10.1 Why run your own?

If your parent CA does not offer publication as a service, then you will need to run your own server. But another reason why you may want to run your own Publication Server is that it will allow you to delegate your CA's resources to your own child CAs - e.g. for business units - and allow your children to publish at your central repository as well.

In this model you will need to set up your CA as a *child under your parent*, and *set it up to publish* at your local Publication Server:

10.2 Install

Krill comes with an embedded Publication Server. You can use this to offer an *rfc*:8181 Publication Protocol service to your own CA, as well as remote CAs - for example CAs for relations that you delegated Internet Number Resources to.

In principle you can enable the Publication Server on the same Krill instance that you use to operate your CAs. But, it may be better to use a separate instance for this purpose. This will allow more fine grained access control to either instance, and it makes it somewhat easier to parse the log files in case of issues.

Here we will document a setup using a separate Publication Server instance.

10.3 Configure

Your Publication Server can use a very minimal configuration file, similar in style to the one used by the Krill CA server. You should configure the following settings:

```
# Choose your own secret for the authorization token for the CLI and API
admin_token =

# If you installed krill using a package, then the default data directory
```

(continues on next page)

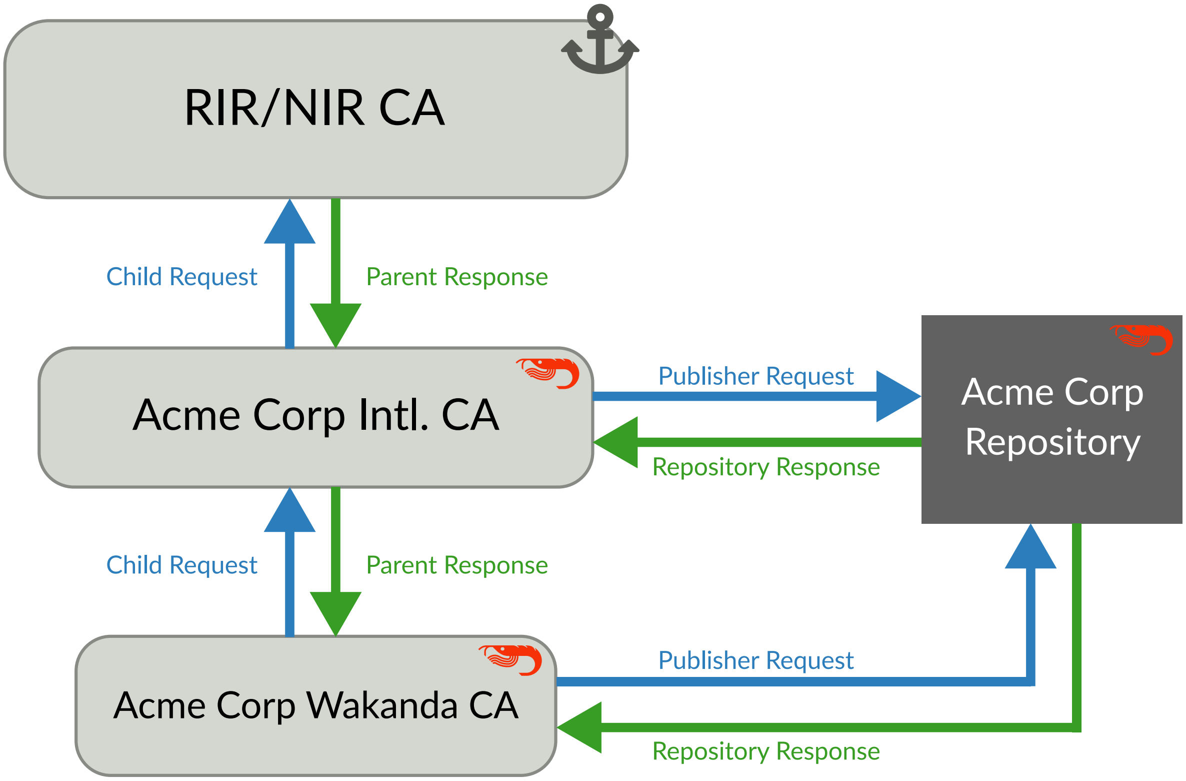


Fig. 1: Running a publication server for yourself and your children

(continued from previous page)

```
# and pid options are probably fine.
#
# If you installed krill by hand then you may wish to set the following:
data_dir = "/path/to/your/krillpubd/data/"
pid_file = "/path/to/your/krillpubd/krill.pid"

# Similarly, if you installed krill as a package it will use syslog, and
# this is probably desirable. If you want to use file logging you can
# configure this as follows, but note that there is no built-in log rotation
# in Krill.
log_type = "file"
log_file = "/path/to/your/krill.log"

# We recommend that you let the Krill daemon listen on localhost
# only, and use a proxy with proper HTTPS set up in front of it.
# However, you should configure the 'service_uri' property in your
# configuration file, so that your CAs will be able to connect to
# your server to publish. You should provide the 'base' hostname
# and optional port only. The actual URI that your CAs will connect
# to is: $service_uri/rfc8181
#
# NOTE: This can be a different base URI from the one used to
#       to serve the content of your repository - that URI is
#       is configured when you initialise your Publication Server
#       through the CLI.
service_uri = "https://krill-repo-server.example.com/"

# Disable the download of BGP information. Unless you are also using
# this server to host your CAs there is no need to keep this information
# in memory.
bgp_risdumps_enabled = false
```

If you want to review all options, you can download the `default config file`.

10.4 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. Note that 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.

10.5 Proxy for CLI and API

If you are okay with only using the `krillc` CLI on the machine where you run your Publication Server, then your safest option is to **not** proxy access to the API.

However, if you need to use the CLI or API from other machines, then you should proxy access to the path '/api' to Krill.

10.6 Configure the Repository

Note: We use the term **Publication Server** to describe the (Krill) server that CAs will connect to over the RFC 8181 protocol in order to publish their content. We use the term **Repository Server** to describe a server which makes this content available to RPKI Validators.

10.6.1 Synchronise Repository Data

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.

The Krill **Publication Server** 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.

Krill Sync

The preferred approach is to synchronise the data written by the Publication Server to your Repository Servers in a background process. A simple rsync command in crontab would work most of the time, but unfortunately that approach will lead to regular issues where inconsistent, or incomplete, data will be served to RPKI validators.

However, we have developed a separate tool krill-sync which can be used for this purpose. Krill-sync essentially works by retrieving consistent RRDP deltas from your back-end Publication Server to ensure that it can write consistent sets of data to disk for use by your Repository Servers.

Shared Data

Another option is to use some kind of shared file system (NFS, clustered filesystem, network storage) where the **Krill Publication Server** can write, and your **Repository Servers** can read.

If you go down this path, then make sure that the entire *$DATA_DIR/repo* is on a share. In particular: don't use a mount point at *$DATA_DIR/repo/rsync/current* as this directory is recreated by Krill whenever it publishes new data.

There can be issues with this approach with regards to availability and atomicity of updates to files on disk. The Krill Publication Server takes care to write files in the right order to avoid issues like Relying Parties retrieving a new notification.xml file *before* the snapshot or deltas are available. It will also write new files to temporary files and then rename them to avoid that partially written files are shown to users. However, dependent on the implementation details of the shared data these strategies may not work.

10.6.2 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
```

Note: we recommend that you use a limit for 'max connections'. Which value works best for you depends on your local situation, so you may want to monitor and tune this to your needs. Generally speaking though, it is better to limit the number of connections because RPKI validators will simply try to reconnect, rather then end up in a situation where your rsync server is unable to handle requests.

10.6.3 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.

Note: If desired, you can also use a **CDN** or your own caching infrastructure to reduce load. You could set it up to serve 'stale' content if your back-end system is unavailable to reduce the impact of short outages of your server. If you cache content make sure that you do not cache the main 'notification.xml' file (see more below) for longer than one minute (unless the back-end is unavailable). Other RRDP files will use unique names and can be cached for as long as you please.

10.6.4 Initialise Publication Server

You need to initialise your **Publication Server** using the base URIs as exposed by your **Repository Servers**. Use the following command, well, make sure the URIs reflect **your** setup of course:

Example CLI:

```
$ krillc pubserver server init --rrdp https://krillrepo.example.com/rrdp/ --rsync rsync:/
↪/krillrepo.example.com/repo/
```

There is probably no reason to use the API directly for this initialisation process, except perhaps for automation of test environments:

```
 $ krillc pubserver server init --rrdp https://krillrepo.example.com/rrdp/ --rsync␣
↪rsync://krillrepo.example.com/repo/ --api
 POST:
   https://krill-ui-dev.do.nlnetlabs.nl/api/v1/pubd/init
Headers:
```

(continues on next page)

(continued from previous page)

```
  content-type: application/json
  Authorization: Bearer secret
Body:
{
  "rrdp_base_uri": "https://krillrepo.example.com/rrdp/",
  "rsync_jail": "rsync://krillrepo.example.com/repo/"
}
```

Provided that you also set up your Repository Servers, and that they are in sync, you can now verify that the set up works. Try to get the 'notification.xml' file under your base URI, e.g. https://krillrepo.example.com/rrdp/notification.xml. Verify that access to your rsync server works by doing:

```
$ rsync --list-only rsync://krillrepo.example.com/repo/
```

If you are satisfied that things work, you can proceed to add publishers for your CAs. If not, then this is the moment to clear your Publication Server instance so that it can be re-initialised:

```
$ krillc pubserver server clear
```

Or through the API:

```
$ krillc pubserver server clear --api
DELETE:
  https://localhost:3000/api/v1/pubd/init
Headers:
  Authorization: Bearer secret
```

Note that you can NOT clear a Publication Server instance if it has any active publishers. Those CAs would not be aware that they would need to use new URIs on their certificates.

If you should end up in this situation, then you could set up a new Publication Server instead, and then migrate your existing CAs to that server, and then remove your current server altogether. Alternatively, you can remove all publishers from your server first, then clear and re-inialise it, and then add your CAs again and migrate them to this newly initialised version.

In short: it is best to avoid this and ensure that your are happy with the URIs used before adding publishers.

10.6.5 Repository Stats

You can review Publication Server stats, including the number of files and space used by publishers.

Example CLI:

```
$ krillc pubserver server stats
RRDP updated: 2021-04-08T06:40:01.337191+00:00
RRDP session: ec00a09d-45f9-43ff-9e4d-2739f5e05c05
RRDP serial:  29

Publisher, Objects, Size, Last Updated
testbed, 2, 3908, 2021-04-08T07:38:25.106777+00:00
ta, 3, 7592, 2021-04-08T07:38:25.557323+00:00
```

Example JSON response:

```
$ krillc pubserver server stats --format json
{
  "publishers": {
    "ta": {
      "objects": 3,
      "size": 7592,
      "last_update": "2021-04-08T07:38:25.557323Z"
    },
    "testbed": {
      "objects": 2,
      "size": 3908,
      "last_update": "2021-04-08T07:38:25.106777Z"
    }
  },
  "session": "ec00a09d-45f9-43ff-9e4d-2739f5e05c05",
  "serial": 29,
  "last_update": "2021-04-08T06:40:01.337191Z"
}
```

Example API:

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

10.7 Manage Publishers

10.7.1 Add a Publisher

In order to add a CA as a publisher you will need to get its **RFC 8183** Publisher Request XML. If you had no repository defined in your CA, you can get this XML from the UI, as described *here*.

The XML will include a so-called 'handle' - essentially the name that the CA likes to use for itself. This handle needs to be unique on the server side - we can't have all CAs calling themselves *mr-black*. For this reason the CLI offers an optional argument `--publisher` that allows overriding the handle in the reqeust with a locally unique value - e.g. a UUID.

After adding a publisher the server will respond with the unique **RFC 8183** Repository Response XML for this publisher. You can also retrieve this response again later (see below).

Example CLI:

```
$ krillc pubserver publishers add --publisher localname --request ./data/new-ca-
↪publisher-request.xml
<repository_response xmlns="http://www.hactrn.net/uris/rpki/rpki-setup/" version="1"␣
↪publisher_handle="localname" service_uri="https://localhost:3000/rfc8181/localname/"␣
↪sia_base="rsync://localhost/repo/localname/" rrdp_notification_uri="https://
↪localhost:3000/rrdp/notification.xml">
  <repository_bpki_ta>
↪MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyg4OEJBMzA2QkMzMUVFRkU3NzRDNzYzRUY1N0VBNUZEQ
↪c7x5sy3JbmUWaQHtkl6N9l8vcRlQQfhk0vwlVCHcQQrcMViJ5GmGtEjo7+Uf9e0TDA+rrkdqQkpQLcGRKjs1SZNqCRktubQU7Ndc0
↪gYAnYssX26kObXan0fD9rgv4aWK0Xzp5hwz1ECAwEAAaNTMFEwDwYDVR0TAQH/BAUwAwEB/
↪zAdBgNVHQ4EFgQUiLowa8Me7+d0x2PvV+pf3HoZ3iUwHwYDVR0jBBgwFoAUiLowa8Me7+d0x2PvV+pf3HoZ3iUwDQYJKoZIhvcNAQ
↪YZ1V[illegible]A8+IrYELSth4aCNSgPkhZfDL238MflAxptNRAoIeRGn8l3oSg4AUzBuScErwvBbHW[illegible]066nV0wzVFb+mL
↪GMiZHI/MwGZpj86Q/8wvyyw2C0b0ddWaoXwDyJjuxja0nHPDHVriJ8/
↪xsOfBk144n1zyP++apQXmXorCy4hs9GPyr+HGeoL6kNydDxdwzJLCqWW7u3wSnxjCJk+hfGq82qNm90ALv5PaOb58fDgWwBwuvTP0
↪</repository_bpki_ta>
```

(continues on next page)

(continued from previous page)

```
</repository_response>
```

Note that the API expects the JSON equivalent of the Publisher Request. But if there is demand then we can extend this in future to also accept the plain XML.

Example API:

```
$ krillc pubserver publishers add --publisher localname --request ./data/new-ca-
↪publisher-request.xml --api
POST:
  https://localhost:3000/api/v1/pubd/publishers
Headers:
  content-type: application/json
  Authorization: Bearer secret
Body:
{
  "tag": null,
  "publisher_handle": "localname",
  "id_cert":
↪"MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyhFRjJENzgwRkNCRkU1QjZBMkExMjA1OUM0MDlDN0M5
↪X3Ho5gC7uuKf4LYqbJj+Z9ltr/236/
↪hDYJfWMXZVcEuL+wUble1zhe2NKrgnAkpReVMSdiugoqZ9ICK2Fwkj5jCGc/qHiWOba7T78zfij8OlB/
↪dGlJvkAY8b/XTNKsTrLozi1uVAC8GqDrV5MEgY/NfzUvgA024yxx/rC6QBDEoBjnP7wDFiaZ2lwvL2beVYu6/
↪hVcXQzsVN+ijy7cGdkE6zi0meXJLTHPEpoA88hi3Pi+pIDBIQ3wTcpQIOqAq/
↪SZuh4dbZK7BV8MCAwEAAaNTMFEwDwYDVR0TAQH/BAUwAwEB/
↪zAdBgNVHQ4EFgQU7y14D8v+W2oqEgWcQJx8kndUGVYwHwYDVR0jBBgwFoAU7y14D8v+W2oqEgWcQJx8kndUGVYwDQYJKoZIhvcNAQ
↪gpJtONdgIWV1EqwEzhKKA2EP6tLDF9ejsdMFNYrYr+2hVWaoLsSuarfwfLFSgKDFqR6sh3ljYq6mIz9gdkjBOJsR9JyHFEtsDsRpf
↪YPMPVBpmG15Z9iKantzC1tck+E1xYW5awvj+YZqGVqyFdPJOZWmaYoS83kWvg4g4IucXTH6wwy23MQ7+0gyoK4wxfXRQmWjlXpLue
↪d7srQA4IxZCRGh9B+JdAIOKuXBA0nncmMJLQN8Qpxlz2bxKKAgXBLdoDqjbTDVbXTPM8YLRgc="
}
```

10.7.2 List Publishers

You can list all current publishers using the following command:

Example CLI:

```
$ krillc pubserver publishers list
Publishers: testbed, ta
```

JSON reponse:

```
$ krillc pubserver publishers list --format json
{
  "publishers": [
    {
      "handle": "testbed"
    },
    {
      "handle": "ta"
    }
  ]
}
```

Example API:

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

10.7.3 List Stale Publishers

You can list all publishers which have not published in a while. This may help to identify 3rd party publishers which are no longer active.

Example CLI:

```
$ krillc pubserver publishers stale --seconds 60
Publishers: testbed, ta
```

Example JSON response:

```
$ krillc pubserver publishers stale --seconds 60 --format json
{
  "publishers": [
    {
      "handle": "ta"
    },
    {
      "handle": "testbed"
    }
  ]
}
```

Example API:

```
$ krillc pubserver publishers stale --seconds 60 --api
GET:
  https://localhost:3000/api/v1/pubd/stale/60
Headers:
  Authorization: Bearer secret
```

10.7.4 Show a Publisher

Show details for a publisher, including the files that they published.

Example CLI:

```
$ krillc pubserver publishers show --publisher testbed
handle: testbed
id: E90C21734C2C370A91A8475CB4F0E75DA4D0F0BF
base uri: rsync://localhost/repo/testbed/
objects:
  rsync://localhost/repo/testbed/0/0BA5C132B94891CB2D3A89EDE12F01ACA4BCD3DC.crl
  rsync://localhost/repo/testbed/0/0BA5C132B94891CB2D3A89EDE12F01ACA4BCD3DC.mft
```

The JSON response also includes the full base64 encoded objects:

```
{
  "handle": "testbed",
  "id_cert":
↪"MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyhFOTBDMjE3MzRDMkMzNzBBOTFBODQ3NUNCNEYwRTc1
↪vVYxq1F1w2yQ/
↪VoTr1dvEHxJ+SDayMcFVktWCObiY8tcPhvWG+OdaX9ckDJhsOEEvdVEogwiGacNs7yXJPbqDBptJtbR8/
↪CauF9OqMqjkB/8xkGmBoY5OI/
↪V2832jkp7LPsbyET0RMQN7fgSpGbewvkaZVxGU3pHh5kT1nzPTXrwjxNMXgpunSEY7zR20vYCvsYYbxnSwFNbSMSL+Jgpa+HWPUc0
↪qlqlZK+tkppI2LkSBhTV5+n7cGA8ZsCAwEAAaNTMFEwDwYDVR0TAQH/BAUwAwEB/
↪zAdBgNVHQ4EFgQU6Qwhc0wsNwqRqEdctPDnXaTQ8L8wHwYDVR0jBBgwFoAU6Qwhc0wsNwqRqEdctPDnXaTQ8L8wDQYJKoZIhvcNAQ
↪nVmqDlK/1sWZNUXFWP4dt1wLTjDWnceyS8mI7Yx8dH/Fez60m4lp4dD45eeaXfbjP2cWnh3n/
↪PLGE70Nj+G0AnUhUmwiTl0H6Px1xn8fZouhv9MEheaZJA+M4NF77+Nmkp2P3WI4cvIS7Te7R/
↪7XpwSr29lVNtYjmRlrBDXx/bMFSgFL61mrtj/
↪l6G8OB40w+sAwO0XKUj1vUUpfIXc3ISCo0LNT9JSPcgy1SZWfmLb98q4HuvxekhkIPRzW7vlb/
↪NBXGarZmKc+HQjE2aXcIewhen2OoTSNda2jSSuEWZuWzZu0aMCKwFBNHLqs=",
  "base_uri": "rsync://localhost/repo/testbed/",
  "current_files": [
    {
      "base64":
↪"MIIJRAYJKoZIhvcNAQcCoIIJNTCCCTECAQMxDTALBglghkgBZQMEAgEwgZsGCyqGSIb3DQEJEAEaoIGLBIGIMIGFAgEJGA8yMDIx
↪Q8DtMdiFbVnO0AEdDZM4plkBKCCBs8wggbLMIIFs6ADAgECAhR4g6/Gg/
↪M8Ht3YdIxWaF5a54TZ4TANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEygwQkE1QzEzMkI5NDg5MUNCMkQzQTg5RURFMTJGMDFBQ0E
↪HXymRpwPLJn9Po21Dau0PC0D0OZviEgNmppY8hcqwExyXXgYM1zk7fyrk/
↪Nk8A7ue09CmhRAPogiwFQrPQc78OTWV27Bb2Yy5tPDIm34sROcxAKvgWneP8aOGvSWErD29MVn8M13gMfSkyaomnpe43QVIZ+5PGr
↪9Y5E49u2rpFFFn/LHzIbtV/AbKbYuetiXfXw6rEFxquJpmA4mfPm/
↪cRkpBMlAG2hW5lJOgZPxg0lxomzRwzldUR2hPAgMBAAGjggHZMIIB1TAdBgNVHQ4EFgQUhYGH74F3iP2GU+dFg556T4ehi88wHwYD
↪BAQDAgeAMF4GA1UdHwRXMFUwU6BRoE+GTXJzeW5jOi8vbG9jYWxob3N0L3JlcG8vdGVzdGJlZC8wLzBCQTVDMTMyQjk0ODkxQ0IyR
↪BA4wDDAKBggrBgEFBQcOAjAhBggrBgEFBQcBBwEB/
↪wQSMBAwBgQCAAEFADAGBAIAAgUAMBUGCCsGAQUFBwEIAQH/
↪BAYwBKACBQAwDQYJKoZIhvcNAQELBQADggEBAGEA7uZJ0/
↪OEo9cifRH5NMm8NETfk2fNIId6PMqjZKJym5j4D3EnuU72FgDuXSQuI/
↪ncOelq0Y0ABUzCjGYx41UhPYNzBLznw1WXgSDq7DviXx6hm60cpuIP++srMAWPR5yrBuX3WtJhDPmZMkOb9Z5OLs1A1It0Om1n9Sv
↪aQHQ+EWYZMP7CFULqVEY8qPMp1GDNUhI83YdGGmbbBNy0hSfY0CJio58v/
↪aPYRuUEcfBJQL0fi3O6PdFHF5I5hQFKzT6XX+09+UdxmAYd7TRidqIr3mBs5TbLd+e7eWwUdcut/
↪cUxd9mjWbrCBpHKDADT3KGpe6G4TSO9BFzU51zlKP9bJos7NfpJFyu75G7NyM1ebLD+2U2PImZaAnbyyVlGyXxUd8cmEO/
↪fweAzYk4eGvjtQIpnXpgtztdCkDE740KmYEn3XaVyNVvOk2oCyzRjkaKIPK70vRZ1HAW6IYSELLUbaFb6oSZJ9OfnVMq3Qragk/
↪Mo6Z",
      "uri": "rsync://localhost/repo/testbed/0/0BA5C132B94891CB2D3A89EDE12F01ACA4BCD3DC.
↪mft"
    },
    {
      "base64":
↪"MIICJTCCAQ0CAQEwDQYJKoZIhvcNAQELBQAwMzExMC8GA1UEAxMoMEJBNUMxMzJCOTQ4OTFDQjJEM0E4OUVERTEyRjAxQUNBNEJD
↪biiV6VFw0yMjA0MDYxMTUxMTNaMCUCFGSYKUXTAY0eHzm8+Q1j0UZa4f36Fw0yMjA0MDYxMTUxMzFaMCUCFEU/
↪b+tRv1ToUuMk3g3kEbEFv2PIFw0yMjA0MDYxMTUxMTZaoC8wLTAfBgNVHSMEGDAWgBQLpcEyuUiRyy06ie3hLwGspLzT3DAKBgNVH
↪qhiDAVWooDuqqUmdZNrSebYiIUOuLnuhARcWXoIOAU2UwalGX1Wbn3wPaQJ/
↪60PMd5FWEf1JHYp8mUeSWu76E13WEtfDZYszZEGnwyLMt2vEcUauhbzVh4pxf4Yv18PWWqM5xDexc7MjADAwP5Ud8VSjPCs/
↪Cr5M3fv2hLMLqyiyB47i+5fEewmh2IC3PDpo5rpHd5rV2M//BhJrs58a50MKZha43cT7q7qhhPQXXlg==",
      "uri": "rsync://localhost/repo/testbed/0/0BA5C132B94891CB2D3A89EDE12F01ACA4BCD3DC.
↪crl"
    }
```

(continues on next page)

(continued from previous page)

```
  ]
}
```

Example API:

```
$ krillc pubserver publishers show --publisher testbed --api
GET:
  https://localhost:3000/api/v1/pubd/publishers/testbed
Headers:
  Authorization: Bearer secret
```

10.7.5 Remove a Publisher

You can remove a publisher altogether. Doing so will also remove all their current content.

Example CLI:

```
% krillc pubserver publishers remove --publisher publisher
```

Example API:

```
$ krillc pubserver publishers remove --publisher publisher --api
DELETE:
  https://localhost:3000/api/v1/pubd/publishers/publisher
Headers:
  Authorization: Bearer secret
```

If you try to remove an unknown publisher, you will get an error:

```
$ krillc pubserver publishers remove --publisher publisher --format json
Http client error: Status: 404 Not Found, ErrorResponse: {"label":"pub-unknown","msg":
↪"Unknown publisher 'publisher'","args":{"publisher":"publisher"}}
```

10.8 Migrate existing Krill CAs

If you have an existing Krill CA that is currently publishing under another publication server, then you can *migrate it to using a new repository*.

CHAPTER

ELEVEN

DELEGATE TO CHILD CAS

Krill supports delegating resources from your CA(s) to so-called child CAs. This function is primarily used by National Internet Registries (NIRs) that use Krill for their RPKI service. Most non-registry organisations will have no need for this function, as they simply have no members or customers to delegate resources to.

However, this function may still come in useful for example for larger organisations with many resources and complex organisational structure or customers who are in charge of using some of their IP or ASN resources.

There is no UI support for managing child CAs, but you can use the CLI *krillc children* subcommands to achieve this:

```
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        Show the RFC8183 Parent Response XML
    connections     Show connections stats for children of a CA
    suspend         Suspend a child CA: hide certificate(s) issued to child
    unsuspend       Suspend a child CA: republish certificate(s) issued to child
    remove          Remove an existing child from a CA
```

CHAPTER

TWELVE

KEY ROLLOVER

Krill supports the **RFC 6489** RPKI Certification Authority Key Rollover process. In a nutshell this process allows RPKI CAs to replace their key in such a way that the content of all 'objects', like ROAs and possibly certificates issued to child CAs, is preserved in re-issued objects under the new key, without noticeable interruptions to RPKI validators.

Before we can dive in to key rollovers we need to take a step back and talk a bit about RPKI CA certificates and keys.

For most users their CA will have only one parent CA and only one key and CA certificate under that parent. But, krill supports having multiple so-called "Resource Classes" under a parent. The term Resource Class stems from **RFC 6492** - essentially you can think of these as a way to to group a set of resources that can appear on a single certificate. This construct is needed because RPKI CA certificates can have only one signing parent CA certificate. So, if your parent received resources on different certificate (presumably from different sources), then they cannot sign a single certificate to you with all those resources. They would have to give you a signed certificate under each of their own certificates with the applicable resources.

Furthermore, Krill also supports the notion of having multiple parent CAs. Conceptually this is only a small leap from having to deal with potentially multiple Resource Classes under a single parent. Under the hood it's all just more Resource Classes to Krill - it will just remember which parent to talk to in relation to each of them. Each resource class has its own key, or during a key rollover: keys.

12.1 Quick Guide to Key Rollovers

If you want to understand the background of key rollovers better, then we urge you to read the section below this one. Here we will just give you the quick gist of it.

If you want to do a key rollover for your CA, you will need to run two CLI commands.

First you need to initialise a new key to start the process:

```
krillc keyroll init
```

Then, you should wait 24 hours and before activating the new key and retiring the old:

```
krillc keyroll activate
```

Caveats:

- The `init` command will have no effect if your CA is in the middle of a rollover
- The `activate` command will have no effect if your CA does not have a new key

Your ROAs and possible other objects, such as CA certificates delegated to child CAs if you have those, will be safe during a rollover. They will be re-issued under the new key when you run the `activate` command.

12.2 Key Life Cycle Background

The key life cycle for a Resource Class has the following possible stages:

- pending
- active
- roll phase 1: pending and active key
- roll phase 2: new and active key
- roll phase 3: active and old key
- Pending

The 'pending' state indicates that a parent has told your CA that it is entitled to resources under a Resource Class hitherto unknown to your CA. When this happens Krill will create a new local Resource Class associated with this parent with a fresh key pair and a 'pending' Certificate Sign Request (CSR).

This stage is usually short-lived, because it immediately triggers that the CSR is sent to the parent. However, it needs to exist in order for Krill to deal with the possibility that the parent is unreachable or unresponsive to the CSR right after it was told about this entitlement.

- Active

The 'active' state is the normal stable state for keys under a Resource Class. It indicates that Krill has a single key under a resource class and it has received a certificate for it from its parent.

Krill will continue to query the parent for entitlements and in case there is a change in eligible resources or certificate validity it will create a CSR which is sent to the parent. The key as such remains in the 'active' state even if there are pending CSRs.

At this point we should probably also mention that if a Resource Class no longer appears in a parent's **RFC 6492** list response, Krill will simply clean up the lost resource class and all its (one or more) keys in whatever state they happen to be, and withdraw any objects published.

- roll phase 1: pending and active key

This state indicates that key rollover was initiated for a Resource Class. This can only be done for Resource Classes that are in an 'active' state. In other words: if your Resource Class is in the middle of a key rollover, then that has to be finished before you can initialise a new rollover.

You can use the following CLI command to start this process for all your eligible Resource Classes:

```
krillc keyroll init
```

When your Resource Class enters this stage, it will generate a new key and corresponding CSR. This phase is normally short-lived, because as above Krill will immediately send the CSR(s) to the appropriate parent(s).

- roll phase 2: new and active key

This state indicates that we received a new certificate for the 'new' key in the Resource Class. In conformance with **RFC 6489** Krill will now start publishing a CRL and manifest for this key, but it will continue to publish all of its objects such as ROAs under the previous, still 'active' key.

You can check whether your CA has reached this stage by running `krillc show`. This will print a section for **each** of your Resource Classes with their current 'state'. For example:

```
Resource Class: 0
Parent: testbed
State: roll phase 2: new and active key    Resources:
```

(continues on next page)

(continued from previous page)

```
    ASNs:
    IPv4: 192.168.0.0/16
    IPv6:
```

- roll phase 3: active and old key

You can complete your key rollover for any Resource Class that is currently in phase 2 by issuing the following CLI command:

```
krillc keyroll activate
```

Note that according to **RFC 6489** you should wait **at least 24 hours** before initiating this step.

This stage will trigger that the ‘new’ key is activated. All objects, like ROAs, which were issued under the previous ‘active’ key will now be published under that new key. Furthermore Krill will generate a revocation revocation request for the previous active key. But, until it is indeed confirmed to be revoked by the parent Krill will continue to issue a CRL and manifest, but no other objects for it.

This stage should be short-lived. The revocation request is sent to the parent immediately. But it exists in order to deal with a possible failure to communicate with the parent when the revocation request is sent. In that case Krill will continue to try in the background. As soon as the old key is revoked Krill will remove it. After this has been done there is only one key again, and it’s ‘active’.

CHAPTER

THIRTEEN

MIGRATE TO A NEW REPOSITORY

There may be times when you need to migrate your CA(s) to a new Repository. For example, you may want to do this if you were running your own Publication Server to provide a Repository, but you can now use a service provided by another organisation, e.g. your RIR. Another reason may be that you are running your own server, but you decided that you need to change your server setup.

Whatever your reason may be Krill supports migration to a new Repository by doing a specialised key rollover. Essentially it will allow you to configure a new Publication Server for your CA, at which point your CA will create a new key that will use the new server, and the base URIs it got from that server. Then you need to complete the key rollover (activate the new key), to remove the old key and the dependency on the old server.

There is no web UI support for this (yet), but you can do this using the CLI.

First, get the so-called **RFC 8183** Publisher Request XML for your CA:

```
krillc repo request
```

Then provide this XML to your new Publication Server (e.g. through a web portal). They should return an **RFC 8183** Repository Response XML file. Configure your CA to use this by running:

```
krillc repo configure --response </path/to/repo-response.xml>
```

Note: Krill will verify that it can successfully connect to the new server and perform an **RFC 8181** 'list' query to see its currently published objects, before accepting it. If this query fails you will get an error message and nothing will change for your CA.

As with normal key rollovers **RFC 6489** demands that you wait 24 hours before activating the new key, and removing the old one. However, there may be reasons why you need to move more quickly. In particular, if your old Publication Server or its Repository is unreachable. Run the following command to complete the process when you are ready:

```
krillc keyroll activate
```

Note: Krill will try to remove objects published at the old repository on completion of this process. This is a best effort attempt. If the old server is unresponsive, which may well have been the reason for migration, then it will not try again. Furthermore, while **RFC 8181** supports that a CA asks to withdraw all objects, it does not support that a CA informs a server that they no longer wish to be publish with them ever. I.e. it would be polite if you told your server to remove your CA as a publisher through another channel.

Note: There is no way to cancel the migration once it has been initiated. You will need to complete it, but then you can migrate again. Furthermore, because this relies on the key rollover process you cannot do this migration if there is a key rollover in progress. Krill will check for this, and refuse to do the repository migration in this case.

CHAPTER

FOURTEEN

HARDWARE SECURITY MODULES

14.1 Overview

Krill uses OpenSSL by default for key generation, and it stores the private keys thus generated in the *keys* directory under its data directory. For many organisations this will be good enough, provided of course that they ensure that access to the server and backup data is restricted.

If you want to take security one (big) step up, then you may want to use a Hardware Security Module (HSM) instead. HSMs are devices that can be used to safeguard and manage digital keys. They are designed to allow applications to use private keys, whilst ensuring that the actual private keys are never leaked outside of the device - not even to the application that is using the key.

However, be aware that your existing keys cannot (easily) be imported into your HSM. One could also argue that if a key had not been generated inside the actual HSM importing it will not increase security significantly, because there is no way of knowing for sure that the key was never leaked.

So, in order to use HSMs on an existing Krill installation you will have to perform a *key rollover*. This will ensure that the keys used for your RPKI CA certificates will use the HSM. But, unfortunately, there is no standards supported way to perform a key rollover for the identity key that Krill uses for its communication with parent (and child) CAs and its Publication Server. We are planning to work on a solution for this and will reach out to the IETF to seek standardisation.

14.2 Integrating with an HSM

Krill uses what it calls a “signer” to create and manage keys and to sign data with them. For the most part the Krill CA code is unaware of which signer implementation is associated with a key. For long-lived key-pairs such as the keys used in RPKI CA certificates (the resource certificate signed by the parent) and the ID certifcate used in the **RFC 6492** (provisioning) and **RFC 8181** (publication) protocols, it will only keep track of the public key identifier.

It falls to the “signer” then to map these public key identifier to an actual private key that can be used for signing operations. As mentioned, the default signer uses OpenSSL, in which case the actual private keys are simply stored on disk in the *keys* sub-directory of Krill data directory.

If you have access to a Hardware Security Module (HSM) you can instead configure “signer” implementations which will use the HSM to create and safeguard the private keys and perform any signing operation inside the HSM.

Note: Krill uses one-off signing keys for the EE certificates used in RPKI Signed Objects (such as ROAs and Manifests). These keys are generated whenever such an object is created, and used only once for signing, and then they are destroyed.

Such keys will **NOT** be created with, stored in or signed with the HSM. This is because it can be slow to generate, sign with and destroy one-off signing keys using an HSM.

On the other hand, because thes one-off keys are immediately destroyed, they do not need to be protected to the same degree as RPKI CA private keys, or CA identity keys. Assuming that the OpenSSL generation of a 2048 bit RSA key pair is secure enough.

14.3 Compatible HSMs

In theory Krill supports any HSM that is compatible with the PKCS#11 and/or the KMIP 1.2 standards. The HSM must already be setup and you must already be in possession of any access credentials which Krill will need to use to connect to the HSM.

Krill has been tested with the following (in alphabetical order):

Cryptographic Token Name	Tested Protocols	Tested Token Form	Test Results
AWS CloudHSM	PKCS#11	Cloud Service	#556
Kryptus kNET HSM v1.25.0	PKCS#11 & KMIP	Cloud Service	#554, #565
PyKMIP v0.10.0	KMIP	Software	#564
SoftHSMv2 v2.6.1	PKCS#11	Software	#553
Utimaco Security Server v4.45.3	PKCS#11	Software Simulator	#732
YubiHSM 2	PKCS#11	USB key	#555

In order to work with Krill the HSM must support the following operations:

PKCS#11	KMIP
C_CloseSession	Activate
C_DeleteObject	Create Key Pair
C_Finalize	Destroy
C_FindObjects	Get
C_FindObjectsFinal	Modify Attribute
C_FindObjectsInit	Query
C_GenerateKeyPair	Revoke
C_GetAttributeValue	Sign
C_GetInfo	
C_GetSlotInfo	
C_GetSlotList	
C_GetTokenInfo	
C_Initialize	
C_Login	
C_OpenSession	
C_Sign	
C_SignInit	

Krill can use a cluster of HSMs if the cluster appears to Krill as a single HSM, i.e. if Krill is not aware that the “single” HSM is in fact a cluster of HSMs.

14.3.1 PKCS#11 or KMIP?

PKCS#11 and KMIP are very similar in the capabilities they provide, so much so that there are commercial offerings that can bridge from one to the other, HSMs may offer support for both and both standards are maintained by OASIS. From a Krill server operation perspective however they are very different and each has its own pros and cons.

PKCS#11 works by delegating configuration, logging, administration, maintenance and upgrade of the interface with the HSM to a library file outside of Krill that Krill loads when it runs. You therefore have to manage and monitor this library and its logs as a separate component on the system running Krill. However, as a separate component it can connect in any way it needs to the backend which can be local or remote, or possibly even to a cluster of systems. Krill sees only the library, it has no way of knowing whether the backend is local or remote, singular or clustered. This means it also has no way of controlling how long the library will block to wait for a task to complete or how many requests it can handle at once or how many system resources it uses.

KMIP is arguably simpler to setup. With KMIP you only need to manage Krill and the HSM, there is no additional library component to manage as with PKCS#11. Krill itself communicates directly with the HSM and so all configuration, logging and resource usage is determined by Krill. In addition, monitoring is done by monitoring Krill itself. Krill connects to the KMIP server via TLS encrypted TCP and thus could also potentially be routed to one of many backend servers in a cluster, or the server could be a process running locally on the same host such as PyKMIP.

14.4 Scenarios

14.4.1 Fresh installation

With a fresh installation of Krill you can use the HSM from the start. No keys will be stored locally, instead all long-lived keys will be stored in the HSM.

14.4.2 Migrating to or between HSMs

Krill does not support migration of existing RPKI CA private keys from one signer to another. Instead you will need to perform a *key rollover* for each CA.

Note: Not all keys can be rolled. See the warning above about migration of ID keys used in parent/child and CA/publication server relationships.

To perform a key roll from one signer to another you must first change the `default_signer` in `krill.conf` to the new signer, and then restart Krill. After this point any new keys that are created by Krill, including the new key resulting from a rollover, will be created in using the new `default_signer`.

14.5 Configuration

See `krill.conf` for full details.

Note: Any changes to the configuration file will not take effect until Krill is restarted.

For backward compatibility if no `[[signers]]` sections exist in `krill.conf` then Krill will use the default OpenSSL signer for all signing related operations. To use a signer other than the default you must add one or more `[[signers]]` sections to your `krill.conf` file, one for each signer that you wish to define.

All signers must have a type and a name and properties specific to the type of signer.

The default configuration is equivalent to addding the following in krill.conf:

```
[[signers]]
type = "OpenSSL"
name = "Default OpenSSL signer"
```

14.5.1 Signer Roles

When configuring more than one signer, one may be designated the default_signer and another (or the same one) may be designated the one_off_signer. The default_signer is used to create all new keys, except in the case of one-off signing for which the one_off_signer signer will be used to create a new temporary key, sign with it then destroy it.

Specifying the default_signer and one_off_signer is done by referencing the name of the signer. For example the above is equivalent to:

```
default_signer = "Default OpenSSL signer"
one_off_signer = "Default OpenSSL signer"

[[signers]]
type = "OpenSSL"
name = "Default OpenSSL signer"
```

When only a single signer is defined it will implicitly be the default_signer. When defining more than one signer the default_signer must be set explicitly.

If the default_signer is not of type OpenSSL and is not explicitly set as the one_off_signer, an OpenSSL signer will automatically be used as the one_off_signer.

14.5.2 Configuring a PKCS#11 signer

Note: To actually use a PKCS#11 based signer you must first set it up according to the vendor's instructions. This may require creating additional configuration files outside of Krill, setting passwords, provisioning users, exporting shell environment variables for use by the library while running as part of the Krill process, creating or determining a slot ID or label, etc.

For a PKCS#11 signer you must specify the path to the dynamic library file for the HSM that was supplied by the HSM provider and a slot ID or label, and if needed, a user pin.

```
[[signers]]
type = "PKCS#11"
name = "SoftHSMv2 via PKCS#11"
lib_path = "/usr/local/lib/softhsm/libsofthsm2.so"
slot = 0x12a9f8f7
user_pin = "xxxx"                                       # optional
login = true                                            # optional, default = true
```

Note:

- If using a slot label rather than ID you can supply the label using slot = "my label".

- You can also supply an integer slot ID, e.g. `slot = 123456`.
- If your HSM does not require you to login you can set `login = false`.
- If your HSM requires you to supply a pin via an external key pad you can omit the `user_pin` setting.

14.5.3 Configuring a KMIP signer

Note: To actually use a KMIP based signer you must first set it up according to the vendors instructions. This may require setting up users and passwords and/or obtaining certificates in order to populate the associated settings in the `krill.conf` file.

For a KMIP signer you must specify the fully-qualified domain name (FQDN) or IP address of the host, and optionally other connection details such as port number, client certificate, server CA certificate, username and password.

```
[[signers]]
type = "KMIP"
name = "Kryptus via KMIP"
host = "my.hsm.example.com"
port = 5696                                             # optional, default = 5696
server_ca_cert_path = "/path/to/some/ca.pem"            # optional
client_cert_path = "/path/to/some/cert.pem"             # optional
client_cert_private_key_path = "/path/to/some/key.pem"  # optional
username = "user1"                                      # optional
password = "xxxxxx"                                     # optional
insecure = false                                        # optional
force = false                                           # optional
```

Note:

- `host` can also be an IP address.
- `insecure` will disable verification of any certificate presented by the server.
- `force` should only be used if the HSM fails to advertize support for a feature that Krill requires but actually the HSM **does** support the feature.

14.6 Signer Lifecycle

At startup Krill will announce the configured signers in its logs but will not yet attempt to connect to them. Only once a signing related operation needs to be performed will Krill attempt to connect to the signer.

If there is a problem connecting to a signer Krill will retry, unless the problem is fatal such as the signer lacking support for required operations. A problem with a signer will not stop Krill from running and continuing to serve the UI and API or from executing background tasks. Thus if some keys are owned by one signer that is reachable and another signer is not reachable, Krill will continue to operate correctly for operations involving the reachable signer.

On initial connection to a new signer Krill will create a “signer identity key” in the HSM. This serves to verify that the signer is able to create and sign with keys and in future that the signer is the one that owns keys attributed to it.

New keys are created by the `default_signer` unless they are one-off keys in which case they are created by the `one_off_signer`. Signing with a key is handled by the signer that possesses the key.

Note: Krill determines the signer that possesses a key by consulting a mapping that it keeps from key identifier to a Krill internal signer ID and associated metadata.

On initial connection to a signer it "binds" the internal representation of the connected signer to the matching internal signer ID and updates the metadata about the signer. It verifies that the internal signer ID corresponds to the backend by verifying the existence of a previously created "signer identity key" within the backend and that the backend is able to correctly sign with that key.

Krill is able to maintain the mapping between keys associated with a signer ID and the actual connected signer even if the name and server connection details in `krill.conf` are changed so you are free to rename the signer or replace the physical server by a (synchronized) spare or upgrade or change its IP address or the credentials used to access it and Krill will still know when connecting to it which keys it possesses.

Warning: If Krill is not configured to connect to the signer that possesses a key that Krill needs to sign with, or is unable to connect to it using the configured settings, then Krill will be unable to sign with that key!

One particular scenario to watch out for is when reconfiguring an existing Krill instance to use an HSM when that Krill instance already has at least one CA (and thus already created at least one key pair using OpenSSL).

In this scenario, if the changes to `krill.conf` to use the HSM define only the one signer (the HSM) and do NOT set that signer as the `one_off_signer`, then Krill will activate the default OpenSSL signer for one-off key signing and will use it to find the previously created OpenSSL keys.

If however the one and only HSM signer is also set as the `one_off_signer` then Krill will not activate the OpenSSL signer and so will not find the previously created OpenSSL keys. In this case you must explicitly add a `[[signers]]` block of `type = "OpenSSL"` with default settings thereby causing Krill to activate the default OpenSSL signer.

14.7 SoftHSMv2 Example

Let's see how to setup SoftHSMv2 with Krill. This example uses commands suitable for an Ubuntu operating system, for other operating systems you may need to use slightly different commands.

First, install and setup SoftHSM v2:

```
$ sudo apt install -y softhsm2
$ softhsm2-util --init-token --slot 0 --label "My token 1" --so-pin 1234 --pin 5678
```

Next add the following to your *krill.conf* file:

```
[[signers]]
type = "PKCS#11"
name = "SoftHSMv2"
lib_path = "/usr/lib/softhsm/libsofthsm2.so"
slot = "My token 1"
user_pin = 5678
```

Now (re)start Krill.

That's it! When you next create a CA Krill will create a key pair for it in SoftHSMv2 instead of using OpenSSL.

One way to inspect the keys stored inside OpenSSL is using the `pkcs11-tool` command:

```
$ sudo apt install -y opensc
$ pkcs11-tool --module /usr/lib/softhsm/libsofthsm2.so -O -p 5678
Using slot 0 with a present token (0x542bc831)
Public Key Object; RSA 2048 bits
  label:      Krill
  ID:         e83e96883ee73e69e0e57d54b6726c9d45f788c5
  Usage:      verify
  Access:     local
Public Key Object; RSA 2048 bits
  label:      Krill
  ID:         9ecd3796786c7a073d5384c155d8d475d103df74
  Usage:      verify
  Access:     local
...
```

14.8 Configuration Reference

The following configuration file description should give you all the pointers you need to get this setup working:

```
######################################################################################
#                                                                                    #
#                       ---==== SIGNER CONFIGURATION ====----                        #
#                                                                                    #
#       The settings below can be used to configure the signer used by Krill.        #
#                                                                                    #
######################################################################################

# Signers
# -------
#
# A signer is a cryptographic token, either hardware or software, local or remote,
# that can create RSA public/private key pairs and can sign data with the private key.
#
# Supported signer types
# ----------------------
#
# Krill supports three types of signer:
#
#   - OpenSSL based: Uses the OpenSSL library installed on the host O/S. On older
#     operating systems it might be that a newer version of OpenSSL than is supported
#     by the host O/S has been compiled into Krill itself and will be used instead.
#
#   - PKCS#11 based: Uses a PKCS#11 v2.20 conformant library file from the filesystem.
#     How the library handles the requests on behalf of Krill is library specific. A
#     library such as SoftHSMv2 contains all of the code needed to handle the request
#     and stores generated keys on the host filesystem. Libraries provided by well
#     known HSM vendors will dispatch requests to one or a cluster of hardware
#     security modules connected either physically or by network connection to the
#     host on which Krill is running.
#
#   - KMIP based: Makes TLS encrypted TCP connections to an operator specified server
```

(continues on next page)

(continued from previous page)

```
#     running a KMIP v1.2 conformant service.
#
# Key creation policy
# -------------------
#
# Krill creates keys at different times for different purposes. Some keys are fixed
# such as the identity key for the RFC 8183 defined provisioning protocol, others can
# be rolled (e.g. the keys used for RPKI CA certificates in resource classes) and
# still others are one-off keys (e.g. keys used for EE certificates in CMS) that are
# discarded after use.
#
# Signer roles
# ------------
#
# Signers can be assigned to roles to implement the desired policy. Roles are assigned
# by setting the following top level configuration file settings:
#
#   - default_signer: The signer will be used to generate new long-term key pairs.
#     Only one signer may be designated as the default. If only one signer is defined
#     it will be the default. If more than one signer is defined one must be
#     explicitly set as the default.
#
#   - one_off_signer: The signer will be used to generate, sign with and destroy
#     one-off key pairs. Only one signer may be designated as the oneoff signer. When
#     not specified an OpenSSL signer will be used for this.
#
# These settings must be set to the name of a single signer, e.g.:
#
#   default_signer = "My signer"
#
#   [[signers]]
#   type = "OpenSSL"
#   name = "My signer"
#
# Required capabiliites
# ---------------------
#
# When Krill first connects to a new signer it will verify that the signer meets its
# requirements. In particular it will require the signer to generate an RSA key pair
# and to demonstrate that it can sign data correctly using the generated private key.
#
# Config file settings
# --------------------
#
# At a minimum the "name" and "type" must be specified for a signer.
#
# One optional setting can also be set for all signers:
#
# - signer_probe_retry_seconds: When initially connecting to the signer on first use
#   after Krill startup, wait at least N seconds between attempts to connect and
#   test the signer for compatibility with Krill. Defaults to 30 seconds.
#
```

(continues on next page)

(continued from previous page)

```
# The remaining details that must be supplied to configure a signer vary by signer
# type and by specific implementation. For example an OpenSSL signer doesn't require
# a path to a library file to load, while a PKCS#11 signer does, and one PKCS#11
# vendor may require login by PIN code while another might allow operations to be
# performed with external PIN entry or no PIN entry at all.
#
# Default configuration
# ---------------------
#
# The default configuration is equivalent to:
#
#   [[signers]]
#   type = "OpenSSL"
#   name = "Default OpenSSL signer"
#
# Changing the configuration
# --------------------------
#
# The number, type, order, settings, names of signers can be changed at any time.
# Krill will apply the changes when next restarted. Via the use of identity key
# based signer binding Krill will still find the keys that it has created as long as
# the same backend is connected to, irrespective of name or connection details, and
# that the identity key in the signer has not been deleted.
#
# Warning about removing an in-use signer
# ---------------------------------------
#
# Removing a signer that owns keys that Krill is still using will prevent Krill from
# accessing those keys!
#
# Example configuration
# ---------------------
#
# Below is an example configuration. This example defines many signers but normally
# one would define only a single signer, or two signers if migrating from one signer
# to another.
#
#   default_signer = "SoftHSMv2 via PKCS#11"
#
#   [[signers]]
#   type = "OpenSSL"
#   name = "Signer 1"
#
#   [[signers]]
#   type = "OpenSSL"
#   name = "Signer 2"
#   keys_path = "/tmp/keys"
#
#   [[signers]]
#   type = "PKCS#11"
#   name = "Kryptus via PKCS#11"
#   lib_path = "/usr/local/lib/kryptus/libknetpkcs11_64/libkNETPKCS11.so"
```

(continues on next page)

(continued from previous page)

```
#   user_pin = "xxxxxx"
#   slot = 313129207
#
#   [[signers]]
#   type = "PKCS#11"
#   name = "SoftHSMv2 via PKCS#11"
#   lib_path = "/usr/local/lib/softhsm/libsofthsm2.so"
#   user_pin = "xxxx"
#   slot = 0x12a9f8f7
#
#   [[signers]]
#   type = "KMIP"
#   name = "Kryptus via KMIP"
#   host = "my.hsm.example.com"
#   port = 5696
#   server_ca_cert_path = "/path/to/some/ca.pem"
#   username = "user1"
#   password = "xxxxxx"


# OpenSSL signer configuration
# ----------------------------
#
# This signer uses the operating system provided OpenSSL library (or on older
# operating systems it may use a modern version of the OpenSSL library compiled into
# Krill itself) to generate keys, to sign data using them and to generate random
# values. Keys are persisted as files on disk in a dedicated directory.
#
# Key        Value Type   Default          Req'd  Description
# ====================================================================================
# keys_path  path string  "$datadir/keys"  No     The directory in which key files
#                                                 should be created.
#


# PKCS#11 signer configuration
#
# Krill interacts with a PKCS#11 v2.20 compatible cryptographic device via the Cryptoki
# interface which involves loading a library file from disk at runtime to which all
# cryptographic operations will be delegated. The library will in turn communicate
# with the actual cryptographic device.
#
# Note: The PKCS#11 library is not part of Krill nor is it supplied with Krill. Please
# consult the documentation for your PKCS#11 compatible cryptographic device to learn
# where you can find the .so library file and how to set up and configure it. For
# example when using SoftHSMv2 the library is commonly available at filesystem path
# /usr/lib/softhsm/libsofthsm2.so.
#
# Key        Value Type   Default          Req'd  Description
# ====================================================================================
# lib_path           path string  None     Yes    The path to the .so dynamic library
#                                                 file to load.
```

(continues on next page)

(continued from previous page)

```
# slot                integer or  None     Yes    An integer PKCS#11 "slot" ID or a
#                     string                      string "slot" label. Can also be
#                                                 given in hexadecimal, e.g. 0x12AB.
#                                                 When a label is given Krill will
#                                                 inspect all available slots and use
#                                                 the first slot whose label matches.
# ------------------------------------------------------------------------------------
# user_pin            string      None     No     The pin or password or secret value
#                                                 used to authenticate with the
#                                                 PKCS#11 provider. The format varies
#                                                 by provider, SoftHSMv2 uses numeric
#                                                 PINs such as "12345" while AWS
#                                                 CloudHSM expects this to be in the
#                                                 form "username:password".
# login               boolean     True     No     Whether the signer must be logged in
#                                                 to before performing other
#                                                 operations.
# ------------------------------------------------------------------------------------
# retry_seconds       integer     2        No     Wait N seconds before retrying a
#                                                 failed request.
# backoff_multiplier  float       1.5      No     How much longer to wait before retry
#                                                 N+1 compared to retry N.
# max_retry_seconds   integer     30       No     Stop retrying after N seconds.


# KMIP signer configuration
#
# Krill interacts with a KMIP v1.2 compatible cryptographic device via the TCP+TTLV
# protocol. This requires knowing the hostname, port number, and details required to
# authenticate with the provider.
#
# Key                 Value Type  Default  Req'd  Description
# ====================================================================================
# host                string      None     Yes    The domain name or IP address to
#                                                 connect to.
# port                integer     5696     No     The port number to connect to.
# ------------------------------------------------------------------------------------
# insecure            boolean     false    No     If true, do not verify the servers
#                                                 TLS certificate.
# force               boolean     false    No     If true, ignore server claims that
#                                                 it lacks functionality that we
#                                                 require. For example PyKMIP 0.10.0
#                                                 says it doesn't support operation
#                                                 ModifyAttribute but sending a
#                                                 modify attribute request succeeds.
# ------------------------------------------------------------------------------------
# server_cert_path                                File system paths to certificate
#                     string      None     No     files (in PEM format) for verifying
# server_ca_cert_path                             the identity of the server.
#                     string      None     No
# ------------------------------------------------------------------------------------
# client_cert_path                                File system paths to certificate and
```

(continues on next page)

(continued from previous page)

```
#                     string      None     No     key files (in PEM format) for
# client_cert_private_key_path                    proving our identity to the server.
#                     string      None     No
# ------------------------------------------------------------------------------------
# username            string      None     No     Credentials for authenticating with
# password            string      None     No     the server.
# ------------------------------------------------------------------------------------
# retry_seconds       integer     2        No     Wait N seconds before retrying a
#                                                 failed request.
# backoff_multiplier  float       1.5      No     How much longer to wait before retry
#                                                 N+1 compared to retry N.
# max_retry_seconds   integer     30       No     Stop retrying after N seconds.
# ------------------------------------------------------------------------------------
# connect_timeout_seconds                         Wait at most N seconds to make a TCP
#                     integer     5        No     connection to the KMIP server.
# read_timeout_seconds                            Wait at most N seconds for more
#                     integer     5        No     response bytes to be received from
#                                                 the KMIP server.
# write_timeout_seconds                           Wait at most N seconds to write more
#                     integer     5        No     request bytes to the connection to
#                                                 the KMIP server.
# max_use_seconds     integer     60*30    No     Don't use an idle connection to the
#                                                 KMIP server if it has been connected
#                                                 for at least N seconds.
# max_idle_seconds    integer     60*10    No     Close open connections to the KMIP
#                                                 server if not used in the last N
#                                                 seconds.
# ------------------------------------------------------------------------------------
# max_connections     integer     5        No     The maximum number of concurrent
#                                                 connections to permit to the server.
# max_response_bytes  integer     64*1024  No     The maximum number of response bytes
#                                                 to accept from the KMIP server, or
#                                                 otherwise treat the request as
#                                                 failed.
```

CHAPTER FIFTEEN

MANAGE BGPSEC ROUTER CERTIFICATES

Krill supports signing **RFC 8209** BGPSec Router Certificates as of release 0.10.0. These certificates are used publish the router keys for **RFC 8205** BGPSec protocol capable routers. Unfortunately, this protocol is not (yet) supported by many routers. We hope that by adding support for signing router certificates to the Krill CLI and API we can help support the future development and deployment of BGPSec. However, because BGPSec deployment is still lacking we have chosen not to support this in the UI at this time. But, of course, we are more than willing to add this in future if BGPSec deployment takes off and / or there is user demand for this.

The CLI commands are documented *here*.

CHAPTER

SIXTEEN

MANAGE ASPA OBJECTS

Important: This is an EXPERIMENTAL feature to support the discussion and development of the ASPA concept which is currently being discussed in the sidrops WG in the IETF. Do NOT use this in production environments.

To use this feature you will need to enable support for it when building the CLI. No UI support has been added yet.

16.1 Install CLI

You will need to install Krill from source and enable ASPA to ensure that the CLI binary gets built with additional subcommands needed for this feature:

```
cargo install krill --git https://github.com/NLnetLabs/krill \
                    --tag v0.10.0 \
                    --features aspa \
                    --locked
```

16.2 ASPA Configurations

As with ROA support, Krill lets operators *define* the ASPA *configurations* for which they want to have ASPA objects. The actual ASPA *objects* are then created by Krill under any parent where the 'customer AS' is in the set of received resources. I.e. if theoretically your CA would receive this same ASN under two different parents, then Krill would create an ASPA object with the same content under each.

Furthermore, just like with ROAs, Krill issues these objects with a default validity time of 52 weeks, and will automatically re-issue these objects 4 weeks before they would expire - as long as a configuration still exists and the customer ASN is held by your CA.

16.3 ASPA Configuration Notation

ASPA objects allow operators to specify a list of provider ASNs, in the sense of BGP rather than in terms of business relations, where their own ‘customer’ ASN can send updates. Providers can optionally be restricted to IPv4 or IPv6 only.

Krill uses the following notation style to make it easy to define such configurations when using the CLI:

```
AS65000 => AS65001, AS65002(v4), AS65003(v6)
AS65001 => <none>
```

Important: You can only have ONE ASPA configuration for each customer ASN. This is because Krill MUST (RFC) create a single ASPA object, for all provider ASNs. If you wish to have an explicit empty list of provider ASN, use ‘<none>’.

16.4 Add an ASPA

You can add ASPA definition using the following command:

```
$ krillc aspas add --aspa "AS65000 => AS65001, AS65002(v4), AS65003(v6)"
```

This uses the following API call:

16.5 List ASPAs

CLI:

```
$ krillc aspas list
AS65000 => AS65001, AS65002(v4), AS65003(v6)
```

API:

```
GET:
  https://localhost:3000/api/v1/cas/ca/aspas
Headers:
  Authorization: Bearer secret
```

JSON response:

```
$ krillc aspas list --format json
[
  {
    "customer": "AS65000",
    "providers": [
      "AS65001",
      "AS65002(v4)",
      "AS65003(v6)"
    ]
```

(continues on next page)

(continued from previous page)

```
  }
]
```

16.6 Update an ASPA

You can add or remove providers to/from the ASPA configuration for one of your customer ASNs:

Using the CLI:

```
$ krillc aspas update --customer AS65000 --add "AS65005" --remove "AS65001"
```

Or using the API:

```
krillc aspas update --customer AS65000 --add "AS65005" --remove "AS65001" --api
POST:
  https://localhost:3000/api/v1/cas/ca/aspas/as/AS65000
Headers:
  content-type: application/json
  Authorization: Bearer secret
Body:
{
  "added": [
    "AS65005"
  ],
  "removed": [
    "AS65001"
  ]
}
```

16.7 Remove an ASPA

You can remove the ASPA configuration for a given customer ASN.

Using the CLI:

```
$ krillc aspas remove --customer AS65000
```

Or using the API:

```
krillc aspas remove --customer AS65000 --api
POST:
  https://localhost:3000/api/v1/cas/ca/aspas
Headers:
  content-type: application/json
  Authorization: Bearer secret
Body:
{
  "add_or_replace": [],
  "remove": [
```

(continues on next page)

(continued from previous page)

```
    "AS65000"
  ]
}
```

CHAPTER

SEVENTEEN

RUNNING A KRILL TEST ENVIRONMENT

You do not need to run your own Krill Test Environment if you just want to try out Krill. You can simply install Krill on a test machine and set it up under the public Krill based RPKI testbed that NLnet Labs runs here: https://testbed.krill.cloud/index.html#/testbed

Read more about this in our blog.

But of course, you are perfectly welcome to run your own Test environment as well. That way you have total control over your test environment. This may be particularly useful for training purposes, and for testing code integration using the API or the RFC 8181 Publication Protocol or RFC 6492 Provisioning Protocol.

Here we will document how we set up a simple testbed. This is not a strict guide. You may want to do things differently and that would be fine, but we hope that this provides a useful walkthrough.

17.1 Install a Proxy Server

Here we use NGINX, but of course you can use an alternative if you prefer:

```
apt install nginx
cd /etc/nginx/sites-enabled
rm default
```

Then we created a configuration file that will proxy all traffic to our Krill server (which we will install in a moment):

/etc/nginx/sites-enabled/krill.example.org

```
server {
      server_name krill.example.org;
      client_max_body_size 100M;

      location / {
              proxy_pass https://localhost:3000/;
      }

  listen 80;
}
```

Restart nginx and we have our proxy server set up:

```
systemctl restart nginx
```

17.2 Set up Letsencrypt

This is as easy as installing the *certbot* and running its interactive script to setup integration with NGINX:

```
apt install certbot
apt install python3-certbot-nginx
certbot --nginx
```

17.3 Install Krill

We use our debian package to install Krill on a test system:

```
echo "deb [arch=amd64] https://packages.nlnetlabs.nl/linux/ubuntu/ focal main" >> /etc/
↪apt/sources.list
apt-get update
apt-get install krill
```

You can read more about Krill installation options here.

17.4 Configure Testbed

Before you start Krill edit */etc/krill.conf*. Make sure that you are okay with the autogenerated random value for *auth_token*, or override it with something to your liking.

If you need your testbed to be accessible from other CAs, either for publication or to be added as a child under the 'testbed' CA, then make sure that you set the following directive to use the public URI for your proxy server:

```
service_uri = "https://krill.example.org/"
```

Add the following section to your config and change the values to your machine's hostname.

```
[testbed]
# RRDP BASE URI
#
# Set the base RRDP uri for the testbed repository server.
#
# It is highly recommended to use a proxy in front of Krill.
#
# To expose the RRDP files you can actually proxy back to your testbed
# krill server (https://<yourkrill>/rrdp/), or you can expose the
# files as they are written to disk ($data_dir/repo/rrdp/)
#
# Set the following value to *your* public proxy hostname and path.
rrdp_base_uri = "https://krill.example.org/rrdp/"

# RSYNC BASE URI
#
# Set the base rsync URI (jail) for the testbed repository server.
#
# Make sure that you have an rsyncd running and a module which is
```

(continues on next page)

(continued from previous page)

```
# configured to expose the rsync repository files. By default these
# files would be saved to: $data/repo/rsync/current/
rsync_jail = "rsync://krill.example.org/repo/"

# TA AIA
#
# Set the rsync location for your testbed trust anchor certificate.
#
# You need to configure an rsync server to expose another module for the
# TA certificate. Don't use the module for the repository as its
# content will be overwritten.
#
# Manually retrieve the TA certificate from krill and copy it
# over - it won't change again. You can get it at:
# https://<yourkrill>/ta/ta.cer
ta_aia = "rsync://krill.example.org/ta/ta.cer"

# TA URI
#
# Like above, make the TA certificate available over HTTPS and
# specify the url here so that it may be included in the TAL.
ta_uri = "https://krill.example.org/ta/ta.cer"
```

17.5 Start / Enable krill

```
root@krill-test-09-rc:/etc# systemctl start krill
root@krill-test-09-rc:/etc# journalctl -u krill
-- Logs begin at Thu 2021-04-01 11:23:39 UTC, end at Thu 2021-04-01 19:55:19 UTC. --
Apr 01 19:55:17 krill-test-09-rc systemd[1]: Starting Krill...
Apr 01 19:55:17 krill-test-09-rc systemd[1]: Started Krill.
Apr 01 19:55:17 krill-test-09-rc krill[35246]: 2021-04-01 19:55:17 [INFO] Krill uses␣
↪configuration file: /etc/krill.conf
Apr 01 19:55:17 krill-test-09-rc krill[35246]: 2021-04-01 19:55:17 [INFO] Starting Krill␣
↪v0.8.2-bis
Apr 01 19:55:17 krill-test-09-rc krill[35246]: 2021-04-01 19:55:17 [INFO] Krill uses␣
↪service uri: https://localhost:3000/
Apr 01 19:55:17 krill-test-09-rc krill[35246]: 2021-04-01 19:55:17 [INFO] Enabling␣
↪TESTBED mode - ONLY USE THIS FOR TESTING AND TRAINING!
Apr 01 19:55:17 krill-test-09-rc krill[35246]: 2021-04-01 19:55:17 [INFO] Initializing␣
↪repository
Apr 01 19:55:18 krill-test-09-rc krill[35246]: 2021-04-01 19:55:18 [INFO] Creating␣
↪embedded Trust Anchor
Apr 01 19:55:18 krill-test-09-rc krill[35246]: 2021-04-01 19:55:18 [INFO] Sending␣
↪command to publisher '0', version: 1: id '0' version 'any' details 'Added publisher 'ta
↪''
Apr 01 19:55:18 krill-test-09-rc krill[35246]: 2021-04-01 19:55:18 [INFO] Sending␣
↪command to CA 'ta', version: 1: id 'ta' version 'any' details 'Update repo to server␣
↪at: https://localhost:3000/rfc8181/ta/'
Apr 01 19:55:18 krill-test-09-rc krill[35246]: 2021-04-01 19:55:18 [INFO] Sending␣
↪command to CA 'ta', version: 2: id 'ta' version 'any' details 'Turn into Trust Anchor'
```

(continues on next page)

(continued from previous page)

```
Apr 01 19:55:18 krill-test-09-rc krill[35246]: 2021-04-01 19:55:18 [INFO] Sending␣
↪command to CA 'ta', version: 3: id 'ta' version 'any' details 'Update received cert in␣
↪RC '0', with resources 'asn: 1 blocks, v4: >
Apr 01 19:55:18 krill-test-09-rc krill[35246]: 2021-04-01 19:55:18 [INFO] Creating␣
↪embedded Testbed CA
Apr 01 19:55:18 krill-test-09-rc krill[35246]: 2021-04-01 19:55:18 [INFO] Sending␣
↪command to publisher '0', version: 2: id '0' version 'any' details 'Added publisher
↪'testbed''
Apr 01 19:55:18 krill-test-09-rc krill[35246]: 2021-04-01 19:55:18 [INFO] Sending␣
↪command to CA 'testbed', version: 1: id 'testbed' version 'any' details 'Update repo␣
↪to server at: https://localhost:3000/rfc8181/>
Apr 01 19:55:18 krill-test-09-rc krill[35246]: 2021-04-01 19:55:18 [INFO] CA 'ta'␣
↪process add child request: handle 'testbed' resources 'asn: AS0-AS4294967295, v4: 0.0.
↪0.0/0, v6: ::/0'
Apr 01 19:55:18 krill-test-09-rc krill[35246]: 2021-04-01 19:55:18 [INFO] Sending␣
↪command to CA 'ta', version: 4: id 'ta' version 'any' details 'Add child 'testbed'␣
↪with RFC8183 key '98A7BBA3491C84000FADFF48AA53E>
Apr 01 19:55:18 krill-test-09-rc krill[35246]: 2021-04-01 19:55:18 [INFO] Sending␣
↪command to CA 'testbed', version: 2: id 'testbed' version 'any' details 'Add parent 'ta
↪' as 'RFC 6492 Parent''
Apr 01 19:55:18 krill-test-09-rc krill[35246]: 2021-04-01 19:55:18 [INFO] Upgraded Krill␣
↪to version: 0.8.2-bis
Apr 01 19:55:19 krill-test-09-rc krill[35246]: 2021-04-01 19:55:19 [INFO] Will re-sync␣
↪all CAs with their parents and repository after startup
```

If all is well you should be able to see the public testbed page now, that allows *ANY* unauthorised user to register a CA under your testbed and use it for publication, claiming any resources they please. This is by design. This meant for testing. Did we mention testing? You get the point.. don't use the TAL, also listed on that page, for anything BUT testing.

To get to the page you need to know the URI: https://krill.example.org/index.html#/testbed

CHAPTER

EIGHTEEN

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.

18.1 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.

18.2 Fetching and Running Krill

The **docker run** command will automatically fetch the Krill image for your CPU architecture 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.

Note: The CPU architectures supported by the Krill Docker image are shown on the *Docker Hub Krill page <https://hub.docker.com/r/nlnetlabs/krill/tags>* per Krill version (aka Docker "tag") in the *OS/ARCH* column.

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.

18.3 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.

18.4 Running the Krill CLI

18.4.1 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": []
}
```

18.4.2 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
```

18.5 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.

18.6 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.

18.6.1 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/
```

18.6.2 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
```

18.7 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
```

18.8 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/
```

18.9 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.

18.10 Running as a non-root user

The Krill Docker image supports running Krill as the non-root user “krill” (UID 1012, GID 1012) but for backward compatibility runs by default as user “root”.

One can specify that Krill should run as user “krill” like so:

```
docker run -u krill
```

Running as a different username, UID and/or GID requires building the Docker image yourself, e.g.:

```
cd path/to/krill/git/clone
docker build -t mykrill \
  --build-arg RUN_USER=myuser \
  --build-arg RUN_USER_UID=1234 \
  --build-arg RUN_USER_GID=5678 \
  .
```

Note: If running Krill inside the container as a non-root user and mounting the host filesystem or a Docker volume under the Krill data directory you must ensure that the Krill data directory and subdirectories are writable by Krill.

CHAPTER

NINETEEN

UPGRADING KRILL

19.1 Upgrade

Krill upgrades may sometimes require that existing data is migrated to a new format used by the new release. Krill will perform these migrations automatically in case you install a new version of Krill and restart it.

As the first step of this upgrade, any data that needs to be migrated is prepared under a new directory called `upgrade-data` under the `data_dir` you configured. If you used a package to install Krill then the latter would be `/var/lib/krill/data`.

If all is well then Krill will rename directories under the `data_dir` and archive your old data structures under directories called `arch_cas_version` and/or `arch_pubd_version`. You can safely remove these directories in order to save space later.

It is unlikely that a data migration should fail. We use automated and manual testing to make sure that these migrations work. But, of course even with testing things can still go wrong. If the preparation step fails then krill will exit with an error and refuse to start the new version.

If this happens, then you can abort the upgrade by re-installing your previous version of krill and starting that. And, please do let us know by making an issue.

19.2 Prepare Upgrade with krillup

If the fully automated upgrade process seems a bit too scary to you, then we recommend that you perform this step manually *before* upgrading krill itself.

Starting with Krill 0.9.5 we have introduced a new command line tool that can be used to help prepare for krill migrations.

If you built Krill using Cargo then you will find that a new binary called **krillup** is installed alongside with krill. But, if you are using the packages that we provide then you can install and upgrade this binary separately. For example on a Debian system:

```
sudo apt install krillup
```

If you install and/or upgrade **krillup** first, before upgrading Krill itself then you will be able to prepare and verify an upgrade while Krill is running. This is especially useful for large operations because some of these upgrades can take a while. By using the separate tool any downtime is limited. Furthermore, if the preparation should unexpectedly fail, then there will be no need to reinstall a previous version of Krill. You can simply abort the upgrade.

krillup only needs to be told where your config file lives. Here we use it to prepare an upgrade, where no actual data migration is needed. This is not an error, so it will just report that the upgrade does not require preparation:

```
$ krillup -c ./defaults/krill.conf
2022-02-18 16:51:26 [INFO] Prepare upgrade using configuration file: ./defaults/krill.
↪conf
2022-02-18 16:51:26 [INFO] Processing data from: ./data
2022-02-18 16:51:26 [INFO] Saving prepared data to: ./data/upgrade-data
2022-02-18 16:51:26 [INFO] No preparation is needed for the upgrade from 0.9.3-rc1 to 0.
↪9.5-rc1.
```

Important: Once migrated data cannot be rolled back to the format of a previous Krill version. So, while an upgrade can be aborted, it cannot be undone — other than by restoring data from the point before the upgrade and accepting that any changes since then will have been lost.

So, please read up on *important changes* to see if you would be affected by functionality or API changes before you upgrade.

19.3 Important Changes

19.3.1 v0.10.0

JSON Field Name Changes

When migrating support for RFC 6492, 8181 and 8183 into the base library *rpki-rs* (issue #765) we renamed some fields which are also used in the JSON structures of the Krill API:

pre-0.10.0	0.10.0	reason
v4	ipv4	More decscriptive
v6	ipv6	More decscriptive
base_uri	sia_base	Term used in RFC 8183
rpki_notify	rrdp_notification_uri	Term used in RFC 8183

We still accept the old names as aliases on input, but if you are parsing JSON responses yourself then you will need to update your code to accept the new names.

Parent Status Reporting

The parent status API and CLI text response now include the last known full **RFC 6492** “Resource Class List Response” content that your CA received.

The json structure of the parent statuses response changed from:

```
{
  "my_parent": {
  "last_exchange": {
    "timestamp": 1617881400,
    "uri": "https://localhost:3000/rfc8181/localname/",
    "result": "Success"
  },
  "next_exchange_before": 1617882000,
```

(continues on next page)

(continued from previous page)

```
  "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+tYwDQYJKoZIhvcNAQELBQAwggItMYICKTCCAiUGA1UEAxOCAhwzMDgy
↪q4q9w/gwwsAjoIP+cTm0FSmqhvqsc1GVI4DQ4mspjZ+O7esFqQywmcnU9MphnGq4EJwYKqT417fU8OQj/
↪WbiCfFhnTrVTiz/LdLdDB4+VaypGfDwPuHb8pavj2dysKiGjLcF8zdon7a/
↪xErHqOdetKlbY20TlvVvmLUeVVKfcnDkT8nsu2k+P+5BHBrb6oQoG4IhZ/w5n65m/
↪ozLsq7pfLrsLgFe2b4zTXhu8KdJ/W1vsshM73jkpUdkvKxif6+H4mBrlMnWg7Jo0bRuff/
↪C0dOAWdiPMXUs53Nw3+SBUjRxhXVWdbcHflkje58pcMkGKSBwIDAQABo4ICNjCCAjIwDwYDVR0TAQH/
↪BAUwAwEB/zAdBgNVHQ4EFgQUC6XBMrlIkcstOont4S8BrKS809wwHwYDVR0jBBgwFoAUS9B/
↪WEM89XrPSCIpOOkwBxZdNKQwDgYDVR0PAQH/
↪BAQDAgEGMFkGA1UdHwRSMFAwTqBMoEqGSHJzeW5jOi8vbG9jYWxob3N0L3JlcG8vdGEvMC80QkQwN0Y1ODQzM0NGNTdBQ0Y0ODIyM
↪wQOMAwwCgYIKwYBBQUHDgIwJwYIKwYBBQUHAQcBAf8EGDAWMAkEAgABMAMDAQAwCQQCAAIwAwMBADAhBggrBgEFBQcBCAEB/
↪wQSMBCgDjAMMAoCAQACBQD/////
↪MA0GCSqGSIb3DQEBCwUAA4IBAQA3rQv0h6x5zX6iGfUZsH0wFSbQQrZgWoql8PsHANokm+Kaxeq3waemrp1/
↪LCzdsMF4+74m6ijDmdbDbHlPyiQwpu3L1vZafj4eBPMdI7xFYgEgabddAGR60b272BgVIO6yND3B6UMeT56NzcCOtOcPtjlgucU3pu
↪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-----\
↪nMIIFYDCCBEigAwIBAgIUN5PzATTKVrjgual4CpJMaggW2EIwDQYJKoZIhvcNAQELBQAwMzExMC8GA1UEAxMoMEJBNUMxMzJCOTQ4
↪7OHN8TU6crIu1/
↪w1gkf6UCXFrv+poW9EJHnLonMa4ZFLSFsvQACIGUpxIuiQjaSYFltTbb+o2c9KWoKsX0kZqt5zOrgAP8cke8SFGHdqqenPInXKTgy
↪ngql4OV0bLkbb63J/26c8FZOThZAgMBAAGjggJqMIICZjAPBgNVHRMBAf8EBTADAQH/
↪MB0GA1UdDgQWBBQWsxyS6xFrxgAmxQlErUQgXdmsvTAfBgNVHSMEGDAWgBQLpcEyuUiRyy06ie3hLwGspLzT3DAOBgNVHQ8BAf8EB
↪BA4wDDAKBggrBgEFBQcOAjAsBggrBgEFBQcBBwEB/
↪wQdMBswCgQCAAEwBAMCAAowDQQCAAIwBwMFACABDbgwGgYIKwYBBQUHAQgBAf8ECzAJoAcwBQIDAP3oMA0GCSqGSIb3DQEBCwUAA4
↪Yja+sYhyg/pG1/
↪ZTvhOLIxGWap8JmqOnYa9XgX8uUlsV8LgJoEH3Gde3txcGtfLO99ugvbnKKGOcPxB8AX5hAhhfdiSnt3V06dEz3HUoTYdUKTV0bZr
↪K8349vN0QJUZGm3gAUsM5PlnAqbkM7VFIyu8g2Yp9g+M/
↪iwaHar8CqABKxLBThYgqrPLLv6CsZD3mjk5BkXVZh6R9dBcR7sPbSfGBWPWCv8SwLknyQDOvsWTho1Ga6AibjUQp\
↪n-----END CERTIFICATE-----\n"
        }
      ]
```

(continues on next page)

(continued from previous page)

```
    }
  }
}
```

To:

```
{
  "my_parent": {
    "last_exchange": {
      "timestamp": 1617881400,
      "uri": "https://localhost:3000/rfc8181/localname/",
      "result": "Success"
    },
    "last_success": 1617881400,
    "all_resources": {
      "asn": "AS65000",
      "ipv4": "10.0.0.0/8",
      "ipv6": "2001:db8::/32"
    },
    "classes": [
      {
        "class_name": "0",
        "resource_set": {
          "asn": "AS65000",
          "ipv4": "10.0.0.0/8",
          "ipv6": "2001:db8::/32"
        },
        "not_after": "2023-03-15T14:23:57Z",
        "issued_certs": [
          {
            "uri": "rsync://localhost/repo/testbed/0/
↪16B31C92EB116BC60026C50944AD44205DD9ACBD.cer",
            "req_limit": {},
            "cert": "MII..."
          }
        ],
        "signing_cert": {
          "url": "rsync://localhost/repo/ta/0/0BA5C132B94891CB2D3A89EDE12F01ACA4BCD3DC.
↪cer",
          "cert": "MII..."
        }
      }
    ]
  }
}
```

19.3.2 v0.9.3 to v0.9.5

There are no API changes or data migrations.

After upgrading the Publication Server (if you run one) will use `1` as the first RRDP serial number, instead of `0`. Furthermore, you will now be able to configure the timeout for a complete **RFC 6492** and **RFC 8181** client HTTP request-response round-trip to the parent or publisher, excluding the time required to establish the connection, using *post_protocol_msg_timeout_seconds*.

19.3.3 v0.9.0/1/2 to v0.9.3

There are no API changes, but users may want to be aware that the 'next update' time for manifests and CRLs has been changed from a fixed 24 hours (by default) to 24 hours and a random amount of extra time between 0 and 240 minutes (4 hours). This does not affect the validity of objects, but may lead to surprises if you are monitoring that republication would happen withing 17 hours after last publication (8 hours before objects would expire). This can now take up to 21 hours (using defaults).

Furthermore experimental ASPA support was added, but it's hidden in the CLI until the ASPA standards reach stability in the IETF. If you want to read more about the experimental ASPA support in Krill then have a look here:

https://krill.docs.nlnetlabs.nl/en/prototype-aspa-support/manage-aspas.html

19.3.4 v0.9.0/1 to v0.9.2

The Prometheus metrics have been updated. The metric `krill_cas_roas` has been renamed to `krill_cas_bgp_roas_total` for consistency. Please have a look at the updated *monitoring page* for more details.

19.3.5 v0.8.2 and below to v0.9.x

There are a number of API changes between v0.9.0 and previous versions. The main reasons for these changes are:

1. Krill no longer has the concept of embedded CA parent-child or repo-ca relations. If you have multiple CAs in a single Krill instance and/or a Publication Server, then Krill will now always use the official RFC protocol - even if both entities live in the same Krill instance.
2. We wanted to make the API consistent.

But most importantly: **We wanted to make the API stable so we can work towards Krill 1.0**

Here we will list all CLI commands and API calls that were changed between Krill 0.8.2 and this version. This list should be complete, so old CLI commands not listed here should not have changed.

In case you do find something that we overlooked please let us know!

krillc parents update

The **`update`** command has been removed and is now folded in to **`krillc parents add`**.

krillc parents add

If you add a parent which already exists for your CA, then this will act as an 'update' instead. I.e. the previously known **RFC 8183** Parent Response for the parent will be replaced.

The CLI command is unchanged:

```
$ krillc parents add --ca newca --parent testbed --response ./parent-response.xml
```

But there were changes to the API.

Adding a parent can be done by posting XML or JSON to on of the following paths:

```
/api/v1/cas/<ca>/parents
/api/v1/cas/<ca>/parents/<handle>
```

The `<handle>` is the LOCAL name that your CA will use for this parent. Regardless of how they like to call themselves. If it is omitted then it will be extracted from the XML `parent_handle`. If it is specified for a JSON POST but _differs_ from the `handle` in the JSON body, then an error is returned.

The server will verify in all cases that the parent can be reached. If there was no parent for the name a parent will be added, otherwise the parent contact details will be updated.

The JSON body has to include the local name by which the CA will refer to its parent, this is also the name shown to the user in the UI. The local name maps to the handle field in the JSON below. The second component is the contact. Krill used to support an embedded type, but this is no longer supported.

Instead of a JSON member under `contact` we now have `"type":  "rfc6492"` here. We still have this type because this allows for the notion of Trust Anchor - which we use in test setups - and it keeps the door open to future additions (eg if there ever is an RFC 6492 bis). The remainder of the structure is unchanged, and maps to the **RFC 8183** Parent Response XML, but then in JSON format. Note that the parent_handle is the handle that the parent wants the CA to use in messages sent to it - and it may be different from the local name stored in handle.

OLD JSON:

```
{
  "handle": "testbed",
  "contact": {
    "rfc6492": {
      "tag": null,
      "id_cert":
↪"MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyg5N0VEOUFCMUQ4Q0JBNzFBMTJEQjE2MTU4OTA3Njk4
↪sXvoG1C1pRPs/SBZ7gK6WvFlqdScZ6kbTVH51e+pLUV9Q7Uxqm4lSzWTmnjT/
↪CmRRXqmcPlcPcAm8rhUW6GrZQ2mllil4pkZ+JNGugSQUJJb1bGg5+Et/
↪YdIEDEO1stAIsNkfkAyELAeFULLhs0MuXpSKp/ZKu+IgMSt+Z/
↪7is+qFt4cgMuiZRuADw8hTDoMuZpoxIqXeh4Nf3bUU06MXGgrpabVzArs11UVyXDC4ZG4oOsYDTNgIL5VYaBjiHtw+s+FWHYI3iTz
↪BAUwAwEB/
↪zAdBgNVHQ4EFgQUl+2asdjLpxoS2xYViQdpirSgMdkwHwYDVR0jBBgwFoAUl+2asdjLpxoS2xYViQdpirSgMdkwDQYJKoZIhvcNAQ
↪bYp/
↪NnAk+u+Hw5AIwxuoGWgwyHXZh1xJFhwD35SqjMhxbo15J090+22zwAa8t6aqQAZhvACs2Jst1aHnnJEduQzGVZYLIYvGv5/
↪K0t0i0eE5hINhtAy0hFGwteXms8/qA/
↪mExsrjubC69SudPlMA3q8p2RmuwqmjSlwDjU1XrJ1j5wMCqeBoh8EnaMe+HVduQGHm0nHJbF3klz9mz3Tc6CILT4XA5mJq1g0LXyp
↪enulLYw=",
```

(continues on next page)

(continued from previous page)

```
      "parent_handle": "testbed",
      "child_handle": "newca",
      "service_uri": "https://testbed.krill.cloud/rfc6492/testbed"
    }
  }
}
```

Was changed to:

```
{
  "handle": "my_parent",
  "contact": {
    "type": "rfc6492",
    "tag": null,
    "id_cert":
↪"MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyhFOTBDMjE3MzRDMkMzNzBBOTFBODQ3NUNCNEYwRTc1
↪vVYxq1F1w2yQ/
↪VoTr1dvEHxJ+SDayMcFVktWCObiY8tcPhvWG+OdaX9ckDJhsOEEvdVEogwiGacNs7yXJPbqDBptJtbR8/
↪CauF9OqMqjkB/8xkGmBoY5OI/
↪V2832jkp7LPsbyET0RMQN7fgSpGbewvkaZVxGU3pHh5kT1nzPTXrwjxNMXgpunSEY7zR20vYCvsYYbxnSwFNbSMSL+Jgpa+HWPUc0
↪qlqlZK+tkppI2LkSBhTV5+n7cGA8ZsCAwEAAaNTMFEwDwYDVR0TAQH/BAUwAwEB/
↪zAdBgNVHQ4EFgQU6Qwhc0wsNwqRqEdctPDnXaTQ8L8wHwYDVR0jBBgwFoAU6Qwhc0wsNwqRqEdctPDnXaTQ8L8wDQYJKoZIhvcNAQ
↪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"
  }
}
```

krillc parents contact

The CLI command was unchanged:

```
$ krillc parents contact --parent testbed
```

And the default text response is still the RFC 8183 Parent Response XML for the parent. But, the JSON response body was changed, and now includes an explicit "type": "rfc6492":

OLD:

```
{
  "rfc6492": {
    "tag": null,
    "id_cert":
↪"MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyg5N0VEOUFCMUQ4Q0JBNzFBMTJEQjE2MTU4OTA3Njk4
↪sXvoG1C1pRPs/SBZ7gK6WvFlqdScZ6kbTVH51e+pLUV9Q7Uxqm4lSzWTmnjT/
↪CmRRXqmcPlcPcAm8rhUW6GrZQ2mllil4pkZ+JNGugSQUJJb1bGg5+Et/
↪YdIEDEO1stAIsNkfkAyELAeFULLhs0MuXpSKp/ZKu+IgMSt+Z/
↪7is+qFt4cgMuiZRuADw8hTDoMuZpoxIqXeh4Nf3bUU06MXGgrpabVzArs11UVyXDC4ZG4oOsYDTNgIL5VYaBjiHtw+s+FWHYI3iTz
↪BAUwAwEB/
↪zAdBgNVHQ4EFgQUl+2asdjLpxoS2xYViQdpirSgMdkwHwYDVR0jBBgwFoAUl+2asdjLpxoS2xYViQdpirSgMdkwDQYJKoZIhvcNAQ
↪bIp/
↪NnAk+u+Hw5AIwxuoGWgwyHXZh1xJFhwD35SqjMhxbo15J090+22zwAa8t6aqQAZhvACs2Jst1aHnnJEduQzGVZYLIYvGv5/
↪K0t0i0eE5hINhtAy0hFGwteXms8/qA/
↪mExsrjubC69SudPlMA3q8p2RmuwqmjSlwDjU1XrJ1j5wMCqeBoh8EnaMe+HVduQGHm0nHJbF3klz9mz3Tc6CILT4XA5mJq1g0LXyp
```

(continues on next page)

(continued from previous page)

```
    "parent_handle": "testbed",
    "child_handle": "newca",
    "service_uri": "https://testbed.krill.cloud/rfc6492/testbed"
  }
}
```

NEW:

```
{
  "type": "rfc6492",
  "tag": null,
  "id_cert":
↪"MIIDNDCCAhygAwIBAgIBATANBgkqhkiG9w0BAQsFADAzMTEwLwYDVQQDEyg5N0VEOUFCMUQ4Q0JBNzFBMTJEQjE2MTU4OTA3Njk4
↪sXvoG1C1pRPs/SBZ7gK6WvFlqdScZ6kbTVH51e+pLUV9Q7Uxqm4lSzWTmnjT/
↪CmRRXqmcPlcPcAm8rhUW6GrZQ2mllil4pkZ+JNGugSQUJJb1bGg5+Et/
↪YdIEDEO1stAIsNkfkAyELAeFULLhs0MuXpSKp/ZKu+IgMSt+Z/
↪7is+qFt4cgMuiZRuADw8hTDoMuZpoxIqXeh4Nf3bUU06MXGgrpabVzArs11UVyXDC4ZG4oOsYDTNgIL5VYaBjiHtw+s+FWHYI3iTz
↪BAUwAwEB/
↪zAdBgNVHQ4EFgQUl+2asdjLpxoS2xYViQdpirSgMdkwHwYDVR0jBBgwFoAUl+2asdjLpxoS2xYViQdpirSgMdkwDQYJKoZIhvcNAQ
↪bYp/
↪NnAk+u+Hw5AIwxuoGWgwyHXZh1xJFhwD35SqjMhxbo15J090+22zwAa8t6aqQAZhvACs2Jst1aHnnJEduQzGVZYLIYvGv5/
↪K0t0i0eE5hINhtAy0hFGwteXms8/qA/
↪mExsrjubC69SudPlMA3q8p2RmuwqmjSlwDjU1XrJ1j5wMCqeBoh8EnaMe+HVduQGHm0nHJbF3klz9mz3Tc6CILT4XA5mJq1g0LXyp
↪enulLYw=",
  "parent_handle": "testbed",
  "child_handle": "newca",
  "service_uri": "https://testbed.krill.cloud/rfc6492/testbed"
}
```

krillc repo request

The CLI is unchanged, but the endpoints for getting the **RFC 8183** Publisher Request XML and JSON have moved from `repo`, and are now under `id`:

```
/api/v1/cas/<name>/repo/request.xml  -> /api/v1/cas/<name>/id/publisher_request.xml
/api/v1/cas/<name>/repo/request.json -> /api/v1/cas/<name>/id/publisher_request.json
```

krillc repo update

This command has been renamed to **`krillc repo configure`**:

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

The API has also changed. The path is unchanged, but the following to add an “embedded” repository is **no longer supported**:

```
{
  "tag": "string",
  "id_cert": "string",
  "child_handle": "string"
}
```

The API end-point will accept either plain **RFC 8183** Repository Response XML, or a JSON equivalent. In comparison to previous versions of Krill *rfc8181* was renamed to *repository_response*:

```
{
  "repository_response": {
    "tag": null,
    "publisher_handle": "publisher",
    "id_cert": "MIID..6g==",
    "service_uri": "https://repo.example.com/rfc8181/publisher/",
    "repo_info": {
      "base_uri": "rsync://localhost/repo/ca/",
      "rpki_notify": "https://localhost:3000/rrdp/notification.xml"
    }
  }
}
```

krillc repo show

The CLI command and API path are unchanged, but `rfc8181` was renamed to `repository_response` in the JSON response.

krillc children add

The CLI is unchanged, but because 'embedded' children are no longer supported we were able to simplify the JSON from:

```
{
  "handle": "ca",
  "resources": {
    "asn": "AS1",
    "v4": "10.0.0.0/8",
    "v6": "::"
  },
  "auth": {
    "rfc8183": {
      "tag": null,
      "child_handle": "ca",
      "id_cert": "<base64>"
    }
  }
}
```

To this:

```
{
  "handle": "ca",
  "resources": {
    "asn": "AS1",
    "v4": "10.0.0.0/8",
    "v6": "::"
  },
```

(continues on next page)

(continued from previous page)

```
  "id_cert": "<base64>"
}
```

krillc history and krillc action

The API and JSON are unchanged, but these commands have now been renamed to `krillc history commands` and `krillc history details`.

CHAPTER

TWENTY

FAILURE AND RECOVERY SCENARIOS

20.1 CA Temporarily Unavailable

Issue
The Krill instance for your CA is temporarily unavailable

Consequences
You cannot change ROAs
You cannot change delegations to child CAs
Krill will not update its repository

If the outage is short, e.g. because you are performing a planned upgrade, then this will have little or no impact. The RPKI objects which were published by your CA will remain unchanged, so your ROAs will still be found and considered valid by RPKI validators.

Note that if you are using automation to keep your ROAs in sync with your routing configuration, you should take care to ensure that your set up can deal with a short outage of your CA and tries to re-apply any possible ROA changes in case your CA was unavailable.

If the outage takses longer than 8 hours (using default settings), then your *CA publication point will become expired* and the impact will be bigger.

20.2 Parent Temporarily Unavailable

Issue
The parent of your CA is temporarily unavailable

Consequences
You will not receive changes to your resource certificate
You cannot perform a key roll

As long as the parent repository is not expired (*see below*) this has minimal impact on your CA. Krill CAs will check for updated resource and validity time entitlements every 10 minutes, and they will just keep trying.

The status is shown in the UI, but you can also use the following CLI command:

```
krillc parents statuses
```

If you parent CA is unavailable due to an outage, or an ongoing upgrade, then there is not much that you can do. You may want to talk to them, but a responsible parent should monitor their own operations, so they are expected to become available again without the need for you to take action. But note that you should verify whether the issue on your side. E.g. there may be a network issue, or firewall rule preventing your CA for contacting the parent CA.

20.3 Publication Point Expired

Issue
The manifest or CRL of your CA expired

Consequences
Your published objects are no longer valid
Your routes become “not found” in most cases

When your manifest or CRL become expired your RPKI objects will become invalid. This problem can occur if your CA is down, or if your CA cannot publish updated objects at its publication server, for a prolonged period of time.

Krill uses a default validity time of 24 hours for manifests and CRLs, and replaces them 8 hours before they would expire. This means that from the moment of the outage you have 8-24 hours to prevent that your objects will be invalidated.

It is possible to change these defaults if you want to have more time to deal with potential issues. However, we recommend that you avoid using long validity times because in theory they could make you vulnerable to replay attacks where a malicious actor feeds old objects to RPKI validators. This attack is not trivial, but it’s not impossible either.

A reasonable compromise could be to use a validity time of 36 hours, and have Krill reissue manifests and CRLs 24 hours before they would expire. You can achieve this by adding the following directives to your configuration file:

```
timing_publish_next_hours = 36
timing_publish_hours_before_next = 24
```

When your objects, most importantly ROAs, become invalid your routes will usually become “not found”, rather than “invalid”. Meaning that your routes will no longer benefit from Route Origin Validation, but they will still be accepted.

For a route to become RPKI “invalid” it would need to be covered by one or more valid ROA objects which include this prefix, none of which allow the possibly more specific prefix and ASN.

In the set up we see today this is unlikely to happen as most Krill CAs will operate directly under a parent RIR or NIR, and will not delegate prefixes to children. RIRs and NIRs do not issue ROAs for delegated prefixes, so in case your publication point would be rejected there would be no remaining valid ROA objects for your announcement. The result is that they then get an RPKI validity state “not found”.

However, in complicated setups your routes can become invalid. For example if your organisation operates a main CA under an RIR, and it publishes ROAs, while delegating some resources covered by those ROAs to you (e.g. a business unit or customer), and your publication point is expired while your parent’s publication point is still current.. then your routes can become “invalid”.

If it can be helped it would therefore be advisable that your parent does not delegate resources for which they also manage ROAs.

20.4 Parent Publication Point Expired

Issue
The manifest or CRL of your parent CA expired

Consequences
Your published objects are no longer valid
Your routes become “not found” in most cases

If your parent CA’s publication point is expired, then its objects will become invalid. This includes the certificate for the delegation done to you, and therefore your objects will also no longer be considered valid by RPKI validators.

As described *above* this will typically mean that your routes end up with the RPKI validity state “not found”. The chances of them becoming “invalid” are actually somewhat lower still becuase any possible ROAs issued by your parent or siblings (other children under the same parent) covering your resources would also be invalid.

INDEX