RPKI Tools Manual

http://rpki.net

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

RPKI Tools Manual

This collection of tools implements both the production (CA) and relying party (RP) sides of an RPKI environment.

The Subversion repository for the entire project is available for (read-only) anonymous access at http://subvert-rpki.hactrn.net/.

If you just want to browse the code you might find the Trac source code browser interface more convenient.

Download and Install

Full source code is available, as are binary packages for a few platforms.

See the installation instructions for how to download the code and install it once you've downloaded it.

Relying Party Tools

If you operate routers and want to use RPKI data to help secure them, you should look at the relying party tools.

CA Tools

If you control RPKI resources and need an engine let you request certificates, issue ROAs, or issue certificates to other entities, you should look at the CA tools.

Thanks

This work was funded from 2006 through 2008 by ARIN, in collaboration with the other Regional Internet Registries. Current work is funded by DHS.

Download and Installation

There are a few different ways to install the RPKI code, depending on what the platform on which you're trying to install.

- On Ubuntu 12.04 LTS ("Precise Pangolin") or Debian 7 ("Wheezy"), you can use Debian binary packages.

  At present we only generate binary packages for Precise Pangolin and Wheezy. This may change in the future.

- On FreeBSD, you can use FreeBSD ports.

- On all other platforms, or on the above platforms if the pre-packaged versions don't suit your needs, you will have to install from source code.

Once you've finished installing the code, you will need to configure it. Since CAs are generally also relying parties (if only so that they can check the results of their own actions), you will generally want to start by configuring the relying party tools, then configure the CA tools if you're planning to use them.

Installation Using Debian Packages on Debian and Ubuntu Systems

Precompiled binary packages for Ubuntu 12.04 LTS ("Precise Pangolin") and Debian 7 ("Wheezy") are available from download.rpki.net using the Debian Advanced Package Tools (APT). To use these, you need to configure APT on your machine to know about our APT repository, but once you've done this you should be able to install and update these packages like any other precompiled package.

Initial APT Setup

You should only need to perform these steps once for any particular machine.

- Add the GPG public key for this repository (optional, but APT will whine unless you do this):
  ```
  wget -q -O - http://download.rpki.net/APT/apt-gpg-key.asc | sudo apt-key add -
  ```
- Configure APT to use this repository (for Ubuntu systems):
  ```
  sudo wget -q -O /etc/apt/sources.list.d/rpki.list http://download.rpki.net/APT/rpki.ubun
  ```
- Configure APT to use this repository (for Debian systems):
  ```
  sudo wget -q -O /etc/apt/sources.list.d/rpki.list http://download.rpki.net/APT/rpki.debi
  ```

Installation Using APT Tools

These instructions assume that you're using apt-get. Other APT tools such as aptitude should also work.

- Update available packages:
  ```
  sudo apt-get update
  ```
- Install the software:
  ```
  sudo apt-get install rpki-rp rpki-ca
  ```
- Customize the default `rpki.conf` for your environment as necessary. In particular, you want to change `handle` and `rpkid_server_host`. There are obsessively detailed instructions.
  ```
  sudo emacs /etc/rpki.conf
  ```

  Again, you want to change `handle` and `rpkid_server_host` at the minimum.
- If you changed anything in `rpki.conf`, you should restart the RPKI CA service:
  ```
  sudo initctl restart rpki-ca
  ```

Upgrading

Once you've performed the steps above you should be able to upgrade to newer version of the code using the normal APT upgrade process, eg:

```
sudo apt-get update
sudo apt-get upgrade
```

Or, if you only want to update the RPKI tools:

```
sudo apt-get update
sudo apt-get upgrade
```

Installation Using FreeBSD Ports

Port skeletons are available for FreeBSD from download.rpki.net. To use these, you need to download the port skeletons then run them using your favorite FreeBSD port installation tool.

Manual Download

To download the port skeletons manually and install from them, do something like this:

```
for port in rpki-rp rpki-ca
do
  fetch http://download.rpki.net/FreeBSD_Packages/${port}-port.tgz
  tar xf ${port}-port.tgz
  cd ${port}
  make install
  cd ..
  rm -rf ${port}
done
```

After performing initial installation, you should customize the default `rpki.conf` for your environment as necessary. In particular, you want to change `handle` and `rpkid_server_host`. There are obsessively detailed instructions.

```
emacs /usr/local/etc/rpki.conf
```

Again, you want to change `handle` and `rpkid_server_host` at the minimum.

To upgrade, you can perform almost the same steps, but the FreeBSD ports system, which doesn't really know about upgrades, will require you to use the `deinstall` and `reinstall` operations instead of plain `install`:

```
for port in rpki-rp rpki-ca
do
  fetch http://download.rpki.net/FreeBSD_Packages/${port}-port.tgz
  tar xf ${port}-port.tgz
  cd ${port}
  make deinstall
  make reinstall
  cd ..
  rm -rf ${port}
done
```

After an upgrade, you may want to check the newly-installed `/usr/local/etc/rpki.conf.sample` against your existing `/usr/local/etc/rpki.conf` in case any important options have changed. We generally try to keep options stable between versions, and provide sane defaults where we can, but if you've done a lot of customization to your `rpki.conf` you will want to keep track of this.

Automated Download and Install with portmaster

There's a script you can use to automate the download steps above and perform the updates using portmaster. First, download the script:

```
fetch http://download.rpki.net/FreeBSD_Packages/rpki-portmaster.sh
```

Then, to install or upgrade, just execute the script:

```
sh rpki-portmaster.sh
```

As with manual download (above) you should customize `rpki.conf` after initial installation.

Automated Download and Install with portupgrade

There's a script you can use to automate the download steps above and perform the updates using portupgrade. First, download the script:

```
fetch http://download.rpki.net/FreeBSD_Packages/rpki-portupgrade.sh
```

Next, you will need to add information about the RPKI ports to two variables in `/usr/local/etc/pkgtools.conf` before portupgrade will know how to deal with these ports:

```
EXTRA_CATEGORIES = [
    'rpki',
]

ALT_INDEX = [
    ENV['PORTSDIR'] + '/INDEX.rpki',
]
```

Once you have completed these steps, you can just execute the script to install or upgrade the RPKI code:

```
sh rpki-portupgrade.sh
```

As with manual download (above) you should customize `rpki.conf` after initial installation.

Installing From Source Code

At present, the entire RPKI tools collection is a single source tree with a shared autoconf configuration. This may change in the future, but for now, this means that the build process is essentially the same regardless of which tools one wants to use. Some of the tools have dependencies on external packages, although we've tried to keep this to a minimum.

Most of the tools require an RFC-3779-aware version of the OpenSSL libraries. If necessary, the build process will generate its own private copy of the OpenSSL libraries for this purpose.

Other than OpenSSL, most of the relying party tools are fairly self-contained. The CA tools have a few additional dependencies, described below.

Note that initial development of this code has been on FreeBSD, so installation will probably be easiest on FreeBSD. We do, however, test on other platforms, such as Fedora, Ubuntu, Debian, and MacOSX.

Downloading the Source Code

The recommended way to obtain the source code is via subversion. To download, do:

```
$ svn checkout http://subvert-rpki.hactrn.net/trunk/
```

Code snapshots are also available from http://download.rpki.net/ as xz-compressed tarballs.

Prerequisites

Before attempting to build the tools from source, you will need to install any missing prerequisites.

Some of the relying party tools and most of the CA tools are written in Python. Note that the Python code requires Python version 2.6 or 2.7.

On some platforms (particularly MacOSX) the simplest way to install some of the Python packages may be the "easy_install" or "pip" tools that comes with Python.

Packages you will need:

- You will need a C compiler. gcc is fine, others such as Clang should also work.
- http://www.python.org/, the Python interpreter, libraries, and sources. On some platforms the Python sources (in particular, the header files and libraries needed when building Python extensions) are in a separate "development" package, on other platforms they are all part of a single package. If you get compilation errors trying to build the POW code later in the build process and the error message says something about the file "Python.h" being missing, this is almost certainly your problem.
    - FreeBSD:
        - /usr/ports/lang/python27 (python)
    - Debian & Ubuntu:
        - python
        - python-dev
        - python-setuptools
- http://codespeak.net/lxml/, a Pythonic interface to the Gnome LibXML2 libraries. lxml in turn requires the LibXML2 C libraries; on some platforms, some of the LibXML2 utilities are packaged separately and may not be pulled in as dependencies.
    - FreeBSD: /usr/ports/devel/py-lxml (py27-lxml)

- Fedora: python-lxml.i386
- Debian & Ubuntu:
  - python-lxml
  - libxml2-utils

- http://www.mysql.com/, MySQL client and server. How these are packaged varies by platform, on some platforms the client and server are separate packages, on others they might be a single monolithic package, or installing the server might automatically install the client as a dependency. On MacOSX you might be best off installing a binary package for MySQL. The RPKI CA tools have been tested with MySQL 5.0, 5.1, and 5.5; they will probably work with any other reasonably recent version.
  - FreeBSD:
    - /usr/ports/databases/mysql55-server (mysql55-server)
    - /usr/ports/databases/mysql55-client (mysql55-client)
  - Debian & Ubuntu:
    - mysql-client
    - mysql-server

- http://sourceforge.net/projects/mysql-python/, the Python "db" interface to MySQL.
  - FreeBSD: /usr/ports/databases/py-MySQLdb (py27-MySQLdb)
  - Fedora: MySQL-python.i386
  - Debian & Ubuntu: python-mysqldb

- http://www.djangoproject.com/, the Django web user interface toolkit. The GUI interface to the CA tools requires this. Django 1.4 is required.
  - FreeBSD: /usr/ports/www/py-django (py27-django)
  - Debian: python-django
  - Ubuntu: **Do not use the python-django package (Django 1.3.1) in 12.04 LTS, as it is known not to work.**
    Instead, install a recent version using easy_install or pip:
    ```
    $ sudo pip install django==1.4.5
    ```

- http://vobject.skyhouseconsulting.com/, a Python library for parsing VCards. The GUI uses this to parse the payload of RPKI Ghostbuster objects.
  - FreeBSD: /usr/ports/deskutils/py-vobject (py27-vobject)
  - Debian & Ubuntu: python-vobject

- Several programs (more as time goes on) use the Python argparse module. This module is part of the Python standard library as of Python 2.7, but you may need to install it separately if you're stuck with Python 2.6. Don't do this unless you must. In cases where this is necessary, you'll probably need to use pip:
  ```
  $ python -c 'import argparse' 2>/dev/null || sudo pip install argparse
  ```

- http://pyyaml.org/. Several of the test programs use PyYAML to parse a YAML description of a simulated allocation hierarchy to test.
  - FreeBSD: /usr/ports/devel/py-yaml (py27-yaml)
  - Debian & Ubuntu: python-yaml

- http://xmlsoft.org/XSLT/. Some of the test code uses xsltproc, from the Gnome LibXSLT package.
  - FreeBSD: /usr/ports/textproc/libxslt (libxslt)
  - Debian & Ubuntu: xsltproc

- http://www.rrdtool.org/. The relying party tools use this to generate graphics which you may find useful in monitoring the behavior of your validator. The rest of the software will work fine without rrdtool, you just won't be able to generate those graphics.
  - FreeBSD: /usr/ports/databases/rrdtool (rrdtool)

- Debian & Ubuntu: rrdtool

- http://www.freshports.org/www/mod_wsgi3/ If you intend to run the GUI with wsgi, its default configuration, you will need to install mod_wsgi v3
  - FreeBSD: /usr/ports/www/mod_wsgi3 (app22-mod_wsgi)
  - Debian & Ubuntu: libapache2-mod-wsgi

- http://south.aeracode.org/ Django South 0.7.6 or later. This tool is used to ease the pain of changes to the web portal database schema.
  - FreeBSD: /usr/ports/databases/py-south (py27-south)
  - Debian: python-django-south
  - Ubuntu: **Do not use the python-django-south 0.7.3 package in 12.04 LTS, as it is known not to work.**
    Instead, install a recent version using easy_install or pip:
    `pip install South>=0.7.6`

Configure and build

Once you have the prerequesite packages installed, you should be able to build the toolkit. cd to the top-level directory in the distribution, run the configure script, then run "make":

```
$ cd $top
$ ./configure
$ make
```

This should automatically build everything, in the right order, including building a private copy of the OpenSSL libraries with the right options if necessary and linking the POW module against either the system OpenSSL libraries or the private OpenSSL libraries, as appopriate.

In theory, `./configure` will complain about any required packages which might be missing.

If you don't intend to run any of the CA tools, you can simplify the build and installation process by telling `./configure` that you only want to build the relying party tools:

```
$ cd $top
$ ./configure --disable-ca-tools
$ make
```

Testing the build

Assuming the build stage completed without obvious errors, the next step is to run some basic regression tests.

Some of the tests for the CA tools require MySQL databases to store their data. To set up all the databases that the tests will need, run the SQL commands in rpkid/tests/smoketest.setup.sql. The MySQL command line client is usually the easiest way to do this, eg:

```
$ cd $top/rpkid
$ mysql -u root -p <tests/smoketest.setup.sql
```

To run the tests, run "make test":

```
$ cd $top
$ make test
```

To run a more extensive set of tests on the CA tool, run "make all-tests" in the rpkid/ directory:

```
$ cd $top/rpkid
```

```
$ make all-tests
```

If nothing explodes, your installation is probably ok. Any Python backtraces in the output indicate a problem.

Installing

Assuming the build and test phases went well, you should be ready to install the code. The `./configure` script attempts to figure out the "obvious" places to install the various programs for your platform: binaries will be installed in `/usr/local/bin` or `/usr/local/sbin`, Python modules will be installed using the standard Python distutils and should end up wherever your system puts locally-installed Python libraries, and so forth.

The RPKI validator, rcynic, is a special case, because the install scripts may attempt to build a chroot jail and install rcynic in that environment. This is straightforward in FreeBSD, somewhat more complicated on other systems, primarily due to hidden dependencies on dynamic libraries.

To install the code, become root (su, sudo, whatever), then run "make install":

```
$ cd $top
$ sudo make install
```

Tools you should not need to install

There's a last set of tools that only developers should need, as they're only used when modifying schemas or regenerating the documentation. These tools are listed here for completeness.

- http://www.doxygen.org/. Doxygen in turn pulls in several other tools, notably Graphviz, pdfLaTeX, and Ghostscript.
    - FreeBSD: /usr/ports/devel/doxygen
    - Debian & Ubuntu: doxygen

- http://www.mbayer.de/html2text/. The documentation build process uses xsltproc and html2text to dump flat text versions of a few critical documentation pages.
    - FreeBSD: /usr/ports/textproc/html2text

- http://www.thaiopensource.com/relaxng/trang.html. Trang is used to convert RelaxNG schemas from the human-readable "compact" form to the XML form that LibXML2 understands. Trang in turn requires Java.
    - FreeBSD: /usr/ports/textproc/trang

- http://search.cpan.org/dist/SQL-Translator/. SQL-Translator, also known as "SQL Fairy", includes code to parse an SQL schema and dump a description of it as Graphviz input. SQL Fairy in turn requires Perl.
    - FreeBSD: /usr/ports/databases/p5-SQL-Translator

- http://www.easysw.com/htmldoc/. The documentation build process uses htmldoc to generate PDF from the project's Trac wiki.
    - FreeBSD: /usr/ports/textproc/htmldoc

Next steps

Once you've finished installing the code, you will need to configure it. Since CAs are generally also relying parties (if only so that they can check the results of their own actions), you will generally want to start by configuring the relying party tools, then configure the CA tools if you're planning to use them.

RPKI Relying Party Tools

These tools implements the "relying party" role of the RPKI system, that is, the entity which retrieves RPKI objects from repositories, validates them, and uses the result of that validation process as input to other processes, such as BGP security.

See the CA tools for programs to help you generate RPKI objects, if you need to do that.

The RP main tools are rcynic and rtr-origin, each of which is discussed below.

The installation process sets up everything you need for a basic RPKI validation installation. You will, however, need to think at least briefly about which RPKI trust anchors you are using, and may need to change these from the defaults.

The installation process sets up a cron job running running rcynic-cron as user "`rcynic`" once per hour at a randomly-selected minute.

rcynic

rcynic is the primary validation tool. It does the actual work of RPKI validation: checking syntax, signatures, expiration times, and conformance to the profiles for RPKI objects. The other relying party programs take rcynic's output as their input.

The installation process sets up a basic rcynic configuration. See the rcynic documentation if you need to know more.

See the discussion of trust anchors.

rtr-origin

rtr-origin is an implementation of the rpki-rtr protocol, using rcynic's output as its data source. rtr-origin includes the rpki-rtr server, a test client, and a utiltity for examining the content of the database rtr-origin generates from the data supplied by rcynic.

See the rtr-origin documentation for further details.

rcynic-cron

rcynic-cron is a small script to run the most common set of relying party tools under cron. See the discussion of running relying party tools under cron for further details.

Selecting trust anchors

As in any PKI system, validation in the RPKI system requires a set of "trust anchors" to use as a starting point when checking certificate chains. By definition, trust anchors can only be selected by you, the relying party.

As with most other PKI software, we supply a default set of trust anchors which you are welcome to use if they suit your needs. These are installed as part of the normal installation process, so if you don't do anything, you'll get these. You can, however, override this if you need something different; see the rcynic documentation for details.

Remember: It's only a trust anchor if **you** trust it. We can't make that decision for you.

Also note that, at least for now, ARIN's trust anchor locator is absent from the default set of trust anchors. This is not an accident: it's the direct result of a deliberate policy decision by ARIN to require anyone using their trust anchor to jump through legal hoops. If you have a problem with this, complain to ARIN. If and when ARIN changes this policy, we will be happy to include their trust anchor locator along with those of the other RIRs.

rcynic RPKI validator

rcynic is the core RPKI relying party tool, and is the code which performs the actual RPKI validation. Most of the other relying party tools just use rcynic's output.

The name is short for "cynical rsync", because rcynic's task involves an interleaved process of rsync retrieval and RPKI validation.

This code was developed on FreeBSD, and has been tested most heavily on FreeBSD versions 6-STABLE through 8-STABLE. It is also known to work on Ubuntu (12.04 LTS), Debian (Wheezy) and Mac OS X (Snow Leopard). In theory it should run on any reasonably POSIX-like system. As far as we know, rcynic does not use any seriously non-portable features, but neither have we done a POSIX reference manual lookup for every function call. Please report any portability problems.

Don't panic

rcynic has a lot of options, but it attempts to choose reasonable defaults where possible. The installation process will create a basic working rcynic configuration for you and arrange for this to run hourly under cron. If all goes well, this should "just work".

rcynic has the ability to do all of its work in a chroot jail. This used to be the default configuration, but integrating this properly with platform-specific packaging systems (FreeBSD ports, apt-get on Ubuntu and Debian, etc) proved impractical. You can still get this behavior if you need it, by installing from source and using the --enable-rcynic-jail option to ./configure.

The default configuration set up by make install and the various packaging systems will run rcynic under cron using the rcynic-cron wrapper script. See the instructions for setting up your own cron jobs if you need something more complicated; also see the instructions for setting up hierarchical rsync if you need to build a complex topology of rcynic validators.

Overview

rcynic depends heavily on the OpenSSL libcrypto library, and requires a reasonably current version of OpenSSL with both RFC 3779 and CMS support.

rcynic expects all certificates, CRLs, and CMS objects to be in DER format. rcynic stores its database using filenames derived from the RPKI rsync URIs at which the data are published.

All configuration is via an OpenSSL-style configuration file, except for selection of the name of the configuration file itself. A few other parameters can also be set from the command line. The default name for the configuration is "rcynic.conf"; you can override this with the -c option on the command line. The configuration file uses OpenSSL's configuration file syntax, and you can set OpenSSL library configuration paramaters (eg, "engine" settings) in the config file as well. rcynic's own configuration parameters are in a section called "[rcynic]".

Most configuration parameters are optional and have defaults which should do something reasonable if you are running rcynic in a test directory. If you're running rcynic as a system program, perhaps under cron via the rcynic-cron script, you'll want to set additional parameters to tell rcynic where to find its data and where to write its output (the installation process sets these parameters for you). The configuration file itself, however, is not optional. In order for rcynic to do anything useful, your configuration file **MUST** at minimum tell rcynic where to find one or more RPKI trust anchors or trust anchor locators (TALs).

Trust anchors

- To specify a trust anchor, use the `trust-anchor` directive to name the local file containing the trust anchor.
- To specify a trust anchor locator (TAL), use the `trust-anchor-locator` directive to name a local file containing the trust anchor locator.
- To specify a directory containing trust anchors or trust anchor locators, use the `trust-anchor-directory` directive to name the directory. Files in the specified directory with names ending in `".cer"` will be processed as trust anchors, while files with names ending in `".tal"` will be processed as trust anchor locators.

You may use a combination of these methods if necessary.

Trust anchors are represented as DER-formatted X.509 self-signed certificate objects, but in practice trust anchor locators are more common, as they reduce the amount of locally configured data to the bare minimum and allow the trust anchor itself to be updated without requiring reconfiguration of validators like rcynic. A trust anchor locator is a file in the format specified in RFC-6490, consisting of the rsync URI of the trust anchor followed by the Base64 encoding of the trust anchor's public key.

Strictly speaking, trust anchors do not need to be self-signed, but many programs (including OpenSSL) assume that trust anchors will be self-signed. See the `allow-non-self-signed-trust-anchor` configuration option if you need to use a non-self-signed trust anchor, but be warned that the results, while technically correct, may not be useful.

See the `make-tal.sh` script in this directory if you need to generate your own TAL file for a trust anchor.

As of this writing, there still is no single global trust anchor for the RPKI system, so you have to provide separate trust anchors for each Regional Internet Registry (RIR) which is publishing RPKI data. The installation process installs the ones it knows about.

Example of a minimal config file specifying nothing but trust anchor locators:

```
[rcynic]

trust-anchor-locator.0 = trust-anchors/apnic.tal
trust-anchor-locator.1 = trust-anchors/ripe.tal
trust-anchor-locator.2 = trust-anchors/afrinic.tal
trust-anchor-locator.3 = trust-anchors/lacnic.tal
```

Eventually, this should all be collapsed into a single trust anchor, so that relying parties don't need to sort this out on their own, at which point the above configuration could become something like:

```
[rcynic]

trust-anchor-locator = trust-anchors/iana.tal
```

Output directories

By default, `rcynic` uses two writable directory trees:

`unauthenticated`::

> Raw data fetched via `rsync`. In order to take full advantage of `rsync`'s optimized transfers, you should preserve and reuse this directory across `rcynic` runs, so that `rcynic` need not

re-fetch data that have not changed.

`authenticated`::

Data which `rcynic` has checked. This is the real output of the validation process.

`authenticated` is really a symbolic link to a directory with a name of the form "`authenticated.`*<timestamp>*", where *<timestamp>* is an ISO 8601 timestamp like `2001-04-01T01:23:45Z`. `rcynic` creates a new timestamped directory every time it runs, and moves the symbolic link as an atomic operation when the validation process completes. The intent is that `authenticated` always points to the most recent usable validation results, so that programs which use `rcynic`'s output don't need to worry about whether an `rcynic` run is in progress.

`rcynic` installs trust anchors specified via the `trust-anchor-locator` directive in the `unauthenticated` tree just like any other fetched object, and copies them into the `authenticated` trees just like any other object once they pass `rcynic`'s checks.

`rcynic` copies trust anchors specified via the `trust-anchor` directive into the top level directory of the `authenticated` tree with filenames of the form *<xxxxxxxx>*`.`*<n>*`.cer`, where *<xxxxxxxx>* and *<n>* are the OpenSSL object name hash and index within the resulting virtual hash bucket, respectively. These are the same values that OpenSSL's `c_hash` Perl script would produce. The reason for this naming scheme is that these trust anchors, by definition, are not fetched automatically, and thus do not really have publication URIs in the sense that every other object in these trees do. So `rcynic` uses a naming scheme which insures:

- that each trust anchor has a unique name within the output tree and
- that trust anchors cannot be confused with certificates: trust anchors always go in the top level of the tree, data fetched via rsync always go in subdirectories.

Trust anchors and trust anchor locators taken from the directory named by the `trust-anchor-directory` directive will follow the same naming scheme trust anchors and trust anchor locators specified via the `trust-anchor` and `trust-anchor-locator` directives, respectively.

Usage and configuration

Logging levels

`rcynic` has its own system of logging levels, similar to what `syslog()` uses, but customized to the specific task `rcynic` performs.

`log_sys_err`	Error from operating system or library
`log_usage_err`	Bad usage (local configuration error)
`log_data_err`	Bad data (broken certificates or CRLs)
`log_telemetry`	Normal chatter about rcynic's progress
`log_verbose`	Extra verbose chatter
`log_debug`	Only useful when debugging

Command line options

`-c` *configfile*	Path to configuration file (default: `rcynic.conf`)
`-l` *loglevel*	Logging level (default: `log_data_err`)
`-s`	Log via syslog
`-e`	Log via stderr when also using syslog

`-j`	Start-up jitter interval (see below; default: `600`)
`-V`	Print rcynic's version to standard output and exit
`-x`	Path to XML "summary" file (see below; no default)

Configuration file reference

`rcynic` uses the OpenSSL `libcrypto` configuration file mechanism. All `libcrypto` configuration options (eg, for engine support) are available. All `rcynic`-specific options are in the "`[rcynic]`" section. You **MUST** have a configuration file in order for `rcynic` to do anything useful, as the configuration file is the only way to list your trust anchors.

authenticated

Path to output directory (where `rcynic` should place objects it has been able to validate).

Default: `rcynic-data/authenticated`

unauthenticated

Path to directory where `rcynic` should store unauthenticatd data retrieved via `rsync`. Unless something goes horribly wrong, you want `rcynic` to preserve and reuse this directory across runs to minimize the network traffic necessary to bring your repository mirror up to date.

Default: `rcynic-data/unauthenticated`

rsync-timeout

How long (in seconds) to let `rsync` run before terminating the `rsync` process, or zero for no timeout. You want this timeout to be fairly long, to avoid terminating `rsync` connections prematurely. It's present to let you defend against evil `rsync` server operators who try to tarpit your connection as a form of denial of service attack on `rcynic`.

Default: `300`

max-parallel-fetches

Upper limit on the number of copies of `rsync` that `rcynic` is allowed to run at once. Used properly, this can speed up synchronization considerably when fetching from repositories built with sub-optimal tree layouts or when dealing with unreachable repositories. Used improperly, this option can generate excessive load on repositories, cause synchronization to be interrupted by firewalls, and generally creates create a public nuisance. Use with caution.

As of this writing, values in the range 2-4 are reasonably safe. Values above 10 have been known to cause problems.

`rcynic` can't really detect all of the possible problems created by excessive values of this parameter, but if rcynic's report shows that both successful retrivial and skipped retrieval from the same repository host, that's a pretty good hint that something is wrong, and an excessive value here is a good first guess as to the cause.

Default: `1`

rsync-program

Path to the rsync program.

Default: `rsync`, but you should probably set this variable rather than just trusting the `PATH` environment variable to be set correctly.

log-level

Same as `-l` option on command line. Command line setting overrides config file setting.

Default: `log_log_err`

use-syslog

Same as `-s` option on command line. Command line setting overrides config file setting.

Values: `true` or `false`.

Default: `false`

use-stderr

Same as -e option on command line. Command line setting overrides config file setting.

Values: `true` or `false`.

Default: `false`, but if neither `use-syslog` nor `use-stderr` is set, log output goes to `stderr`.

syslog-facility

Syslog facility to use.

Default: local0

syslog-priority-xyz

(where xyz is an rcynic logging level, above)

Override the syslog priority value to use when logging messages at this rcynic level.

Defaults:

```
syslog-priority-log_sys_err   err
syslog-priority-log_usage_err err
syslog-priority-log_data_err  notice
syslog-priority-log_telemetry info
syslog-priority-log_verbose   info
syslog-priority-log_debug     debug
```

jitter

Startup jitter interval, same as `-j` option on command line. Jitter interval, specified in number of seconds. `rcynic` will pick a random number within the interval from zero to this value, and will delay for that many seconds on startup. The purpose of this is to spread the load from large numbers of `rcynic` clients all running under cron with synchronized clocks, in particular to avoid hammering the global RPKI `rsync` servers into the ground at midnight UTC.

Default: `600`

lockfile

Name of lockfile, or empty for no lock. If you run `rcynic` directly under cron, you should use this parameter to set a lockfile so that successive instances of rcynic don't stomp on each other. If you run `rcynic` under `rcynic-cron`, you don't need to touch this, as `rcynic-cron` maintains its own lock.

Default: no lock

xml-summary

Enable output of a per-host summary at the end of an `rcynic` run in XML format.

Value: filename to which XML summary should be written; "-" will send XML summary to standard output.

Default: no XML summary.

allow-stale-crl

Allow use of CRLs which are past their `nextUpdate` timestamp. This is usually harmless, but since there are attack scenarios in which this is the first warning of trouble, it's configurable.

Values: `true` or `false`.

Default: `true`

prune

Clean up old files corresponding to URIs that `rcynic` did not see at all during this run. `rcynic` invokes `rsync` with the `--delete` option to clean up old objects from collections that `rcynic` revisits, but if a URI changes so that `rcynic` never visits the old collection again, old files will remain in the local mirror indefinitely unless you enable this option.

Note: Pruning only happens when `run-rsync` is true. When the `run-rsync` option is false, pruning is not done regardless of the setting of the prune option option.

Values: `true` or `false`.

Default: `true`

allow-stale-manifest

Allow use of manifests which are past their `nextUpdate` timestamp. This is probably harmless, but since it may be an early warning of problems, it's configurable.

Values: `true` or `false`.

Default: `true`

require-crl-in-manifest

Reject publication point if manifest doesn't list the CRL that covers the manifest EE certificate.

Values: `true` or `false`.

Default: `false`

allow-object-not-in-manifest

Allow use of otherwise valid objects which are not listed in the manifest. This is not supposed to happen, but is probably harmless.

Enabling this does, however, often result in noisier logs, as it increases the chance that `rcynic` will attempt to validate data which a CA removed from the manifest but did not completely remove and revoke from the repository.

Values: `true` or `false`

Default: `false`

allow-digest-mismatch

Allow use of otherwise valid objects which are listed in the manifest with a different digest value.

You probably don't want to touch this.

Values: `true` or `false`

Default: `true`

allow-crl-digest-mismatch

Allow processing to continue on a publication point whose manifest lists a different digest value for the CRL than the digest of the CRL we have in hand.

You probably don't want to touch this.

Values: `true` or `false`

Default: `true`

allow-non-self-signed-trust-anchor

Experimental. Attempts to work around OpenSSL's strong preference for self-signed trust anchors.

We're not going to explain this one in any further detail. If you really want to know what it does, Use The Source.

Do not even consider enabling this option unless you are intimately familiar with both X.509 and the internals of OpenSSL's `X509_verify_cert()` function and really know what you are doing.

Values: `true` or `false`.

Default: `false`

run-rsync

Whether to run `rsync` to fetch data. You don't generally want to change this except when building complex topologies where `rcynic` running on one set of machines acts as aggregators for another set of validators. A large ISP might want to build such a topology so that they could have a local validation cache in each POP while minimizing load on the global repository system and maintaining some degree of internal consistancy between POPs. In such cases, one might want the `rcynic` instances in the POPs to validate data fetched from the aggregators via an external process, without the POP `rcynic` instances attempting to fetch anything themselves.

Values: `true` or `false`.

Default: `true`

use-links

Whether to use hard links rather than copying valid objects from the unauthenticated to authenticated tree. Using links is slightly more fragile (anything that stomps on the unauthenticated file also stomps on the authenticated file) but is a bit faster and reduces the number of inodes consumed by a large data collection. At the moment, copying is the default behavior, but this may change in the future.

Values: `true` or `false`.

Default: `false`

rsync-early

Whether to force `rsync` to run even when we have a valid manifest for a particular publication point and its `nextUpdate` time has not yet passed.

This is an experimental feature, and currently defaults to **true**, which is the old behavior (running `rsync` regardless of whether we have a valid cached manifest). This default may change once we have more experience with `rcynic`'s behavior when run with this option set to `false`.

Skipping the `rsync` fetch when we already have a valid cached manifest can significantly reduce the total number of `rsync` connections we need to make, and significantly reduce the load that each validator places on the authoritative publication servers. As with any caching scheme, however, there are some potential problems involved with not fetching the latest data, and we don't yet have enough experience with this option to know how this will play out in practice, which is why this is still considered experimental.

Values: `true` or `false`

Default: `true` (but may change in the future)

trust-anchor

Specify one RPKI trust anchor, represented as a local file containing an X.509 certificate in DER format. Value of this option is the pathname of the file.

No default.

trust-anchor-locator

Specify one RPKI trust anchor locator, represented as a local file in the format specified in RFC-6490. This a simple text format containing an rsync URI and the RSA public key of the X.509 object specified by the URI; the first line of the file is the URI, the remainder is the public key in Base64 encoded DER format.

Value of this option is the pathname of the file.

No default.

trust-anchor-directory

Specify a directory containing trust anchors, trust anchor locators, or both. Trust anchors in such a directory must have filenames ending in "`.cer`"; trust anchor locators in such a directory must have names ending in "`.tal`"; any other files will be skipped.

This directive is an alternative to using the `trust-anchor` and trust-anchor-locator` directives. This is probably easier to use than the other trust anchor directives when dealing with a collection of trust anchors. This may change on that promised day when we have only a single global trust anchor to deal with, but we're not there yet.

No default.

Post-processing rcynic's XML output

The distribution includes several post-processors for the XML output `rcynic` writes describing the actions it has taken and the validation status of the objects it has found.

rcynic-html

`rcynic-html` converts `rcynic`'s XML output into a collection of HTML pages summarizing the results, noting problems encountered, and showing some history of `rsync` transfer times and repository object counts in graphical form.

`rcynic-cron` runs `rcynic-html` automatically, immediately after running `rcynic`. If for some reason you need to run `rcynic-html` by hand, the command syntax is:

```
$ rcynic-html rcynic.xml /web/server/directory/
```

`rcynic-html` will write a collection of HTML and image files to the specified output directory, along with a set of RRD databases. `rcynic-html` will create the output directory if necessary.

`rcynic-html` requires `rrdtool`, a specialized database and graphing engine designed for this sort of work. You can run `rcynic-html` without `rrdtool` by giving it the `--no-show-graphs` option, but the result won't be as useful.

`rcynic-html` gets its idea of where to find the `rrdtool` program from autoconf, which usually works. If for some reason it doesn't work in your environment, you will need to tell `rcynic-html` where to find `rrdtool`, using the `--rrdtool-binary` option:

```
$ rcynic-html --rrdtoolbinary /some/where/rrdtool rcynic.xml /web/server/directory/
```

rcynic.xsl

`rcynic.xsl` was an earlier attempt at the same kind of HTML output as rcynic-html generates. XSLT was a convenient language for our initial attempts at this, but as the processing involved got more complex, it became obvious that we needed a general purpose programming language.

If for some reason XSLT works better in your environment than Python, you might find this stylesheet to be a useful starting point, but be warned that it's significantly slower than `rcynic-html`, lacks many features, and is no longer under development.

rcynic-text

`rcynic-text` provides a quick flat text summary of validation results. This is useful primarily in test scripts (smoketest uses it).

Usage:

```
$ rcynic-text rcynic.xml
```

validation_status

`validation_status` provides a flat text translation of the detailed validation results. This is useful primarily for checking the detailed status of some particular object or set of objects, perhaps using a program like `grep` or `awk` to filter `validation_status`'s output.

Usage:

```
$ validation_status rcynic.xml
$ validation_status rcynic.xml | fgrep rpki.misbehaving.org
$ validation_status rcynic.xml | fgrep object_rejected
```

rcynic-svn

`rcynic-svn` is a tool for archiving `rcynic`'s results in a Subversion repository. `rcynic-svn` is not integrated into `rcynic-cron`, because this is not something that every relying party is going to want to do. However, for relying parties who want to analyze `rcynic`'s output over a long period of time, `rcynic-svn` may provide a useful starting point starting point.

To use `rcynic-svn`, you first must set up a Subversion repository and check out a working directory:

```
$ svnadmin create /some/where/safe/rpki-archive
$ svn co file:///some/where/safe/rpki-archive /some/where/else/rpki-archive
```

The name can be anything you like, in this example we call it "`rpki-archive`". The above sequence creates the repository, then checks out an empty working directory `/some/where/else/rpki-archive`.

The repository does not need to be on the same machine as the working directory, but it probably should be for simplicity unless you have some strong need to put it elsewhere.

Once you have the repository and working directory set up, you need to arrange for `rcynic-svn` to be run after each `rcynic` run whose results you want to archive. One way to do this would be to run `rcynic-svn` in the same cron job as `rcynic-cron`, immediately after `rcynic-cron` and specifying the same lock file that `rcynic-cron` uses.

Sample usage, assuming that `rcynic`'s data is in the usual place:

```
$ rcynic-svn --lockfile /var/rcynic/data/lock   \
        /var/rcynic/data/authenticated          \
        /var/rcynic/data/unauthenticated        \
        /var/rcynic/data/rcynic.xml             \
        /some/where/else/rpki-archive
```

where the last argument is the name of the Subversion working directory and the other arguments are the names of those portions of `rcynic`'s output which you wish to archive. Generally, the above set (`authenticated`, `unauthenticated`, and `rcynic.xml`) are the ones you want, but feel free to experiment.

rpki-rtr

`rtr-origin` is an implementation of the "RPKI-router" protocol (RFC-6810).

`rtr-origin` depends on `rcynic` to collect and validate the RPKI data. `rtr-origin`'s's job is to serve up that data in a lightweight format suitable for routers that want to do prefix origin authentication.

To use `rtr-origin`, you need to do two things beyond just running `rcynic`:

1. You need to post-process `rcynic`'s output into the data files used by rtr-origin. The `rcynic-cron` script handles this automatically, so the default installation should already be taking care of this for you.
2. You need to set up a listener for the `rtr-origin` server, using the generated data files. The platform-specific packages for FreeBSD, Debian, and Ubuntu automatically set up a plain TCP listener, but you will have to do something on other platforms, or if you're using a transport protocol other than plain TCP.

Post-processing rcynic's output

`rtr-origin` is designed to do the translation from raw RPKI data into the rpki-rtr protocol only once. It does this by pre-computing the answers to all the queries it is willing to answer for a given data set, and storing them on disk. `rtr-origin`'s `--cronjob` mode handles this computation.

To set this up, add an invocation of `rtr-origin --cronjob` to the cron job you're already running to run `rcynic`. As mentioned above, if you're running the `rcynic-cron` script, this is already being done for you automatically, so you don't need to do anything. If you've written your own cron script, you'll need to add something like this to your script:

```
cd /var/rcynic/rpki-rtr
/usr/local/bin/rtr-origin --cronjob /var/rcynic/data/authenticated
```

In `--cronjob` mode, `rtr-origin`, needs write access to a directory where it can store pre-digested versions of the data it pulls from `rcynic`. In the example above, the directory `/var/rcynic/rpki-rtr` should be writable by the user ID that is executing the cron script.

`rtr-origin` creates a collection of data files, as well as a subdirectory in which each instance of the program running in `--server` mode can write a `PF_UNIX` socket file. At present, `rtr-origin` creates these files under the directory in which you run it, hence the `cd` command shown above.

You should make sure that `rtr-origin --cronjob` runs at least once before attempting to configure `--server` mode. Nothing terrible will happen if you don't do this, but `--server` invocations started before the first `--cronjob` run may behave oddly.

Setting up the rpki-rtr server

You need to to set up a server listener that invokes `rtr-origin` in `--server` mode. What kind of server listener you set up depends on which network protocol you're using to transport this protocol. `rtr-origin` is happy to run under inetd, xinetd, sshd, or pretty much anything -- `rtr-origin` doesn't really care, it just reads from `stdin` and writes to `stdout`.

`--server` mode should be run as a non-privileged user (it is read-only for a reason). You may want to set up a separate UNIX userid for this purpose.

`--server` mode takes an optional argument specifying the path to its data directory; if you omit this argument, it uses the directory in in which you run it.

The details of how you set up a listener for this vary depending on the network protocol and the operating system on which you run it. Here are two examples, one for running under `inetd` on FreeBSD, the other for running under `sshd`.

Running rtr-origin --server under inetd

Running under `inetd` with plain TCP is insecure and should only be done for testing, but you can also run it with TCP-MD5 or TCP-AO, or over IPsec. The inetd configuration is generally the same, the details of how you set up TCP-MD5, TCP-AO, or IPsec are platform specific.

To run under `inetd`, you need to:

1. Add an entry to `/etc/services` defining a symbolic name for the port, if one doesn't exist already. At present there is no well-known port defined for this protocol, for this example we'll use port 42420 and the symbolic name `rpki-rtr`.

   Add to `/etc/services`:

```
rpki-rtr          42420/tcp
```

2. Add the service line to `/etc/inetd.conf`:

```
rpki-rtr stream tcp nowait nobody /usr/local/bin/rtr-origin rtr-origin --server /var/rcynic/rpk
```

   This assumes that you want the server to run as user "nobody", which is generally a safe choice, or you could create a new non-priviledged user for this purpose. **DO NOT** run the server as root; it shouldn't do anything bad, but it's a network server that doesn't need root access, therefore it shouldn't have root access.

Running rtr-origin --server under sshd

To run `rtr-origin` under `sshd`, you need to:

1. Decide whether to run a new instance of sshd on a separate port or use the standard port. `rtr-origin` doesn't care, but some people seem to think that it's somehow more secure to run this service on a different port. Setting up `sshd` in general is beyond the scope of this documention, but most likely you can copy the bulk of your configuration from the standard config.
2. Configure `sshd` to know about the `rpki-rtr` subsystem. Add something like this to your `sshd.conf`:

```
Subsystem rpki-rtr /usr/local/bin/rtr-origin
```

3. Configure the userid(s) you expect SSH clients to use to connect to the server. For operational use you almost certainly do *NOT* want this user to have a normal shell, instead you should configure its shell to be the server (`/usr/local/bin/rtr-origin` or wherever you've installed it on your system) and its home directory to be the `rpki-rtr` data directory (`/var/rcynic/rpki-rtr` or whatever you're using). If you're using passwords to authenticate instead of ssh keys (not recommended) you will always need to set the password(s) here when configuring the userid(s).
4. Configure the `.ssh/authorized_keys` file for your clients; if you're using the example values given above, this would be `/var/rcynic/rpki-rtr/.ssh/authorized_keys`. You can have multiple SSH clients using different keys all logging in as the same SSH user, you just have to list all of the SSH keys here. You may want to consider using a `command=` parameter in the key line

(see the `sshd(8)` man page) to lock down the SSH keys listed here so that they can only be used to run the `rpki-rtr` service.

If you're running a separate `sshd` for this purpose, you might also want to add an `!AuthorizedKeysFile` entry pointing at this `authorized_keys` file so that the server will only use this `authorized_keys` file regardless of what other user accounts might exist on the machine:

```
AuthorizedKeysFile /var/rcynic/rpki-rtr/.ssh/authorized_keys
```

There's a sample `sshd.conf` in the source directory. You will have to modify it to suit your environment. The most important part is the `Subsystem` line, which runs the `server.sh` script as the "`rpki-rtr`" service, as required by the protocol specification.

Other transports

You can also run this code under `xinetd`, or the netpipes "`faucet`" program, or `stunnel`...other than a few lines that might need hacking to log the connection peer properly, the program really doesn't care.

You *should*, however, care whether the channel you have chosen is secure; it doesn't make a lot of sense to go to all the trouble of checking RPKI data then let the bad guys feed bad data into your routers anyway because you were running the rpki-rtr link over an unsecured TCP connection.

Other modes

`rtr-origin` has two other modes which might be useful for debugging:

1. `--client` mode implements a dumb client program for this protocol, over SSH, raw TCP, or by invoking `--server` mode directly in a subprocess. The output is not expected to be useful except for debugging. Either run it locally where you run the cron job, or run it anywhere on the net, as in

   ```
   $ rtr-origin --client tcp <hostname> <port>
   ```
2. `--show` mode will display a text dump of pre-digested data files in the current directory.

`rtr-origin` has a few other modes intended to support specific research projects, but they're not intended for general use.

Running relying party tools under cron

rcynic is the primary relying party tool, and it's designed to run under the cron daemon. Consequently, most of the other tools are also designed to run under the cron daemon, so that they can make use of rcynic's output immediately after rcynic finishes a validation run.

rcynic-cron runs the basic set of relying party tools (`rcynic`, `rcynic-html`, and `rtr-origin --cronjob`); if this suffices for your purposes, you don't need to do anything else. This section is a discussion of alternative approaches.

Which tools you want to run depends on how you intend to use the relying party tools. Here we assume a typical case in which you want to gather and validate RPKI data and feed the results to routers using the rpki-rtr protocol. We also assume that everything has been installed in the default locations.

The exact sequence for invoking rcynic itself varies depending both on whether you're using a chroot jail or not and on the platform on which you're running rcynic, as the chroot utilities on different platforms behave slightly differently. Using a chroot jail used to be the default for rcynic, but it turned out that many users found the setup involved to be too complex.

If you're not using rcynic-cron, it's probably simplest to generate a short shell script which calls the tools you want in the correct order, so that's what we show here.

Once you've written this script, install it in your crontab, running at some appropriate interval: perhaps hourly, or perhaps every six hours, depending on your needs. You should run it at least once per day, and probably should not run it more frequently than once per hour unless you really know what you are doing. Please do *NOT* just arrange for the script to run on the hour, instead pick some random minute value within the hour as the start time for your script, to help spread the load on the repository servers.

On FreeBSD or MacOSX, this script might look like this:

```
#!/bin/sh -
/usr/sbin/chroot -u rcynic -g rcynic /var/rcynic /bin/rcynic -c /etc/rcynic.conf || exit
/var/rcynic/bin/rcynic-html /var/rcynic/data/rcynic.xml /usr/local/www/data/rcynic
cd /var/rcynic/rpki-rtr
/usr/bin/su -m rcynic -c '/usr/local/bin/rtr-origin --cronjob /var/rcynic/data/authenticated'
```

This assumes that you have done

```
mkdir /var/rcynic/rpki-rtr
chown rcynic /var/rcynic/rpki-rtr
```

On GNU/Linux systems, the script might look like this if you use the chrootuid program:

```
#!/bin/sh -
/usr/bin/chrootuid /var/rcynic rcynic /bin/rcynic -c /etc/rcynic.conf || exit
/var/rcynic/bin/rcynic-html /var/rcynic/data/rcynic.xml /var/www/rcynic
cd /var/rcynic/rpki-rtr
/usr/bin/su -m rcynic -c '/usr/local/bin/rtr-origin --cronjob /var/rcynic/data/authenticated'
```

If you use the chroot program instead of chrootuid, change the line that invokes rcynic to:

```
/usr/sbin/chroot --userspec rcynic:rcynic /var/rcynic /bin/rcynic -c /etc/rcynic.conf || exit
```

Running a hierarchical rsync configuration

Having every relying party on the Internet contact every publication service is not terribly efficient. In many cases, it may make more sense to use a hierarchical configuration in which a few "gatherer" relying parties contact the publication servers directly, while a collection of other relying parties get their raw data from the gatherers.

Note

The relying parties in this configuration still perform their own validation, they just let the gatherers do the work of collecting the unvalidated data for them.

A gatherer in a configuration like this would look just like a stand-alone relying party as discussed above. The only real difference is that a gatherer must also make its unauthenticated data collection available to other relying parties. Assuming the standard configuration, this will be the directory `/var/rcynic/data/unauthenticated` and its subdirectories.

There are two slightly different ways to do this with rsync:

1. Via unauthenticated rsync, by configuring an `rsyncd.conf` "module", or
2. Via rsync over a secure transport protocol such as ssh.

Since the downstream relying party performs its own validation in any case, either of these will work, but using a secure transport such as ssh makes it easier to track problems back to their source if a downstream relying party concludes that it's been receiving bad data.

Script for a downstream relying party using ssh might look like this:

```
#!/bin/sh -

PATH=/usr/bin:/bin:/usr/local/bin
umask 022
eval `/usr/bin/ssh-agent -s` >/dev/null
/usr/bin/ssh-add /root/rpki_ssh_id_rsa 2>&1 | /bin/fgrep -v 'Identity added:'
hosts='larry.example.org moe.example.org curly.example.org'
for host in $hosts
do
  /usr/bin/rsync --archive --update --safe-links rpkisync@${host}:/var/rcynic/data/unauthentica
done
eval `/usr/bin/ssh-agent -s -k` >/dev/null
for host in $hosts
do
  /usr/sbin/chroot -u rcynic -g rcynic /var/rcynic /bin/rcynic -c /etc/rcynic.conf -u /data/una
  /var/rcynic/bin/rcynic-html /var/rcynic/data/rcynic.xml /usr/local/www/data/rcynic.${host}
done
cd /var/rcynic/rpki-rtr
/usr/bin/su -m rcynic -c '/usr/local/bin/rtr-origin --cronjob /var/rcynic/data/authenticated'
```

where /root/rpki_ssh_id_rsa is an SSH private key authorized to log in as user "rpkisync" on the gatherer machines. If you want to lock this down a little tighter, you could use ssh's `command="..."` mechanism as described in the sshd documentation to restrict the rpkisync user so that it can only run this one rsync command.

If you prefer to use insecure rsync, perhaps to avoid allowing the downstream relying parties any sort of login access at all on the gatherer machines, the configuration would look more like this:

```
#!/bin/sh -

PATH=/usr/bin:/bin:/usr/local/bin
umask 022
```

```
hosts='larry.example.org moe.example.org curly.example.org'
for host in $hosts
do
  /usr/bin/rsync --archive --update --safe-links rsync://${host}/unauthenticated/ /var/rcynic/d
done
for host in $hosts
do
  /usr/sbin/chroot -u rcynic -g rcynic /var/rcynic /bin/rcynic -c /etc/rcynic.conf -u /data/una
  /var/rcynic/bin/rcynic-html /var/rcynic/data/rcynic.xml /usr/local/www/data/rcynic.${host}
done
cd /var/rcynic/rpki-rtr
/usr/bin/su -m rcynic -c '/usr/local/bin/rtr-origin --cronjob /var/rcynic/data/authenticated'
```

where "unauthenticated" here is an rsync module pointing at `/var/rcynic/data/unauthenticated` on each of the gatherer machines. Configuration for such a module would look like:

```
[unauthenticated]
    read only           = yes
    transfer logging    = yes
    path                = /var/rcynic/data/unauthenticated
    comment             = Unauthenticated RPKI data
```

Running rcynic chrooted

This is an attempt to describe the process of setting up rcynic in a chrooted environment. The installation scripts that ship with rcynic attempt to do this automatically when requested for the platforms we support, but the process is somewhat finicky, so some explanation seems in order. If you're running on one of the supported platforms, the following steps may be handled for you by the Makefiles, but you may still want to understand what all this is trying to do.

rcynic itself does not include any direct support for running chrooted, but is designed to be (relatively) easy to run in a chroot jail.

To enable chroot support during installation, you should install from source and use the `--enable-rcynic-jail` option to `./configure`.

rcynic-cron includes support for running chrooted. To use it, specify the `--chroot` option on rcynic-cron's command line. This will cause rcynic-cron to run rcynic in the chrooted environment. Note that, in order for this to work, rcynic-cron itself must run as root, since only root can issue the `chroot()` system call. When run as root, rcynic-cron takes care of changing the user ID of each process it starts to the unprivileged "`rcynic`" user.

Creating the chroot jail environment

By far the most tedious and finicky part of setting up rcynic to run in a chroot jail is setting the jail itself. The underlying principal is simple and obvious: a process running in the jail can't use files outside the jail. The difficulty is that the list of files that needs to be in the jail is system-dependent, can be rather long, and sometimes can only be discovered by trial and error.

You'll either need staticly linked copies of rcynic and rsync, or you'll need to figure out which shared libraries these programs need (try using the "ldd" command). Here we assume staticly linked binaries, because that's simpler, but be warned that statically linked binaries are not even possible on some platforms, whether due to concious decisions on the part of operating system vendors or due to hidden use of dynamic loading by other libraries at runtime. Once again, the Makefiles attempt to do the correct thing for your environment if they know what it is, but they might get it wrong.

You'll need a chroot wrapper program. As mentioned above, rcynic-cron can act as that wrapper program; if this works for you, we recommend it, because it works the same way on all platforms and doesn't require additional external programs. Otherwise, you'll have to find a suitable wrapper program. Your platform may already have one (FreeBSD does -- `/usr/sbin/chroot`), but if you don't, you can download Wietse Venema's "chrootuid" program from ftp://ftp.porcupine.org/pub/security/chrootuid1.3.tar.gz.

Warning

> The chroot program included in at least some GNU/Linux distributions is not adaquate to this task. You need a wrapper that knows how to drop privileges after performing the chroot() operation itself. If in doubt, use chrootuid.

Unfortunately, the precise details of setting up a proper chroot jail vary wildly from one system to another, so the following instructions may not be a precise match for the preferred way of doing this on your platform. Please feel free to contribute scripts for other platforms.

1. Build the static binaries. You might want to test them at this stage too, although you can defer that until after you've got the jail built.
2. Create a userid under which to run rcynic. Here we'll assume that's a user named "rcynic", whose default group is also named "rcynic". Do not add any other userids to the rcynic group unless you really know what you are doing.

3. Build the jail. You'll need, at minimum, a directory in which to put the binaries, a subdirectory tree that's writable by the userid which will be running rcynic and rsync, your trust anchors, and whatever device inodes the various libraries need on your system. Most likely the devices that matter will be `/dev/null`, `/dev/random`, and `/dev/urandom`; if you're running a FreeBSD system with devfs, you do this by mounting and configuring a devfs instance in the jail, on other platforms you probably use the `mknod` program or something similar.

Important
: Other than the directories that you want rcynic and rsync to be able to modify, *nothing* in the initial jail setup should be writable by the rcynic userid. In particular, rcynic and rsync should *not* be allowed to modify: their own binary images, any of the configuration files, or your trust anchors. It's simplest just to have root own all the files and directories that rcynic and rsync are not allowed to modify, and make sure that the permissions for all of those directories and files make them writable only by root.

Sample jail tree, assuming that we're putting all of this under `/var/rcynic`:

```
$ mkdir /var/rcynic
$ mkdir /var/rcynic/bin
$ mkdir /var/rcynic/data
$ mkdir /var/rcynic/dev
$ mkdir /var/rcynic/etc
$ mkdir /var/rcynic/etc/trust-anchors
```

Copy your trust anchors into `/var/rcynic/etc/trust-anchors`.

Copy the staticly linked rcynic and rsync into `/var/rcynic/bin`.

Copy `/etc/resolv.conf` and `/etc/localtime` (if it exists) into `/var/rcynic/etc`.

Write an rcynic configuration file as `/var/rcynic/etc/rcynic.conf`. Path names in this file must match the jail setup, more on this below.

```
$ chmod -R go-w /var/rcynic
$ chown -R root:wheel /var/rcynic
$ chown -R rcynic:rcynic /var/rcynic/data
```

If you're using devfs, arrange for it to be mounted at `/var/rcynic/dev`; otherwise, create whatever device inodes you need in `/var/rcynic/dev` and make sure that they have sane permissions (copying whatever permissions are used in your system `/dev` directory should suffice).

`rcynic.conf` to match this configuration:

```
[rcynic]

rsync-program           = /bin/rsync
authenticated           = /data/authenticated
unauthenticated         = /data/unauthenticated
xml-summary             = /data/rcynic.xml
trust-anchor-directory  = /etc/trust-anchors
```

Once you've got all this set up, you're ready to try running rcynic in the jail. Try it from the command line first, then if that works, you should be able to run it under cron.

Note: chroot, chrootuid, and other programs of this type are usually intended to be run by root, and should *not* be setuid programs unless you *really* know what you are doing.

Sample command line:

```
$ /usr/local/bin/chrootuid /var/rcynic rcynic /bin/rcynic -s -c /etc/rcynic.conf
```

Note that we use absolute pathnames everywhere. This is not an accident. Programs running in jails under cron should not make assumptions about the current working directory or environment variable settings, and programs running in chroot jails would need different `PATH` settings anyway. Best just to specify everything.

Building static binaries

On FreeBSD, building a staticly linked rsync is easy: one just sets the environment variable `LDFLAGS='-static'` before building rsync and the right thing will happen. Since this is really just GNU configure picking up the environment variable, the same trick should work on other platforms...except that some compilers don't support `-static`, and some platforms are missing some or all of the non-shared libraries you'd need to link the resulting binary.

For simplicity, we've taken the same approach with rcynic, so

```
$ make LDFLAGS='-static'
```

works. This isn't necessary on platforms where we know that static linking works -- the default is static linking where supported.

syslog from chrooted environment

Depending on how the `syslog()` library call and the syslog daemon (`syslogd`, `rsyslogd`, ...) are implemented on your platform, syslog may not work properly with rcynic in a chroot jail. On FreeBSD, the easiest way to fix this is to add the following lines to /etc/rc.conf:

```
altlog_proglist="named rcynic"
rcynic_chrootdir="/var/rcynic"
rcynic_enable="YES"
```

This tells syslogd to listen on an additional `PF_UNIX` socket within rcynic's chroot jail.

RPKI CA Engine

The RPKI CA engine is an implementation of the production-side tools for generating certificates, CRLs, ROAs, and other RPKI objects. The CA tools are implemented primarily in Python, with an extension module linked against an RFC-3779-enabled version of the OpenSSL libraries to handle some of the low-level details.

See the relying party tools for tools for retrieving, verifying, and using RPKI data.

Getting started

If you just want to get started with the CA tools and hate reading documentation, here's a roadmap on what you do need to read:

1. Start with the installation instructions; if you're using pre-built packages you may be able to skip this step.
2. Then read the configuration instructions
3. Then the MySQL setup instructions
4. And finally either the command line tool or web interface

Overview of the CA engine

Terminology

A few special terms appear often enough in code and documentation that they need explaining.

IRBE::

"Internet Registry Back End."

IRDB::

"Internet Registry Data Base."

BPKI::

"Business PKI."

RPKI::

"Resource PKI."

Programs

See the installation instructions for how to build and install the code.

The RPKI CA engine includes the following programs:

rpkid::

The main RPKI engine daemon.

pubd::

The publication engine daemon.

rootd::

A separate daemon for handling the root of an RPKI certificate tree. This is essentially a stripped down version of rpkid with no SQL database, no left-right protocol implementation, and only the parent side of the up-down protocol. It's separate because the root is a special case in several ways and it was simpler to keep the special cases out of the main daemon.

irdbd::

A sample implementation of an IR database daemon. rpkid calls into this to perform lookups via the left-right protocol.

rpkic::

A command line interface to control rpkid and pubd.

GUI::

A web-based graphical interface to control rpkid and pubd.

irdbd, rpkic, and the GUI collectively make up the "Internet registry back end" (IRBE) component of the system.

These programs take configuration files in a common format similar to that used by the OpenSSL command line tool, see the configuration guide for details.

Basic operation consists of creating the appropriate MySQL databases (see MySQL setup), starting the daemons, and using rpkic or the web interface to configure relationships between parents and children, relationships between publication clients and repositories, allocate resources to children, and create ROAs. Once setup is complete, rpkid should maintain the requested data automatically, including re-querying its parent(s) periodically to check for changes, reissuing certificates and other objects as needed, and so forth.

The daemons are all event-driven, and are (in theory) capable of supporting an arbitrary number of hosted RPKI engines to run in a single rpkid instance, up to the performance limits of the underlying hardware.

Starting the servers

You need to follow the instructions in the configuration guide before attempting to start the servers.

Once you've written the servers' configuration file, the easiest way to run the servers is to run the `rpki-start-servers` script, which examines your `rpki.conf` file and starts the appropriate servers in background.

If you prefer, you can run each server by hand instead of using the script, eg, using Bourne shell syntax to run rpkid in background:

```
rpkid &
echo >rpkid.pid  "$!"
```

You can also use separate configuration files for each server if necessary, run multiple copies of the same server with different configuration files, and so forth.

All of the daemons use syslog by default. You can change this by running either the servers themselves or the `rpki-start-servers` script with the "-d" option. Used as an argument to a server directly, "-d" causes that server to log to stderr instead of to syslog. Used as an argument to `rpki-start-servers`, "-d" starts each of the servers with "-d" while redirecting stderr from each server to a separate log file. This is intended primarily for debugging.

Some of the configuration options are common to all daemons: which daemon they affect depends only on which sections of the configuration file they are in. See Common Options for details.

rpkid

rpkid is the main RPKI engine daemon. Configuration of rpkid is a two step process: a config file to bootstrap rpkid to the point where it can speak using the left-right protocol, followed by dynamic configuration via the left-right protocol. The latter stage is handled by the command line tool or the web interface.

rpkid stores dynamic data in an SQL database, which must have been created for it, as explained in in the MySQL setup instructions.

pubd

pubd is the publication daemon. It implements the server side of the publication protocol, and is used by rpkid to publish the certificates and other objects that rpkid generates.

pubd is separate from rpkid for two reasons:

- The hosting model allows entities which choose to run their own copies of rpkid to publish their output under a common publication point. In general, encouraging shared publication services where practical is a good thing for relying parties, as it will speed up rcynic synchronization time.
- The publication server has to run on (or at least close to) the publication point itself, which in turn must be on a publically reachable server to be useful. rpkid, on the other hand, need only be reachable by the IRBE and its children in the RPKI tree. rpkid is a much more complex piece of software than pubd, so in some situations it might make sense to wrap tighter firewall constraints around rpkid than would be practical if rpkid and pubd were a single program.

pubd stores dynamic data in an SQL database, which must have been created for it, as explained in the MySQL setup instructions. pubd also stores the published objects themselves as disk files in a configurable location which should correspond to an appropriate module definition in rsync.conf; see the configuration guide for details.

rootd

rootd is a stripped down implmenetation of (only) the server side of the up-down protocol. It's a separate program because the root certificate of an RPKI certificate tree requires special handling and may also require a special handling policy. rootd is a simple implementation intended for test use, it's not suitable for use in a production system. All configuration comes via the config file; see the configuration guide for details.

irdbd

irdbd is a sample implemntation of the server side of the IRDB callback subset of the left-right protocol. In production use this service is a function of the IRBE stub; irdbd may be suitable for production use in simple cases, but an IR with a complex IRDB may need to extend or rewrite irdbd.

irdbd is part of the IR back-end system, and shares its SQL database with rpkic and the web interface.

The package actually includes a second implementation of irdbd, used only for testing: `rpkid/tests/old_irdbd` is a mininmal implementation, used only by smoketest, which itself constitues a fairly complete (if rather strange) IRBE implementatation. Ordinarly you won't care about this, but if for some reason you need to write your own irdbd implementation, you might find it easier to start from the minimal version.

See the configuration guide for details on configuring irdbd.

Test programs

The package includes two separate test programs, which take similar test description files but use them in different ways. The test tools are only present in the source tree ("make install" does not install them).

Unlike the configuration files used by the other programs, these test programs read test descriptions written in the YAML serialization language (see http://www.yaml.org/ for more information on YAML). Each test script describes a hierarchy of RPKI entities, including hosting relationships and resource assignments, in a relatively compact form. The test programs use these descriptions to generate a set of configuration files, populate the back end database, and drive the test.

See the test configuration language for details on the content of these YAML files.

smoketest

smoketest is a test harness to set up and run a collection of rpkid and irdbd instances under scripted control. The YAML test description defines the test configuration for smoketest to run, including initial resource assignments. Subsequent YAML "documents" in the same description file define an ordered series of changes to be made to the configuration. smoketest runs the rcynic RPKI validator between each update cycle, to check the output of the CA programs.

smoketest is designed to support running a fairly wide set of test configurations as canned scripts, without writing any new control code. The intent is to make it possible to write meaningful regression tests.

yamltest

yamltest is another test harness to set up and run a collection of rpkid and irdbd instances under scripted control. It is similar in many ways to, and uses the same YAML test description language, but its purpose is different: smoketest runs a particular test scenario through a series of changes, then shuts it down; yamltest, on the other hand, sets up a test network using the same tools that a real user would use (principally the rpkic tool), and leaves the test running indefinitely.

At present, this means that yamltest ignores all but the first "document" in a test description file. This may change in the future.

Running yamltest will generate a fairly complete set configuration files, which may be useful as examples.

Configuring the RPKI CA tools: `rpki.conf`

This section describes `rpki.conf`, the the configuration file for the RPKI CA tools.

The first subsection is a quick summary of the options you're most likely to need to configure (or at least check) for a basic setup.

The rest of this section contains a more complete reference to the configuration file and some of the things you might need to do with it if your needs are more complex.

There are a lot of configuration options, but in most cases you will never have to touch more than a few of them. Keep reading, and don't panic.

Quick guide to the most common configuration options

This subsection describes only a handful of `rpki.conf` configuration options. These are the ones you'll need to set, or at least check, as part of initial installation. In general, the installation process will have already set sane values for these, but you may need to a few of them depending on exactly what you're doing.

The location of `rpki.conf` varies depending on the operating system you're running and how you installed the software. Unless you did something unusual during installation, it's either `/etc/rpki.conf` or `/usr/local/etc/rpki.conf`.

- All of the configuration options you're most likely to need to change are in the `[myrpki]` section of `rpki.conf`.

```
[myrpki]
```

- You need to check the setting of `rpkid_server_host`. The installation process sets this to the fully-qualified DNS hostname of the server on which you installed the code, but if you use a service-specific DNS name for RPKI service you will need to change this option to match that service name.

```
rpkid_server_host               = rpkid.example.org
```

- You need to set the value of `run_pubd` to reflect whether you intend to run your own RPKI publication server and rsync server.

```
run_pubd                        = yes
```

or

```
run_pubd                        = no
```

- If you are running your own RPKI publication server, you need to check the setting of `pubd_server_host`. The installation process sets this to the fully-qualified DNS hostname of the server on which you installed the code, but if you use a service-specific DNS name for RPKI publication service you will need to change this option to match that service name.

```
pubd_server_host                = pubd.example.org
```

There are *many* other configuration options, but setting the above correctly should suffice to get you started with the default configuration. Read on for details if you need to know more, otherwise go to next steps.

Configuration file syntax

The general format of `rpki.conf` is the same as the configuration language used by many other programs, including the OpenSSL package. The file is divided into "sections", labeled with square brackets; individual options within a section look like variable assignments, with the option name on the left and the option value on the right.

```
[foo]

bar = fred
baz = 42
```

The configuration file parser supports a limited version of the macro facility used in OpenSSL's configuration parser. An expression such as

```
foo = ${bar::baz}
```

sets foo to the value of the baz variable from section bar.

The section name `ENV` is special: it refers to environment variables.

```
home = ${ENV::HOME}
```

Each of the programs that make up the RPKI tookit can potentially take its own configuration file, but for most uses this is unnecessarily complicated. The recommended approach is to use a single configuration file, and to put all of the parameters that a normal user might need to change into a single section of that configuration file, then reference these common settings from the program-specific sections of the configuration file via macro expansion.

The default name for the shared configuration file is `rpki.conf`. The location of the system-wide `rpki.conf` file is selected by `./configure` during installation. The default location is `/usr/local/etc/rpki.conf` when building from source or on platforms like FreeBSD or MacOSX where packaged software goes in the `/usr/local` tree; on GNU/Linux platforms, binary packages will use `/etc/rpki.conf` per GNU/Linux convention.

Regardless of the default location, you can override the build-time default filename at runtime if necessary by setting the `RPKI_CONF` environment variable to the name of the configuration file you want to use. Most of the programs also take a command-line option (generally "`-c`") specifying the name of the configuration file; if both the command line option and the environment variable are set, the command line option wins.

The installation process builds a sample configuration file `rpki.conf.sample` and installs it alongside of `rpki.conf`. If you have no `rpki.conf` installed, the installation process will copy `rpki.conf.sample` to `rpki.conf`, but it will not overwrite an existing `rpki.conf` file.

Too much information about `rpki.conf` options

The list of options that you can set in `rpki.conf` is ridiculously long. The default configuration includes what we hope are reasonable default settings for all of them, so in many cases you will never need to know about most of these options. A number of the options for individual programs are specified in terms of other options, using the macro facility described above.

In general, if you don't understand what an option does, you probably should leave it alone.

Detailed information about individual options is listed in separate sections, one per section of `rpki.conf`. These documentation sections are generated from the same source file as the sample configuration file.

rsyncd.conf

If you're running pubd, you'll also need to run rsyncd. Your rsyncd configuration will need to match your pubd configuration in order for relying parties to find the RPKI objects managed by pubd.

Here's a sample rsyncd.conf file:

```
pid file        = /var/run/rsyncd.pid
uid             = nobody
gid             = nobody

[rpki]
    use chroot          = no
    read only           = yes
    transfer logging    = yes
    path                = /some/where/publication
    comment             = RPKI publication
```

You may need to adapt this to your system. In particular, you will need to set the `path` option to match the directory you named as `publication_base_directory` in `rpki.conf`.

You may need to do something more complicated if you are already running rsyncd for other purposes. See the `rsync(1)` and `rsyncd.conf(5)` manual pages for more details.

Running your own RPKI root

In general, we do not recommend running your own RPKI root environment, for various reasons. If, however, you need to do so, you should read the documentation for the [rootd] section , and the instructions for creating a RPKI root certificate .

Running rpkid or pubd on a different server

The default configuration runs rpkid, pubd (if enabled) and the back end code all on the same server. For most purposes, this is fine, but in some cases you might want to split these functions up among different servers. If you need to do this, see these instructions.

Configuring the test harness

We expect the test harness to be of interest primarily to developers, but if you need to understand how it works, you will probably want to read these instructions.

Next steps

Once you've finished with configuration, the next thing you should read is the MySQL setup instructions.

RPKI Engine Common Configuration Options

Some of the configuration options are common to all of the daemons. Which daemon they affect depends only on which sections of which configuration file they are in.

The first group of options are boolean flags, which can be set to "true" or "false". If not specified, default values will be chosen (generally false). Many of these flags controll debugging code that is probably of interest only to the developers.

debug_http::

Enable verbose http debug logging.

want_persistent_client::

Enable http 1.1 persistence, client side.

want_persistent_server::

Enable http 1.1 persistence, server side.

use_adns::

Use asynchronous DNS code. Enabling this will raise an exception if the dnspython toolkit is not installed. Asynchronous DNS is an experimental feature intended to allow higher throughput on busy servers; if you don't know why you need it, you probably don't.

enable_ipv6_clients::

Enable IPv6 HTTP client code.

enable_ipv6_servers::

Enable IPv6 HTTP server code. On by default, since listening for IPv6 connections is usually harmless.

debug_cms_certs::

Enable verbose logging about CMS certificates.

sql_debug::

Enable verbose logging about sql operations.

gc_debug::

Enable scary garbage collector debugging.

timer_debug::

Enable verbose logging of timer system.

enable_tracebacks::

Enable Python tracebacks in logs.

There are also a few options which allow you to save CMS messages for audit or debugging. The save format is a simple MIME encoding in a {{http://en.wikipedia.org/wiki/Maildir|Maildir}-format mailbox. The current options are very crude, at some point we may provide finer grain controls.

dump_outbound_cms::

Dump verbatim copies of CMS messages we send to this mailbox.

dump_inbound_cms::

Dump verbatim copies of CMS messages we receive to this mailbox.

[myrpki] section

The "`[myrpki]`" section contains all the parameters that you really need to configure. The name "`myrpki`" is historical and may change in the future.

handle

Every resource-holding or server-operating entity needs a "handle", which is just an identifier by which the entity calls itself. Handles do not need to be globally unique, but should be chosen with an eye towards debugging operational problems: it's best if you use a handle that your parents and children will recognize as being you.

The "`handle`" option in the "`[myrpki]`" section specifies the default handle for this installation. Previous versions of the CA tools required a separate configuration file, each with its own handle setting, for each hosted entity. The current code allows the current handle to be selected at runtime in both the GUI and command line user interface tools, so the handle setting here is just the default when you don't set one explictly. In the long run, this option may go away entirely, but for now you need to set this.

Syntax is an identifier (ASCII letters, digits, hyphen, underscore -- no whitespace, non-ASCII characters, or other punctuation).

No default value.

bpki_servers_directory

Directory for BPKI files generated by rpkic and used by rpkid and pubd. You will not normally need to change this.

```
bpki_servers_directory = ${autoconf::datarootdir}/rpki
```

run_rpkid

Whether you want to run your own copy of rpkid (and irdbd). Leave this alone unless you're doing something unusual like running a pubd-only installation.

```
run_rpkid = yes
```

rpkid_server_host

DNS hostname for rpkid. In most cases, this must resolve to a publicly-reachable address to be useful, as your RPKI children will need to contact your rpkid at this address.

No default value.

rpkid_server_port

Server port number for rpkid. This can be any legal TCP port number that you're not using for something else.

```
rpkid_server_port = 4404
```

irdbd_server_host

DNS hostname for irdbd, or "`localhost`". This should be "`localhost`" unless you really know what you are doing.

```
irdbd_server_host = localhost
```

irdbd_server_port

Server port number for irdbd. This can be any legal TCP port number that you're not using for something else.

```
irdbd_server_port = 4403
```

run_pubd

Whether you want to run your own copy of pubd. In general, it's best to use your parent's pubd if your parent allows you to do so, because this will reduce the overall number of publication sites from which relying parties will need to retrieve data. However, not all parents offer publication service, or you may need to run pubd yourself for reliability reasons, or because you're certifying private address space or private Autonomous System Numbers.

The out of band setup protocol will attempt to negotiate publication service for you with whatever publication service your parent is using, if it can and if you let it.

```
run_pubd = yes
```

pubd_server_host

DNS hostname for pubd, if you're running it. This must resolve to a publicly reachable address to be useful.

No default value.

pubd_server_port

Server port number for pubd. This can be any legal TCP port number that you're not using for something else.

```
pubd_server_port = 4402
```

pubd_contact_info

Contact information to include in offers of repository service. This only matters when you're running pubd. This should be a human readable string, perhaps containing an email address or URL.

No default value.

run_rootd

Whether you want to run your very own copy of rootd. Don't enable this unless you really know what you're doing.

```
run_rootd = no
```

rootd_server_host

DNS hostname for rootd, if you're running it. This should be localhost unless you really know what you are doing.

```
rootd_server_host = localhost
```

rootd_server_port

Server port number for rootd, if you're running it. This can be any legal TCP port number that you're not using for something else.

```
rootd_server_port = 4401
```

publication_base_directory

Root of local directory tree where pubd should write out published data. You need to configure this, and the configuration should match up with the directory where you point rsyncd. Neither pubd nor rsyncd much cares *where* you tell it to put this stuff, the important thing is that the rsync URIs in generated certificates match up with the published objects so that relying parties can find and verify rpkid's published outputs.

```
publication_base_directory = ${autoconf::datarootdir}/rpki/publication
```

publication_root_cert_directory

Root of local directory tree where rootd (sigh) should write out published data. This is just like publication_base_directory, but rootd is too dumb to use pubd and needs its own directory in which to write one certificate, one CRL, and one manifest. Neither rootd nor rsyncd much cares *where* you tell them to put this stuff, the important thing is that the rsync URIs in generated certificates match up with the published objects so that relying parties can find and verify rootd's published outputs.

```
publication_root_cert_directory = ${myrpki::publication_base_directory}.root
```

publication_rsync_module

rsyncd module name corresponding to publication_base_directory. This has to match the module you configured into `rsyncd.conf`. Leave this alone unless you have some need to change it.

```
publication_rsync_module = rpki
```

publication_root_module

rsyncd module name corresponding to publication_root_cert_directory. This has to match the module you configured into `rsyncd.conf`. Leave this alone unless you have some need to change it.

```
publication_root_module = root
```

publication_rsync_server

Hostname and optional port number for rsync URIs. In most cases this should just be the same value as pubd_server_host.

```
publication_rsync_server = ${myrpki::pubd_server_host}
```

start_rpkid

rpkid startup control. This should usually have the same value as run_rpkid: the only case where you would want to change this is when you are running the back-end code on a different machine from one or more of the daemons, in which case you need finer control over which daemons to start on which machines. In such cases, run_rpkid controls whether the back-end code is doing things to manage rpkid, while start_rpkid controls whether rpki-start-servers attempts to start rpkid on this machine.

```
start_rpkid = ${myrpki::run_rpkid}
```

start_irdbd

irdbd startup control. This should usually have the same value as run_rpkid: the only case where you would want to change this is when you are running the back-end code on a different machine from one or more of the daemons, in which case you need finer control over which daemons to start on which machines. In such cases, run_rpkid controls whether the back-end code is doing things to manage rpkid, while start_irdbd controls whether rpki-start-servers attempts to start irdbd on this machine.

```
start_irdbd = ${myrpki::run_rpkid}
```

start_pubd

pubd startup control. This should usually have the same value as run_pubd: the only case where you would want to change this is when you are running the back-end code on a different machine from one or more of the daemons, in which case you need finer control over which daemons to start on which machines. In such cases, run_pubd controls whether the back-end code is doing things to manage pubd, while start_pubd controls whether rpki-start-servers attempts to start pubd on this machine.

```
start_pubd = ${myrpki::run_pubd}
```

start_rootd

rootd startup control. This should usually have the same value as run_rootd: the only case where you would want to change this is when you are running the back-end code on a different machine from one or more of the daemons, in which case you need finer control over which daemons to start on which machines. In such cases, run_rootd controls whether the back-end code is doing things to manage rootd, while start_rootd controls whether rpki-start-servers attempts to start rootd on this machine.

```
start_rootd = ${myrpki::run_rootd}
```

shared_sql_username

If you're comfortable with having all of the databases use the same MySQL username, set that value here. The default setting of this variable should be fine.

```
shared_sql_username = rpki
```

shared_sql_password

If you're comfortable with having all of the databases use the same MySQL password, set that value here. You should use a locally generated password either here or in the individual settings below. The installation process generates a random value for this option, which satisfies this requirement, so ordinarily you should have no need to change this option.

No default value.

rpkid_sql_database

SQL database name for rpkid's database. The default setting of this variable should be fine.

```
rpkid_sql_database = rpkid
```

rpkid_sql_username

If you want to use a separate SQL username for rpkid's database, set it here.

```
rpkid_sql_username = ${myrpki::shared_sql_username}
```

rpkid_sql_password

If you want to use a separate SQL password for rpkid's database, set it here.

```
rpkid_sql_password = ${myrpki::shared_sql_password}
```

irdbd_sql_database

SQL database for irdbd's database. The default setting of this variable should be fine.

```
irdbd_sql_database = irdbd
```

irdbd_sql_username

If you want to use a separate SQL username for irdbd's database, set it here.

```
irdbd_sql_username = ${myrpki::shared_sql_username}
```

irdbd_sql_password

If you want to use a separate SQL password for irdbd's database, set it here.

```
irdbd_sql_password = ${myrpki::shared_sql_password}
```

pubd_sql_database

SQL database name for pubd's database. The default setting of this variable should be fine.

```
pubd_sql_database = pubd
```

pubd_sql_username

If you want to use a separate SQL username for pubd's database, set it here.

```
pubd_sql_username = ${myrpki::shared_sql_username}
```

pubd_sql_password

If you want to use a separate SQL password for pubd's database, set it here.

```
pubd_sql_password = ${myrpki::shared_sql_password}
```

[rpkid] section

rpkid's default config file is the system `rpki.conf` file. Start rpkid with "`-c filename`" to choose a different config file. All options are in the "`[rpkid]`" section. BPKI Certificates and keys may be in either DER or PEM format.

sql-database

MySQL database name for rpkid.

```
sql-database = ${myrpki::rpkid_sql_database}
```

sql-username

MySQL user name for rpkid.

```
sql-username = ${myrpki::rpkid_sql_username}
```

sql-password

MySQL password for rpkid.

```
sql-password = ${myrpki::rpkid_sql_password}
```

server-host

Host on which rpkid should listen for HTTP service requests.

```
server-host = ${myrpki::rpkid_server_host}
```

server-port

Port on which rpkid should listen for HTTP service requests.

```
server-port = ${myrpki::rpkid_server_port}
```

irdb-url

HTTP service URL rpkid should use to contact irdbd. If irdbd is running on the same machine as rpkid, this can and probably should be a loopback URL, since nobody but rpkid needs to talk to irdbd.

```
irdb-url = http://${myrpki::irdbd_server_host}:${myrpki::irdbd_server_port}/
```

bpki-ta

Where rpkid should look for the BPKI trust anchor. All BPKI certificate verification within rpkid traces back to this trust anchor. Don't change this unless you really know what you are doing.

```
bpki-ta = ${myrpki::bpki_servers_directory}/ca.cer
```

rpkid-cert

Where rpkid should look for its own BPKI EE certificate. Don't change this unless you really know what you are doing.

```
rpkid-cert = ${myrpki::bpki_servers_directory}/rpkid.cer
```

rpkid-key

Where rpkid should look for the private key corresponding to its own BPKI EE certificate. Don't change this unless you really know what you are doing.

```
rpkid-key = ${myrpki::bpki_servers_directory}/rpkid.key
```

irdb-cert

Where rpkid should look for irdbd's BPKI EE certificate. Don't change this unless you really know what you are doing.

```
irdb-cert = ${myrpki::bpki_servers_directory}/irdbd.cer
```

irbe-cert

Where rpkid should look for the back-end control client's BPKI EE certificate. Don't change this unless you really know what you are doing.

```
irbe-cert = ${myrpki::bpki_servers_directory}/irbe.cer
```

[irdbd] section

irdbd's default configuration file is the system `rpki.conf` file. Start irdbd with "`-c filename`" to choose a different configuration file. All options are in the "`[irdbd]`" section.

Since irdbd is part of the back-end system, it has direct access to the back-end's SQL database, and thus is able to pull its own BPKI configuration directly from the database, and thus needs a bit less configuration than the other daemons.

sql-database

MySQL database name for irdbd.

```
sql-database = ${myrpki::irdbd_sql_database}
```

sql-username

MySQL user name for irdbd.

```
sql-username = ${myrpki::irdbd_sql_username}
```

sql-password

MySQL password for irdbd.

```
sql-password = ${myrpki::irdbd_sql_password}
```

server-host

Host on which irdbd should listen for HTTP service requests.

```
server-host = ${myrpki::irdbd_server_host}
```

server-port

Port on which irdbd should listen for HTTP service requests.

```
server-port = ${myrpki::irdbd_server_port}
```

startup-message

String to log on startup, useful when debugging a collection of irdbd instances at once.

No default value.

[pubd] section

pubd's default configuration file is the system `rpki.conf` file. Start pubd with "`-c filename`" to choose a different configuration file. All options are in the "`[pubd]`" section. BPKI certificates and keys may be either DER or PEM format.

sql-database

MySQL database name for pubd.

```
sql-database = ${myrpki::pubd_sql_database}
```

sql-username

MySQL user name for pubd.

```
sql-username = ${myrpki::pubd_sql_username}
```

sql-password

MySQL password for pubd.

```
sql-password = ${myrpki::pubd_sql_password}
```

publication-base

Root of directory tree where pubd should write out published data. You need to configure this, and the configuration should match up with the directory where you point rsyncd. Neither pubd nor rsyncd much cares -where- you tell them to put this stuff, the important thing is that the rsync URIs in generated certificates match up with the published objects so that relying parties can find and verify rpkid's published outputs.

```
publication-base = ${myrpki::publication_base_directory}
```

server-host

Host on which pubd should listen for HTTP service requests.

```
server-host = ${myrpki::pubd_server_host}
```

server-port

Port on which pubd should listen for HTTP service requests.

```
server-port = ${myrpki::pubd_server_port}
```

bpki-ta

Where pubd should look for the BPKI trust anchor. All BPKI certificate verification within pubd traces back to this trust anchor. Don't change this unless you really know what you are doing.

```
bpki-ta = ${myrpki::bpki_servers_directory}/ca.cer
```

pubd-cert

Where pubd should look for its own BPKI EE certificate. Don't change this unless you really know what you are doing.

```
pubd-cert = ${myrpki::bpki_servers_directory}/pubd.cer
```

pubd-key

Where pubd should look for the private key corresponding to its own BPKI EE certificate. Don't change this unless you really know what you are doing.

```
pubd-key = ${myrpki::bpki_servers_directory}/pubd.key
```

irbe-cert

Where pubd should look for the back-end control client's BPKI EE certificate. Don't change this unless you really know what you are doing.

```
irbe-cert = ${myrpki::bpki_servers_directory}/irbe.cer
```

[rootd] section

You don't need to run rootd unless you're IANA, are certifying private address space, or are an RIR which refuses to accept IANA as the root of the public address hierarchy.

Ok, if that wasn't enough to scare you off: rootd is a mess, and needs to be rewritten, or, better, merged into rpkid. It doesn't use the publication protocol, and it requires far too many configuration parameters.

rootd was originally intended to be a very simple program which simplified rpkid enormously by moving one specific task (acting as the root CA of an RPKI certificate hierarchy) out of rpkid. As the specifications and code (mostly the latter) have evolved, however, this task has become more complicated, and rootd would have to become much more complicated to keep up.

Don't run rootd unless you're sure that you need to do so.

Still think you need to run rootd? OK, but remember, you have been warned....

rootd's default configuration file is the system `rpki.conf` file. Start rootd with "`-c filename`" to choose a different configuration file. All options are in the "`[rootd]`" section. Certificates and keys may be in either DER or PEM format.

bpki-ta

Where rootd should look for the BPKI trust anchor. All BPKI certificate verification within rootd traces back to this trust anchor. Don't change this unless you really know what you are doing.

```
bpki-ta = ${myrpki::bpki_servers_directory}/ca.cer
```

rootd-bpki-crl

BPKI CRL. Don't change this unless you really know what you are doing.

```
rootd-bpki-crl = ${myrpki::bpki_servers_directory}/ca.crl
```

rootd-bpki-cert

rootd's own BPKI EE certificate. Don't change this unless you really know what you are doing.

```
rootd-bpki-cert = ${myrpki::bpki_servers_directory}/rootd.cer
```

rootd-bpki-key

Private key corresponding to rootd's own BPKI EE certificate. Don't change this unless you really know what you are doing.

```
rootd-bpki-key = ${myrpki::bpki_servers_directory}/rootd.key
```

child-bpki-cert

BPKI certificate for rootd's one and only up-down child (RPKI engine to which rootd issues an RPKI certificate). Don't change this unless you really know what you are doing.

```
child-bpki-cert = ${myrpki::bpki_servers_directory}/child.cer
```

server-host

Server host on which rootd should listen.

```
server-host = ${myrpki::rootd_server_host}
```

server-port

Server port on which rootd should listen.

```
server-port = ${myrpki::rootd_server_port}
```

rpki-root-dir

Where rootd should write its output. Yes, rootd should be using pubd instead of publishing directly, but it doesn't. This needs to match pubd's configuration.

```
rpki-root-dir = ${myrpki::publication_base_directory}
```

rpki-base-uri

rsync URI corresponding to directory containing rootd's outputs.

```
rpki-base-uri = rsync://${myrpki::publication_rsync_server}/${myrpki::publication_rsync_module}
```

rpki-root-cert-uri

rsync URI for rootd's root (self-signed) RPKI certificate.

```
rpki-root-cert-uri = rsync://${myrpki::publication_rsync_server}/${myrpki::publication_root_mod
```

rpki-root-key

Private key corresponding to rootd's root RPKI certificate.

```
rpki-root-key = ${myrpki::bpki_servers_directory}/root.key
```

rpki-root-cert

Filename (as opposed to rsync URI) of rootd's root RPKI certificate.

```
rpki-root-cert = ${myrpki::publication_root_cert_directory}/root.cer
```

rpki-subject-pkcs10

Where rootd should stash a copy of the PKCS #10 request it gets from its one (and only) child

```
rpki-subject-pkcs10 = ${myrpki::bpki_servers_directory}/rootd.subject.pkcs10
```

rpki-subject-lifetime

Lifetime of the one and only RPKI certificate rootd issues.

```
rpki-subject-lifetime = 30d
```

rpki-root-crl

Filename (relative to rootd-base-uri and rpki-root-dir) of the CRL for rootd's root RPKI certificate.

```
rpki-root-crl = root.crl
```

rpki-root-manifest

Filename (relative to rootd-base-uri and rpki-root-dir) of the manifest for rootd's root RPKI certificate.

```
rpki-root-manifest = root.mft
```

rpki-class-name

Up-down protocol class name for RPKI certificate rootd issues to its one (and only) child.

```
rpki-class-name = ${myrpki::handle}
```

rpki-subject-cert

Filename (relative to rootd-base-uri and rpki-root-dir) of the one (and only) RPKI certificate rootd issues.

```
rpki-subject-cert = ${myrpki::handle}.cer
```

Creating an RPKI Root Certificate

rootd does not create RPKI root certificates automatically. If you're running your own root, you have to do this yourself. The usual method of doing this is to use the OpenSSL command line tool. The exact details will depend on which resources you need to put in the root certificate, the URIs for your publication server, and so forth, but the general form looks something like this:

```
[req]
default_bits            = 2048
default_md              = sha256
distinguished_name      = req_dn
prompt                  = no
encrypt_key             = no

[req_dn]
CN                      = Testbed RPKI root certificate

[x509v3_extensions]
basicConstraints        = critical,CA:true
subjectKeyIdentifier    = hash
keyUsage                = critical,keyCertSign,cRLSign
subjectInfoAccess       = @sia
certificatePolicies     = critical,1.3.6.1.5.5.7.14.2
sbgp-autonomousSysNum   = critical,@rfc3779_asns
sbgp-ipAddrBlock        = critical,@rfc3997_addrs

[sia]
1.3.6.1.5.5.7.48.5;URI  = rsync://example.org/rpki/root/
1.3.6.1.5.5.7.48.10;URI = rsync://example.org/rpki/root/root.mft

[rfc3779_asns]
AS.0 = 64496-64511
AS.1 = 65536-65551

[rfc3997_addrs]
IPv4.0 = 192.0.2.0/24
IPv4.1 = 198.51.100.0/24
IPv4.2 = 203.0.113.0/24
IPv6.0 = 2001:0DB8::/32
```

Assuming you save this configuration in a file `root.conf`, you can use it to generate a root certificate as follows:

```
openssl genrsa -out root.key 2048

openssl req                     \
        -new                    \
        -x509                   \
        -config root.conf       \
        -key    root.key        \
        -out    root.cer        \
        -outform    DER         \
        -days       1825        \
        -set_serial 1           \
        -extensions x509v3_extensions
```

You may want to shorten the five year expiration time (1825 days), which is a bit long. It is a root certificate, so a long expiration is not unusual.

When regenerating a certificate using the same key, just skip the `openssl genrsa` step above.

You must copy the generated root.cer to the publication directory as defined in rpki.conf:

```
rpki-root-cert          = ${myrpki::publication_base_directory}/root.cer
```

You must place the generated root.key in a safe location where it is readable by rootd but not accessible to the outside world, then you need to tell rootd where to find it by setting the appropriate variable in rpki.conf. The directory where the daemons keep their BPKI keys and certificates should be suitable for this:

```
rpki-root-key           = ${myrpki::bpki_servers_directory}/root.key
```

To create a TAL format trust anchor locator use the `make-tal.sh` script from `$top/rcynic`:

```
$top/rcynic/make-tal.sh  rsync://example.org/rpki/root/root.cer  root.cer
```

Converting an existing RSA key from PKCS #8 format

If you previously generated a certificate using `openssl req` with the `-newkey` option and are having difficulty getting `rootd` to accept the resulting private key, the problem may be that OpenSSL saved the private key file in PKCS #8 format. OpenSSL's behavior changed here, the `-newkey` option saved the key in PKCS #1 format, but newer versions use PKCS #8. While PKCS #8 is indeed likely an improvement, the change confuses some programs, including versions of `rootd` from before we discovered this problem.

If you think this might be your problem, you can convert the existing private key to PKCS #1 format with a script like this:

```
if ! openssl rsa -in root.key -out root.key.new
then
    echo Conversion failed
    rm root.key.new
elif cmp -s root.key root.key.new
    echo No change
    rm root.key.new
else
    echo Converted
    mv root.key.new root.key
fi
```

[web_portal] section

Glue to allow the Django application to pull user configuration from this file rather than directly editing settings.py.

sql-database

SQL database name the web portal should use.

```
sql-database = ${myrpki::irdbd_sql_database}
```

sql-username

SQL user name the web portal should use.

```
sql-username = ${myrpki::irdbd_sql_username}
```

sql-password

SQL password the web portal should use.

```
sql-password = ${myrpki::irdbd_sql_password}
```

secret-key

Site-specific secret key for Django.

No default value.

allowed-hosts

Name of virtual host that runs the Django GUI, if this is not the same as the system hostname. Django's security code wants to know the name of the virtual host on which Django is running, and will fail when it thinks it's running on a disallowed host.

If you get an error like "Invalid HTTP_HOST header (you may need to set ALLOWED_HOSTS)", you will need to set this option.

No default value.

[autoconf] section

rpki-confgen --autoconf records the current autoconf settings here, so that other options can refer to them. The section name "autoconf" is magic, don't change it.

bindir

Usually /usr/bin or /usr/local/bin.

No default value.

datarootdir

Usually /usr/share or /usr/local/share.

No default value.

sbindir

Usually /usr/sbin or /usr/local/sbin.

No default value.

sysconfdir

Usually /etc or /usr/local/etc.

No default value.

smoketest.yaml

smoketest test description file is named smoketest.yaml by default. Run smoketest with "-y filename" to change it. The YAML file contains multiple YAML "documents". The first document describes the initial test layout and resource allocations, subsequent documents describe modifications to the initial allocations and other parameters. Resources listed in the initial layout are aggregated automatically, so that a node in the resource hierarchy automatically receives the resources it needs to issue whatever its children are listed as holding. Actions in the subsequent documents are modifications to the current resource set, modifications to validity dates or other non-resource parameters, or special commands like "sleep".

Here's an example of current usage:

```
name:           Alice
valid_for:      2d
sia_base:       "rsync://alice.example/rpki/"
kids:
  - name: Bob
 kids:
   - name: Carol
     ipv4: 192.0.2.1-192.0.2.33
     asn:  64533
---
- name: Carol
  valid_add:   10
---
- name: Carol
  add_as: 33
  valid_add:   2d
---
- name: Carol
  valid_sub:   2d
---
- name: Carol
  valid_for:   10d
```

This specifies an initial layout consisting of an RPKI engine named "Alice", with one child "Bob", which in turn has one child "Carol". Carol has a set of assigned resources, and all resources in the system are initially set to be valid for two days from the time at which the test is started. The first subsequent document adds ten seconds to the validity interval for Carol's resources and makes no other modifications. The second subsequent document grants Carol additional resources and adds another two days to the validity interval for Carol's resources. The next document subtracts two days from the validity interval for Carol's resources. The final document sets the validity interval for Carol's resources to ten days.

Operators in subsequent (update) documents:

add_as::

Add ASN resources.

add_v4::

Add IPv4 resources.

add_v6::

Add IPv6 resources.

sub_as::

Subtract ASN resources.

sub_v4::

Subtract IPv4 resources.

sub_v6::

Subtract IPv6 resources.

valid_until::

Set an absolute expiration date.

valid_for::

Set a relative expiration date.

valid_add::

Add to validity interval.

valid_sub::

Subtract from validity interval.

sleep [interval]::

Sleep for specified interval, or until smoketest receives a SIGALRM signal.

shell cmd...::

Pass rest of line verbatim to /bin/sh and block until the shell returns.

Absolute timestamps should be in the form shown (UTC timestamp format as used in XML).

Intervals (`valid_add`, `valid_sub`, `valid_for`, `sleep`) are either integers, in which case they're interpreted as seconds, or are a string of the form "wD xH yM zS" where w, x, y, and z are integers and D, H, M, and S indicate days, hours, minutes, and seconds. In the latter case all of the fields are optional, but at least one must be specified. For example, "3D4H" means "three days plus four hours".

Running rpkid or pubd on a different server

The default configuration runs rpkid, pubd (if enabled) and the back end code all on the same server. For many purposes, this is fine, but in some cases you might want to split these functions up among different servers.

As noted briefly above, there are two separate sets of rpki.conf options which control the necessary behavior: the `run_*` options and the `start_*` options. The latter are usually tied to the former, but you can set them separately, and they control slightly different things: the `run_*` options control whether the back end code attempts to manage the servers in question, while the `start_*` flags control whether the startup scripts should start the servers in question.

Here's a guideline to how to set up the servers on different machines. For purposes of this description we'll assume that you're running both rpkid and pubd, and that you want rpkid and pubd each on their own server, separate from the back end code. We'll call these servers rpkid.example.org, pubd.example.org, and backend.example.org.

Most of the configuration is the same as in the normal case, but there are a few extra steps. The following supplements but does not replace the normal instructions.

WARNING: These setup directions have not (yet) been tested extensively.

- Create rpki.conf as usual on backend.example.org, but pay particular attention to the settings of `rpkid_server_host`, `irbe_server_host`, and `pubd_server_host`: these should name rpkid.example.org, backend.example.org, and pubd.example.org, respectively.

- This example assumes that you're running pubd, so make sure that both `run_rpkid` and `run_pubd` are enabled in rpki.conf.

- Copy the rpki.conf to the other machines, and customize each copy to that machine's role:
    - `start_rpkid` should be enabled on rpkid.example.org and disabled on the others.
    - `start_pubd` should be enabled on pubd.example.org and disabled on the others.
    - `start_irdbd` should be enabled on backend.example.org and disabled on the others.

- Make sure that you set up SQL databases on all three servers; the `rpki-sql-setup` script should do the right thing in each case based on the setting of the `start_*` options.

- Run "rpkic initialize" on the back end host. This will create the BPKI and write out all of the necessary keys and certificates.

- "rpkic initialize" should have created the BPKI files (.cer, .key, and .crl files for the several servers). Copy the .cer and .crl files to the pubd and rpkid hosts, along with the appropriate private key: rpkid.example.org should get a copy of the rpkid.key file but not the pubd.key file, while pubd.example.org should get a copy of the pubd.key file but not the rpkid.key file.

- Run `rpki-start-servers` on each of the three hosts when it's time to start the servers.

- Do the usual setup dance, but keep in mind that the the back end controlling all of these servers lives on backend.example.org, so that's where you issue the rpkic or GUI commands to manage them. rpkic and the GUI both know how to talk to rpkid and pubd over the network, so managing them remotely is fine.

RPKI Engine MySQL Setup

You need to install MySQL and set up the relevant databases before starting rpkid, irdbd, or pubd.

See the Installation Guide for details on where to download MySQL and find documentation on installing it.

See the Configuration Guide for details on the configuration file settings the daemons will use to find and authenticate themselves to their respective databases.

Before you can (usefully) start any of the daemons, you will need to set up the MySQL databases they use. You can do this by hand, or you can use the `rpki-sql-setup` script, which prompts you for your MySQL root password then attempts to do everything else automatically using values from rpki.conf.

Using the script is simple:

```
$ rpki-sql-setup
Please enter your MySQL root password:
```

The script should tell you what databases it creates. You can use the -v option if you want to see more details about what it's doing.

If you'd prefer to do the SQL setup manually, perhaps because you have valuable data in other MySQL databases and you don't want to trust some random setup script with your MySQL root password, you'll need to use the MySQL command line tool, as follows:

```
$ mysql -u root -p

mysql> CREATE DATABASE irdb_database;
mysql> GRANT all ON irdb_database.* TO irdb_user@localhost IDENTIFIED BY 'irdb_password';
mysql> CREATE DATABASE rpki_database;
mysql> GRANT all ON rpki_database.* TO rpki_user@localhost IDENTIFIED BY 'rpki_password';
mysql> USE rpki_database;
mysql> SOURCE $top/rpkid/rpkid.sql;
mysql> COMMIT;
mysql> quit
```

where `irdb_database`, `irdb_user`, `irdb_password`, `rpki_database`, `rpki_user`, and `rpki_password` match the values you used in your configuration file.

If you are running pubd and are doing manual SQL setup, you'll also have to do:

```
$ mysql -u root -p
mysql> CREATE DATABASE pubd_database;
mysql> GRANT all ON pubd_database.* TO pubd_user@localhost IDENTIFIED BY 'pubd_password';
mysql> USE pubd_database;
mysql> SOURCE $top/rpkid/pubd.sql;
mysql> COMMIT;
mysql> quit
```

where `pubd_database`, `pubd_user` `pubd_password` match the values you used in your configuration file.

Once you've finished configuring MySQL, the next thing you should read is the instructions for the user interface tools.

RPKI CA Out-Of-Band Setup Protocol

Not documented yet. Eventually this will be a readable explanation of the out-of-band setup protocol.

The CA user interface tools

The design of rpkid and pubd assumes that certain tasks can be thrown over the wall to the registry's back end operation. This was a deliberate design decision to allow rpkid and pubd to remain independent of existing database schema, business PKIs, and so forth that a registry might already have. All very nice, but it leaves someone who just wants to test the tools or who has no existing back end with a fairly large programming project. The user interface tools attempt to fill that gap. Together with irdbd, these tools consitute the "IR back-end" (IRBE) programs.

rpkic is a command line interface to the the IRBE. The web interface is a Django-based graphical user interface to the IRBE. The two user interfaces are built on top of the same libraries, and can be used fairly interchangeably. Most users will probably prefer the GUI, but the command line interface may be useful for scripted control, for testing, or for environments where running a web server is not practical.

A large registry which already has its own back-end system might want to roll their own replacement for the entire IRBE package. The tools are designed to allow this.

The user interface tools support two broad classes of operations:

1. Relationship management: setting up relationships between RPKI parent and child entities and between publication repositories and their clients. This is primarily about exchange of BPKI keys with other entities and learning the service URLs at which rpkid should contact other servers. We refer to this as the "setup phase".
2. Operation of rpkid once relationships have been set up: issuing ROAs, assigning resources to children, and so forth. We refer to this as the "data maintenance" phase.

During setup phase, the tools generate and processes small XML messages, which they expects the user to ship to and from its parents, children, etc via some out-of-band means (email, perhaps with PGP signatures, USB stick, we really don't care). During data maintenance phase, the tools control the operation of rpkid and pubd.

While the normal way to enter data during maintenance phase is by filling out web forms, there's also a file-based format which can be used to upload and download data from the GUI; the command line tool uses the same file format. These files are simple whitespace-delimited text files (".csv files" -- the name is historical, at one point these were parsed and generated using the Python "csv" library, and the name stuck). The intent is that these be very simple files that are easy to parse or to generate as a dump from relational database, spreadsheet, awk script, whatever works in your environment.

As with rpkid and pubd, the user interface tools use a configuration file, which defaults to the same system-wide rpki.conf file as the other programs.

Overview of setup phase

While the specific commands one uses differ depending on whether you are using the command line tool or the GUI, the basic operations during setup phase are the same:

1. If you haven't already done so, install the software, create the rpki.conf for your installation, and set up the MySQL database.
2. If you haven't already done so, create the initial BPKI database for your installation by running the "rpkic initialize" command. This will also create a BPKI identity for the handle specified in your rpki.conf file. BPKI initialization is tied to creation of the initial BPKI identity for historical reasons. These operations probably ought to be handled by separate commands, and may be in the future.
3. If you haven't already done so, start the servers, using the `rpki-start-servers` script.

4. Send a copy of the XML identity file written out by "rpkic initialize" to each of your parents, somehow (email, USB stick, carrier pigeon, we don't care). The XML identity file will have a filename like ./${handle}.identity.xml where "." is the directory in which you ran rpkic and ${handle} is the handle set in your rpki.conf file or selected with rpkic's `select_identity` command. This XML identity file tells each of your parents what you call yourself, and supplies each parent with a trust anchor for your resource-holding BPKI.
5. Each of your parents configures you as a child, using the XML identity file you supplied as input. This registers your data with the parent, including BPKI cross-registration, and generates a return message containing your parent's BPKI trust anchors, a service URL for contacting your parent via the "up-down" protocol, and (usually) either an offer of publication service (if your parent operates a repository) or a referral from your parent to whatever publication service your parent does use. Referrals include a CMS-signed authorization token that the repository operator can use to determine that your parent has given you permission to home underneath your parent in the publication tree.
6. Each of your parents sends (...) back the response XML file generated by the "configure_child" command.
7. You feed the response message you just got into the IRBE using rpkic's "configure_parent" command. This registers the parent's information in your database, handles BPKI cross-certification of your parent., and processes the repository offer or referral to generate a publication request message.
8. You send (...) the publication request message to the repository. The `contact_info` element in the request message should (in theory) provide some clue as to where you should send this.
9. The repository operator processes your request using rpkic's "configure_publication_client" command. This registers your information, including BPKI cross-certification, and generates a response message containing the repository's BPKI trust anchor and service URL.
10. Repository operator sends (...) the publication confirmation message back to you.
11. You process the publication confirmation message using rpkic's "configure_repository" command.

At this point you should, in theory, have established relationships, exchanged trust anchors, and obtained service URLs from all of your parents and repositories.

Troubleshooting

If you run into trouble setting up this package, the first thing to do is categorize the kind of trouble you are having. If you've gotten far enough to be running the daemons, check their log files. If you're seeing Python exceptions, read the error messages. If you're getting CMS errors, check to make sure that you're using all the right BPKI certificates and service contact URLs.

If you've completed the steps above, everything appears to have gone OK, but nothing seems to be happening, the first thing to do is check the logs to confirm that nothing is actively broken. rpkid's log should include messages telling you when it starts and finishes its internal "cron" cycle. It can take several cron cycles for resources to work their way down from your parent into a full set of certificates and ROAs, so have a little patience. rpkid's log should also include messages showing every time it contacts its parent(s) or attempts to publish anything.

rcynic in fully verbose mode provides a fairly detailed explanation of what it's doing and why objects that fail have failed.

You can use rsync (sic) to examine the contents of a publication repository one directory at a time, without attempting validation, by running rsync with just the URI of the directory on its command line:

```
$ rsync rsync://rpki.example.org/where/ever/
```

If you need to examine RPKI objects in detail, you have a few options:

- The RPKI utilities include several programs for dumping RPKI-specific objects in text form.

- The OpenSSL command line program can also be useful for examining and manipulating certificates and CMS messages, although the syntax of some of the commands can be a bit obscure.

- Peter Gutmann's excellent dumpasn1 program may be useful if you are desperate enough that you need to examine raw ASN.1 objects.

The rpkic tool

rpkic is a command line interface to rpkid and pubd. It implements largely the same functionality as the web interface. In most cases you will want to use the web interface for normal operation, but rpkic is available if you need it.

rpkic can be run either in an interactive mode or by passing a single command on the command line when starting the program; the former mode is intended to be somewhat human-friendly, the latter mode is useful in scripting, cron jobs, and automated testing.

Some rpkic commands write out data files, usually in the current directory.

rpkic uses the same system-wide rpki.conf file as the other CA tools as its default configuration file.

rpkic includes a "help" command which provides inline help for its several commands.

Selecting an identity

The *handle* variable in rpki.conf specifies the handle of the default identity for an rpkic command, but this is just the default. rpkid can host an arbitrary number of identities, and rpkic has to be able to control all of them.

When running rpkic interactively, use rpkic's "select_identity" command to set the current identity handle.

When running rpkic with a single command on the command line, use the "-i" (or "--identity") option to set the current identity handle.

rpkic in setup phase

See the introduction to the user interfaces for an overview of how setup phase works. The general structure of the setup phase in rpkic is as described there, but here we provide the specific commands involved. The following assumes that you have already installed the software and started the servers.

- The rpkic "initialize" command writes out an "identity.xml" file in addition to all of its other tasks.
- A parent who is using rpkic runs the "configure_child" command to configure the child, giving this command the identity.xml file the child supplied as input. configure_child will write out a response XML file, which the parent sends back to the child.
- A child who is running rpkic runs the "configure_parent" command to process the parent's response, giving it the XML file sent back by the parent as input to this command. configure_parent will write out a publication request XML file, which the child sents to the repository operator.
- A repository operator who is using rpkic runs the "configure_publication_client" command to process a client's publication request. configure_publication_client generates a confirmation XML message which the repository operator sends back to the client.
- A publication client who is using rpkic runs the "configure_repository" command to process the repository's response.

rpkic in data maintenance phase

rpkic uses whitespace-delimited text files (called ".csv files", for historical reasons) to control issuance of addresses and autonomous sequence numbers to children, and to control issuance of ROAs. See the

"load_asns", "load_prefixes", and "load_roa_requests" commands.

Maintaining child validity data

All resources issued to child entities are tagged with a validity date. If not updated, these resources will eventually expire. rpkic includes two commands for updating these validity dates:

- "renew_child" updates the validity date for a specific child.
- "renew_all_children" updates the validity date for all children.

BPKI maintenance

Certificates and CRLs in the BPKI have expiration dates and netUpdate dates, so they need to be maintained. Failure to maintain these will eventually cause the CA software to grind to a halt, as expired certificates will cause CMS validation failures.

rpkic's "update_bpki" command takes care of this. Usually one will want to run this periodically (perhaps once per month), under cron.

Forcing synchronization

Most rpkic commands synchronize the back end database with the daemons automatically, so in general it should not be necessary to synchronize manually. However, since these are separate databases, it is theoretically possible for them to get out of synch, perhaps because something crashed at exactly the wrong time.

rpkic's "synchronize" command runs a synchronization cycle with rpkid (if `run_rpkic` is set) and pubd (if `run_pubd` is set).

Installing and Configuring

- GUI/Installing for new installs
- GUI/Upgrading for upgrading from a previous install
- GUI/Configuring
- GUI/UserModel for instructions on managing users

Using the GUI

GUI Examples

Logging in to the GUI

The Dashboard - Let's Make a ROA

ROA List Currently Empty, So Let's Create One

Choose an AS and Prefix - Let MaxLen? Default

What Will the Consequences Be? - Confirm OK

Now We Can See ROAs - Let's Look at Routes

Real Effect on Routing Table

Ghostbusters etc. are Similar

Installing the Web Portal for the First Time

This page documents how to install the web portal software. **If you have previously installed the software**, see doc/RPKI/CA/UI/GUI/Upgrading for instructions.

Prerequisites

This page assumes that you have already followed the steps to install the CA software (see doc/RPKI/Installation)

This page assumes that you have already created `/etc/rpki.conf` (see doc/RPKI/CA/Configuration)

Create Database Tables

This step creates the tables used by the web portal in the database. Run the following commands in the shell (you do not need to be *root*, just have permission to read `/etc/rpki.conf`):

```
rpki-manage syncdb --noinput
rpki-manage migrate
```

Note that at the end of the `syncdb` output you will see the following message:

```
Not synced (use migrations):
 - rpki.gui.app
(use ./manage.py migrate to migrate these)
```

You should **ignore the message about running ./manage.py** since that script does not exist in our setup (we use `rpki-manage` instead`).

Next Step

See doc/RPKI/CA/UI/GUI/Configuring

Upgrading from a Previous Version

- See wiki:doc/RPKI/CA/UI/GUI/Upgrading/BeforeMigration for the special situation where you are upgrading from a release **prior to database migration support being added**.

This page describes the steps you must take if you upgrading from a previous version of the software that is already installed on the system. If you are installing for the first time see doc/RPKI/CA/UI/GUI/Installing.

Run the following commands at a shell prompt. Note that you do not need run these as the *root* user, any user with permission to read `/etc/rpki.conf` is sufficient.

```
rpki-manage syncdb
rpki-manage migrate
```

Note that at the end of the `syncdb` output you will see the following message:

```
Not synced (use migrations):
 - rpki.gui.app
(use ./manage.py migrate to migrate these)
```

You should **ignore the message about running ./manage.py** since that script does not exist in our setup (we use `rpki-manage` instead`).

Restart Apache

In order to cause Apache to reload the web portal software using the newly installed software, it must be restarted. Execute the following command as *root* in a shell:

```
apachectl restart
```

Next Step

See doc/RPKI/CA/UI/GUI/Configuring

Upgrading from a Previous Release without Migration Support

This page documents the steps required to upgrade the web portal when you have a previous version of the software install **prior to migration support via Django South**. Note that this is a special case and will not apply to most situations (see doc/RPKI/CA/UI/GUI/Upgrading for the normal upgrade path). If you have already performed the steps on this page previously, then it does not apply to your situation.

If you are unsure whether or not you have previously run this command, you can verify with the following command:

```
$ rpki-manage migrate --list

 app
  (*) 0001_initial
  (*) 0002_auto__add_field_resourcecert_conf
  (*) 0003_set_conf_from_parent
  (*) 0004_auto__chg_field_resourcecert_conf
  (*) 0005_auto__chg_field_resourcecert_parent
  ( ) 0006_add_conf_acl
  ( ) 0007_default_acls
```

The migrations are an ordered list. The presence of the asterisk `(*)` indicates that the migration has already been performed. `( )` indicates that the specific migration has not yet been applied. In the example above, migrations 0001 through 0005 have been applied, but 0006 and 0007 have not.

Sync databases

Execute the following command in a shell. Note that you do not need to be the *root* user, any user with permission to read `/etc/rpki.conf` is sufficient.

```
$ rpki-manage syncdb
```

Note that at the end of the `syncdb` output you will see the following message:

```
Not synced (use migrations):
 - rpki.gui.app
(use ./manage.py migrate to migrate these)
```

You should **ignore the message about running ./manage.py** since that script does not exist in our setup.

Initial Database Migration

For a completely new install, there will not be any existing tables in the database, and the `rpki-manage migrate` command will create them. However, in the special situation where you are upgrading from a previous release prior to the migration support being added, you will already have the tables created, which will case the initial migration to fail. In order to work around this problem, we have to tell the migration that the initial step has already been performed. This is accomplished via the use the `--fake` command line argument:

```
$ rpki-manage migrate app 0001 --fake
```

Note that this step doesn't actually modify the database, other than to record that the migration has already taken place.

Database Migration

Now bring your database up to date with the current release:

```
$ rpki-manage migrate
```

From this point forward you will follow the steps in doc/RPKI/CA/UI/GUI/Upgrading each time you upgrade.

Restart Apache

In order to make Apache use the new version of the software, it must be restarted:

```
$ apachectl restart
```

Configuring the Web Portal

Also see doc/RPKI/CA/Configuration for documentation on the `/etc/rpki.conf` configuration file.

Creating Users

See doc/RPKI/CA/UI/GUI/UserModel

Configuring Apache

In order to use the web portal, Apache must be installed and configured to serve the application. See doc/RPKI/CA/UI/GUI/Configuring/Apache.

Error Notifications via Email

If an exception is generated while the web portal is processing a request, by default will be logged to the apache log file, and an email will be set to `root@localhost`. If you wish to change where email is sent, you can edit `/etc/rpki/local_settings.py` and add the following lines:

```
ADMINS = (('YOUR NAME', 'YOUR EMAIL ADDRESS'),)
```

For example,

```
ADMINS = (('Joe User', 'joe@example.com'),)
```

Cron Jobs

The web portal makes use of some external data sources to display the validation status of routing entries. Therefore, it is necessary to run some background jobs periodically to refresh this data. The web portal software makes use of the `cron` facility present in POSIX operating systems to perform these tasks.

Importing Routing Table Snapshot

In order for the web portal to display the validation status of routes covered by a resource holder's RPKI certificates, it needs a source of the currently announced global routing table. The web portal includes a script which can parse the output of the RouteViews full snapshot (**warning**: links to very large file!).

When the software is installed, there will be a `/usr/local/sbin/rpkigui-import-routes` script that should be invoked periodically. Routeviews.org updates the snapshot every two hours, so it does not make sense to run it more frequently than two hours. How often to run it depends on how often the routes you are interested in are changing.

Create an entry in root's crontab such as

```
30  */2 *   *   *      /usr/local/sbin/rpkigui-import-routes
```

Importing ROAs

If you want the GUI's "routes" page to see ROAs when you click those buttons, you will need to run rcynic. see the instructions for setting up rcynic.

This data is imported by the `rcynic-cron` script. If you have not already set up that cron job, you should do so now. Note that by default, rcynic-cron is run once an hour. What this means is that the *routes* view in

the GUI will **not** immediately update as you create/destroy ROAs. You may wish to run `rcynic-cron` more frequently, or configure `rcynic.conf` to only include the TAL that is the root of your resources, and run the script more frequently (perhaps every 2-5 minutes).

If you are running rootd, you may want to run with only your local trust anchor. In this case, to have the GUI be fairly responsive to changes, you may want to run the rcynic often. In this case, you may want to look at the value of **jitter** in rcynic.conf.

Expiration Checking

The web portal can notify users when it detects that RPKI certificates will expire in the near future. Run the following script as a cron job, perhaps once a night:

```
/usr/local/sbin/rpkigui-check-expired
```

By default it will warn of expiration 14 days in advance, but this may be changed by using the `-t` command line option and specifying how many days in advance to check.

Apache Configuration

This page documents how to configure Apache to server the web portal application.

During the software install process, `/usr/local/etc/rpki/apache.conf` is created, which needs to be included from the apache configuration inside of a `VirtualHost` section.

Note that the web portal application **requires TLS** to be enabled for the `VirtualHost` it is configured in, otherwise it will fail to operate.

Requirements

- Apache 2.2 or later
- mod_ssl
- mod_wsgi 3 or later

Debian & Ubuntu

First, you need to install `apache` and enable SSL. Run the following commands in a shell as **root**:

```
apt-get install apache2 libapache2-mod-wsgi
a2enmod ssl
a2ensite default-ssl
```

Edit `/etc/apache2/sites-enabled/default-ssl` and place the following line inside the `<VirtualHost>` section:

```
Include /usr/local/etc/rpki/apache.conf
```

Now restart `apache`:

```
service apache2 restart
```

FreeBSD

Now configure apache, using `/usr/local/etc/rpki/apache.conf`, e.g.

```
$ cp apache.conf /usr/local/etc/apache22/Includes/rpki.conf
```

Restart apache

```
$ apachectl restart
```

Running the web portal as a different user (optional)

By default, the web portal is run in embedded mode in mod_wsgi, which means it runs inside the apache process. However, you can make the web portal run in daemon mode as a different user using mod_wsgi.

```
$ ./configure --enable-wsgi-daemon-mode[=user[:group]]
```

Where `user` is the optional user to run the web portal as, and `group` is the optional group to run the web portal as. If `user` is not specified, it will run in a separate process but the same user as apache is configured to run.

Note that when run in daemon mode, a unix domain socket will be created in the same directory as the apache log files. If the user you have specified to run the web portal as does not have permission to read a file in that directory, the web interface will return a **500 Internal Server Error** and you will see a **permission denied** error in your apache logs. The solution to this is to use the `WSGISocketPrefix` apache configuration directive to specify an alternative location, such as:

```
WSGISocketPrefix /var/run/wsgi
```

Note that this directive **must not** be placed inside of the `VirtualHost` section. It **must** be located at the global scope.

see http://code.google.com/p/modwsgi/wiki/ConfigurationDirectives#WSGISocketPrefix for more information.

Verify the Web Portal is Working

Navigate to https://YOURHOST/rpki/ and you should see the login page for the web portal.

Enter the superuser and password in login form (see doc/RPKI/CA/UI/GUI/UserModel if you haven't yet created a superuser). If you've only done the above bootstrap, there will only be a single handle to manage, so the GUI will automatically bring you to the dashboard for that handle.

RPKI Web Portal User Model

Roles

The web portal uses a model where users are distinct from resource holders.

Users

A user is an entity that is granted permission to utilize the web portal. Each user account has an associated password that is used to log in to the web portal.

The web portal maintains an access control list that specifies which resource holders the user is allowed to manage. If a user is authorized to manage more than a single resource holder, the user will be presented with a list of the resource holders upon login.

Database tables: `irdbd.auth_user` and `irdbd.app_confacl`

Changing User Passwords

The password for a user may be changed via the web portal, or on the command line:

```
$ rpki-manage changepassword <USER>
```

Superuser

A user account with the superuser bit set has the special capability that it may assume the role of any resource holder managed by the local RPKI service. Superusers are created via the command line interface:

```
$ rpki-manage createsuperuser
```

Creating user accounts

When logged into the web portal with a #superuser account, select the **web users** link in the sidebar, and then click on the **create** button at the bottom of the page. You may optionally select one or more resource holders that this user is granted authorization to manage.

Note that creating a user does **not** create a matching #resource-holder. See creating resource holders.

Destroying user accounts

When logged into the web portal with a #superuser account, select the **web users** link in the sidebar, and then click on the **Delete** icon next to the user you wish to delete.

Note that this action does **not** remove any of the resource holders the user is granted authorization to manage.

Resource Holders

Resource holders are entities that have authority to manage a set of Internet number resources. When a user logs into the web portal, they select which resource holder role to assume. The user may choose to assume the role of a different resource holder by clicking on the **select identity** link in the sidebar.

The list of resource holders managed by the local RPKI service can be viewed with a #superuser account by clicking on the **resource holders** link in the sidebar of the web portal. From this page the super can manage the resource holders.

Database table: `irdbd.irdb_resourceholderca` (via `irdbd.app_conf` proxy model)

Creating resource holders

Note that creating a new resource holder does **not** create a user account. See #create-user.

GUI

When logged into the web portal with a #superuser account, select the **resource holders** link in the sidebar, and then click on the **create** button at the bottom of the page.

If the new resource holder is going to be a child of another resource holder hosted by the local RPKI service, you may optionally select the parent resource holder from the dropdown box, and the parent-child relationship will automatically be established when the new resource holder is created.

Additionally, one or more #users authorized to manage the new resource holder may be selected from the **Users** list on the creation form.

Command Line

You can also create resource holders on the command line:

```
$ rpkic -i <HANDLE> initialize
$ rpkic synchronize
```

where **HANDLE** is the name of new resource holder. Note that this new resource holder will initially only be allowed to be managed by #superuser accounts. You may wish to create a matching user account, but the name of the user need not be the same as the handle of the resource holder. Additionally, you can manage the list of users allowed to manage this resource holder via the web portal; click on the **Edit** icon next to the resource holder, and select the users you wish to grant permission to manage.

Destroying resource holders

Note that deleting a resource holder does **not** remove any user accounts.

GUI

When logged into the web portal with a #superuser account, select the **resource holders** link in the sidebar, and then click on the **delete** button next to the resource holder you wish to delete.

Command Line

Or you may use the command line interface:

```
$ rpkic -i <HANDLE> delete_self
$ rpkic synchronize
```

where *HANDLE* is the name of the resource holder you wish to destroy.

Modifying the User ACL

Each resource holder may be managed by one or more user accounts. The list of users authorized to assume the role of a particular resource holder may be changed in the web portal. When logged into the web portal with a #superuser account, select the **resource holders** link in the sidebar, and then click on the **Edit** icon next to the resource holder, and select the users you wish to grant permission to manage.

The Left-Right Protocol

The left-right protocol is really two separate client/server protocols over separate channels between the RPKI engine and the IR back end (IRBE). The IRBE is the client for one of the subprotocols, the RPKI engine is the client for the other.

Operations initiated by the IRBE

This part of the protcol uses a kind of message-passing. Each object that the RPKI engine knows about takes five messages: "create", "set", "get", "list", and "destroy". Actions which are not just data operations on objects are handled via an SNMP-like mechanism, as if they were fields to be set. For example, to generate a keypair one "sets" the "generate-keypair" field of a BSC object, even though there is no such field in the object itself as stored in SQL. This is a bit of a kludge, but the reason for doing it as if these were variables being set is to allow composite operations such as creating a BSC, populating all of its data fields, and generating a keypair, all as a single operation. With this model, that's trivial, otherwise it's at least two round trips.

Fields can be set in either "create" or "set" operations, the difference just being whether the object already exists. A "get" operation returns all visible fields of the object. A "list" operation returns a list containing what "get" would have returned on each of those objects.

Left-right protocol objects are encoded as signed CMS messages containing XML as eContent and using an eContentType OID of `id-ct-xml` (1.2.840.113549.1.9.16.1.28). These CMS messages are in turn passed as the data for HTTP POST operations, with an HTTP content type of "application/x-rpki" for both the POST data and the response data.

All operations allow an optional "tag" attribute which can be any alphanumeric token. The main purpose of the tag attribute is to allow batching of multiple requests into a single PDU.

self_obj <self/> object

A `<self/>` object represents one virtual RPKI engine. In simple cases where the RPKI engine operator operates the engine only on their own behalf, there will only be one `<self/>` object, representing the engine operator's organization, but in environments where the engine operator hosts other entities, there will be one `<self/>` object per hosted entity (probably including the engine operator's own organization, considered as a hosted customer of itself).

Some of the RPKI engine's configured parameters and data are shared by all hosted entities, but most are tied to a specific `<self/>` object. Data which are shared by all hosted entities are referred to as "per-engine" data, data which are specific to a particular `<self/>` object are "per-self" data.

Since all other RPKI engine objects refer to a `<self/>` object via a "self_handle" value, one must create a `<self/>` object before one can usefully configure any other left-right protocol objects.

Every `<self/>` object has a self_handle attribute, which must be specified for the "create", "set", "get", and "destroy" actions.

Payload data which can be configured in a `<self/>` object:

use_hsm:: (attribute)

> Whether to use a Hardware Signing Module. At present this option has no effect, as the implementation does not yet support HSMs.

crl_interval:: (attribute)

Positive integer representing the planned lifetime of an RPKI CRL for this <self/>, measured in seconds.

regen_margin:: (attribute)

Positive integer representing how long before expiration of an RPKI certificiate a new one should be generated, measured in seconds. At present this only affects the one-off EE certificates associated with ROAs. This parameter also controls how long before the nextUpdate time of CRL or manifest the CRL or manifest should be updated.

bpki_cert:: (element)

BPKI CA certificate for this <self/>. This is used as part of the certificate chain when validating incoming TLS and CMS messages, and should be the issuer of cross-certification BPKI certificates used in <repository/>, <parent/>, and <child/> objects. If the bpki_glue certificate is in use (below), the bpki_cert certificate should be issued by the bpki_glue certificate; otherwise, the bpki_cert certificate should be issued by the per-engine bpki_ta certificate.

bpki_glue:: (element)

Another BPKI CA certificate for this <self/>, usually not needed. Certain pathological cross-certification cases require a two-certificate chain due to issuer name conflicts. If used, the bpki_glue certificate should be the issuer of the bpki_cert certificate and should be issued by the per-engine bpki_ta certificate; if not needed, the bpki_glue certificate should be left unset.

Control attributes that can be set to "yes" to force actions:

rekey::

Start a key rollover for every RPKI CA associated with every <parent/> object associated with this <self/> object. This is the first phase of a key rollover operation.

revoke::

Revoke any remaining certificates for any expired key associated with any RPKI CA for any <parent/> object associated with this <self/> object. This is the second (cleanup) phase for a key rollover operation; it's separate from the first phase to leave time for new RPKI certificates to propegate and be installed.

reissue::

Not implemented, may be removed from protocol. Original theory was that this operation would force reissuance of any object with a changed key, but as that happens automatically as part of the key rollover mechanism this operation seems unnecessary.

run_now::

Force immediate processing for all tasks associated with this <self/> object that would ordinarily be performed under cron. Not currently implemented.

publish_world_now::

Force (re)publication of every publishable object for this <self/> object. Not currently implemented. Intended to aid in recovery if RPKI engine and publication engine somehow get out of sync.

<bsc/> object

The <bsc/> ("business signing context") object represents all the BPKI data needed to sign outgoing CMS messages. Various other objects include pointers to a <bsc/> object. Whether a particular <self/> uses only one <bsc/> or multiple is a configuration decision based on external requirements: the RPKI engine code doesn't care, it just cares that, for any object representing a relationship for which it must sign messages, there be a <bsc/> object that it can use to produce that signature.

Every <bsc/> object has a bsc_handle, which must be specified for the "create", "get", "set", and "destroy" actions. Every <bsc/> also has a self_handle attribute which indicates the <self/> object with which this <bsc/> object is associated.

Payload data which can be configured in a <isc/> object:

signing_cert:: (element)

BPKI certificate to use when generating a signature.

signing_cert_crl:: (element)

CRL which would list signing_cert if it had been revoked.

Control attributes that can be set to "yes" to force actions:

generate_keypair::

Generate a new BPKI keypair and return a PKCS #10 certificate request. The resulting certificate, once issued, should be configured as this <bsc/> object's signing_cert.

Additional attributes which may be specified when specifying "generate_keypair":

key_type::

Type of BPKI keypair to generate. "rsa" is both the default and, at the moment, the only allowed value.

hash_alg::

Cryptographic hash algorithm to use with this keypair. "sha256" is both the default and, at the moment, the only allowed value.

key_length::

Length in bits of the keypair to be generated. "2048" is both the default and, at the moment, the only allowed value.

Replies to "create" and "set" actions that specify "generate-keypair" include a <bsc_pkcs10/> element, as do replies to "get" and "list" actions for a <bsc/> object for which a "generate-keypair" command has been issued. The RPKI engine stores the PKCS #10 request, which allows the IRBE to reuse the request if and when it needs to reissue the corresponding BPKI signing certificate.

<parent/> object

The <parent/> object represents the RPKI engine's view of a particular parent of the current <self/> object in the up-down protocol. Due to the way that the resource hierarchy works, a given <self/> may obtain resources from multiple parents, but it will always have at least one; in the case of IANA or an RIR, the parent RPKI engine may be a trivial stub.

Every <parent/> object has a parent_handle, which must be specified for the "create", "get", "set", and "destroy" actions. Every <parent/> also has a self_handle attribute which indicates the <self/> object with which this <parent/> object is associated, a bsc_handle attribute indicating the <bsc/> object to be used when signing messages sent to this parent, and a repository_handle indicating the <repository/> object to be used when publishing issued by the certificate issued by this parent.

Payload data which can be configured in a <parent/> object:

peer_contact_uri:: (attribute)

> HTTP URI used to contact this parent.

sia_base:: (attribute)

> The leading portion of an rsync URI that the RPKI engine should use when composing the publication URI for objects issued by the RPKI certificate issued by this parent.

sender_name:: (attribute)

> Sender name to use in the up-down protocol when talking to this parent. The RPKI engine doesn't really care what this value is, but other implementations of the up-down protocol do care.

recipient_name:: (attribute)

> Recipient name to use in the up-down protocol when talking to this parent. The RPKI engine doesn't really care what this value is, but other implementations of the up-down protocol do care.

bpki_cms_cert:: (element)

> BPKI CMS CA certificate for this <parent/>. This is used as part of the certificate chain when validating incoming CMS messages If the bpki_cms_glue certificate is in use (below), the bpki_cms_cert certificate should be issued by the bpki_cms_glue certificate; otherwise, the bpki_cms_cert certificate should be issued by the bpki_cert certificate in the <self/> object.

bpki_cms_glue:: (element)

> Another BPKI CMS CA certificate for this <parent/>, usually not needed. Certain pathological cross-certification cases require a two-certificate chain due to issuer name conflicts. If used, the bpki_cms_glue certificate should be the issuer of the bpki_cms_cert certificate and should be issued by the bpki_cert certificate in the <self/> object; if not needed, the bpki_cms_glue certificate should be left unset.

Control attributes that can be set to "yes" to force actions:

rekey::

This is like the rekey command in the <self/> object, but limited to RPKI CAs under this parent.

reissue::

This is like the reissue command in the <self/> object, but limited to RPKI CAs under this parent.

revoke::

This is like the revoke command in the <self/> object, but limited to RPKI CAs under this parent.

<child/> object

The <child/> object represents the RPKI engine's view of particular child of the current <self/> in the up-down protocol.

Every <child/> object has a child_handle, which must be specified for the "create", "get", "set", and "destroy" actions. Every <child/> also has a self_handle attribute which indicates the <self/> object with which this <child/> object is associated.

Payload data which can be configured in a <child/> object:

bpki_cert:: (element)

BPKI CA certificate for this <child/>. This is used as part of the certificate chain when validating incoming TLS and CMS messages. If the bpki_glue certificate is in use (below), the bpki_cert certificate should be issued by the bpki_glue certificate; otherwise, the bpki_cert certificate should be issued by the bpki_cert certificate in the <self/> object.

bpki_glue:: (element)

Another BPKI CA certificate for this <child/>, usually not needed. Certain pathological cross-certification cases require a two-certificate chain due to issuer name conflicts. If used, the bpki_glue certificate should be the issuer of the bpki_cert certificate and should be issued by the bpki_cert certificate in the <self/> object; if not needed, the bpki_glue certificate should be left unset.

Control attributes that can be set to "yes" to force actions:

reissue::

Not implemented, may be removed from protocol.

<repository/> object

The <repository/> object represents the RPKI engine's view of a particular publication repository used by the current <self/> object.

Every <repository/> object has a repository_handle, which must be specified for the "create", "get", "set", and "destroy" actions. Every <repository/> also has a self_handle attribute which indicates the <self/> object with which this <repository/> object is associated.

Payload data which can be configured in a <repository/> object:

peer_contact_uri:: (attribute)

HTTP URI used to contact this repository.

bpki_cms_cert:: (element)

BPKI CMS CA certificate for this <repository/>. This is used as part of the certificate chain when validating incoming CMS messages If the bpki_cms_glue certificate is in use (below), the bpki_cms_cert certificate should be issued by the bpki_cms_glue certificate; otherwise, the bpki_cms_cert certificate should be issued by the bpki_cert certificate in the <self/> object.

bpki_cms_glue:: (element)

Another BPKI CMS CA certificate for this <repository/>, usually not needed. Certain pathological cross-certification cases require a two-certificate chain due to issuer name conflicts. If used, the bpki_cms_glue certificate should be the issuer of the bpki_cms_cert certificate and should be issued by the bpki_cert certificate in the <self/> object; if not needed, the bpki_cms_glue certificate should be left unset.

At present there are no control attributes for <repository/> objects.

<route_origin/> object

This section is out-of-date. The <route_origin/> object has been replaced by the <list_roa_requests/> IRDB query, but the documentation for that hasn't been written yet.

The <route_origin/> object is a kind of prototype for a ROA. It contains all the information needed to generate a ROA once the RPKI engine obtains the appropriate RPKI certificates from its parent(s).

Note that a <route_origin/> object represents a ROA to be generated on behalf of <self/>, not on behalf of a <child/>. Thus, a hosted entity that has no children but which does need to generate ROAs would be represented by a hosted <self/> with no <child/> objects but one or more <route_origin/> objects. While lumping ROA generation in with the other RPKI engine activities may seem a little odd at first, it's a natural consequence of the design requirement that the RPKI daemon never transmit private keys across the network in any form; given this requirement, the RPKI engine that holds the private keys for an RPKI certificate must also be the engine which generates any ROAs that derive from that RPKI certificate.

The precise content of the <route_origin/> has changed over time as the underlying ROA specification has changed. The current implementation as of this writing matches what we expect to see in draft-ietf-sidr-roa-format-03, once it is issued. In particular, note that the exactMatch boolean from the -02 draft has been replaced by the prefix and maxLength encoding used in the -03 draft.

Payload data which can be configured in a <route_origin/> object:

asn:: (attribute)

Autonomous System Number (ASN) to place in the generated ROA. A single ROA can only grant authorization to a single ASN; multiple ASNs require multiple ROAs, thus multiple <route_origin/> objects.

ipv4:: (attribute)

List of IPv4 prefix and maxLength values, see below for format.

ipv6:: (attribute)

List of IPv6 prefix and maxLength values, see below for format.

Control attributes that can be set to "yes" to force actions:

suppress_publication::

Not implemented, may be removed from protocol.

The lists of IPv4 and IPv6 prefix and maxLength values are represented as comma-separated text strings, with no whitespace permitted. Each entry in such a string represents a single prefix/maxLength pair.

ABNF for these address lists:

```
<ROAIPAddress> ::= <address> "/" <prefixlen> [ "-" <max_prefixlen> ]
                       ; Where <max_prefixlen> defaults to the same
                       ; value as <prefixlen>.

<ROAIPAddressList> ::= <ROAIPAddress> *( "," <ROAIPAddress> )
```

For example, `10.0.1.0/24-32,10.0.2.0/24`, which is a shorthand form of `10.0.1.0/24-32,10.0.2.0/24-24`.

Operations initiated by the RPKI engine

The left-right protocol also includes queries from the RPKI engine back to the IRDB. These queries do not follow the message-passing pattern used in the IRBE-initiated part of the protocol. Instead, there's a single query back to the IRDB, with a corresponding response. The CMS encoding are the same as in the rest of the protocol, but the BPKI certificates will be different as the back-queries and responses form a separate communication channel.

<list_resources/> messages

The `<list_resources/>` query and response allow the RPKI engine to ask the IRDB for information about resources assigned to a particular child. The query must include both a `self_handle` attribute naming the `<self/>` that is making the request and also a `child_handle` attribute naming the child that is the subject of the query. The query and response also allow an optional *tag* attribute of the same form used elsewhere in this protocol, to allow batching.

A `<list_resources/>` response includes the following attributes, along with the tag (if specified), `self_handle`, and `child_handle` copied from the request:

valid_until::

A timestamp indicating the date and time at which certificates generated by the RPKI engine for these data should expire. The timestamp is expressed as an XML `xsd:dateTime`, must be expressed in UTC, and must carry the "Z" suffix indicating UTC.

asn::

A list of autonomous sequence numbers, expressed as a comma-separated sequence of decimal integers with no whitespace.

ipv4::

A list of IPv4 address prefixes and ranges, expressed as a comma-separated list of prefixes and ranges with no whitespace. See below for format details.

ipv6::

A list of IPv6 address prefixes and ranges, expressed as a comma-separated list of prefixes and ranges with no whitespace. See below for format details.

Entries in a list of address prefixes and ranges can be either prefixes, which are written in the usual address/prefixlen notation, or ranges, which are expressed as a pair of addresses denoting the beginning and end of the range, written in ascending order separated by a single "-" character. This format is superficially similar to the format used for prefix and maxLength values in the `<route_origin/>` object, but the semantics differ: note in particular that `<route_origin/>` objects don't allow ranges, while `<list_resources/>` messages don't allow a maxLength specification.

Error handling

Error in this protocol are handled at two levels.

Since all messages in this protocol are conveyed over HTTP connections, basic errors are indicated via the HTTP response code. 4xx and 5xx responses indicate that something bad happened. Errors that make it impossible to decode a query or encode a response are handled in this way.

Where possible, errors will result in a `<report_error/>` message which takes the place of the expected protocol response message. `<report_error/>` messages are CMS-signed XML messages like the rest of this protocol, and thus can be archived to provide an audit trail.

`<report_error/>` messages only appear in replies, never in queries. The `<report_error/>` message can appear on either the "forward" (IRBE as client of RPKI engine) or "back" (RPKI engine as client of IRDB) communication channel.

The `<report_error/>` message includes an optional *tag* attribute to assist in matching the error with a particular query when using batching, and also includes a `self_handle` attribute indicating the `<self/>` that issued the error.

The error itself is conveyed in the `error_code` (attribute). The value of this attribute is a token indicating the specific error that occurred. At present this will be the name of a Python exception; the production version of this protocol will nail down the allowed error tokens here, probably in the RelaxNG schema.

The body of the `<report_error/>` element itself is an optional text string; if present, this is debugging information. At present this capabilty is not used, debugging information goes to syslog.

RPKI utility programs

The distribution contains a few small utility programs. Most of these are nominally relying party tools. Some but not all of them are installed by "make install".

uri

`uri` is a utility program to extract URIs from the SIA, AIA, and CRLDP extensions of one or more X.509v3 certificates.

Usage:

```
$ uri [-p | -d] cert [cert...]
```

-d Input is in DER format

-p Input is in PEM format

-s Single output line per input file

-v Verbose mode

The utils/uri directory also includes a few experimental AWK scripts to post-process the program's output in various ways.

hashdir

`hashdir` copies an authenticated result tree from an rcynic run into the format expected by most OpenSSL-based programs: a collection of "PEM" format files with names in the form that OpenSSL's -CApath lookup routines expect. This can be useful for validating RPKI objects which are not distributed as part of the repository system.

Usage:

```
$ hashdir input-directory output-directory
```

print_rpki_manifest

`print_rpki_manifest` prettyprints the content of a manifest. It does *NOT* attempt to verify the signature. Usage:

```
$ print_rpki_manifest [-c] manifest [manifest...]
```

-c Print text representation of entire CMS blob

print_roa

`print_roa` prettyprints the content of a ROA. It does *NOT* attempt to verify the signature.

Usage:

```
$ print_roa [-b] [-c] [-s] ROA [ROA...]
```

-b Brief mode (only show ASN and prefix)

-c Print text representation of entire CMS blob

-s Show CMS signingTime

find_roa

`find_roa` searches the authenticated result tree from an rcynic run for ROAs matching specified prefixes.

Usage:

```
$ find_roa authtree prefix [prefix...]
```

The `find_roa` directory also includes a script {{{test_roa.sh}, which uses `hashdir`, `print_roa`, `find_roa`, and the OpenSSL command line tool. `find_roa` builds a hashed directory, searches for ROAs matching specified prefixes, verifies the CMS signature and certificate path of each ROA found, and prettyprints each ROA that passes the checks.

Usage:

```
$ test_roa.sh authtree prefix [prefix...]
```

scan_roas

`scan_roas` searchs the authenticated result tree from an rcynic run for ROAs, and prints out the signing time, ASN, and prefixes for each ROA, one ROA per line.

Other programs such as the rpki-rtr client use `scan_roas` to extract the validated ROA payload after an rcynic validation run.

Usage:

```
$ scan_roas authtree
```

Overview Of RPKI Protocols

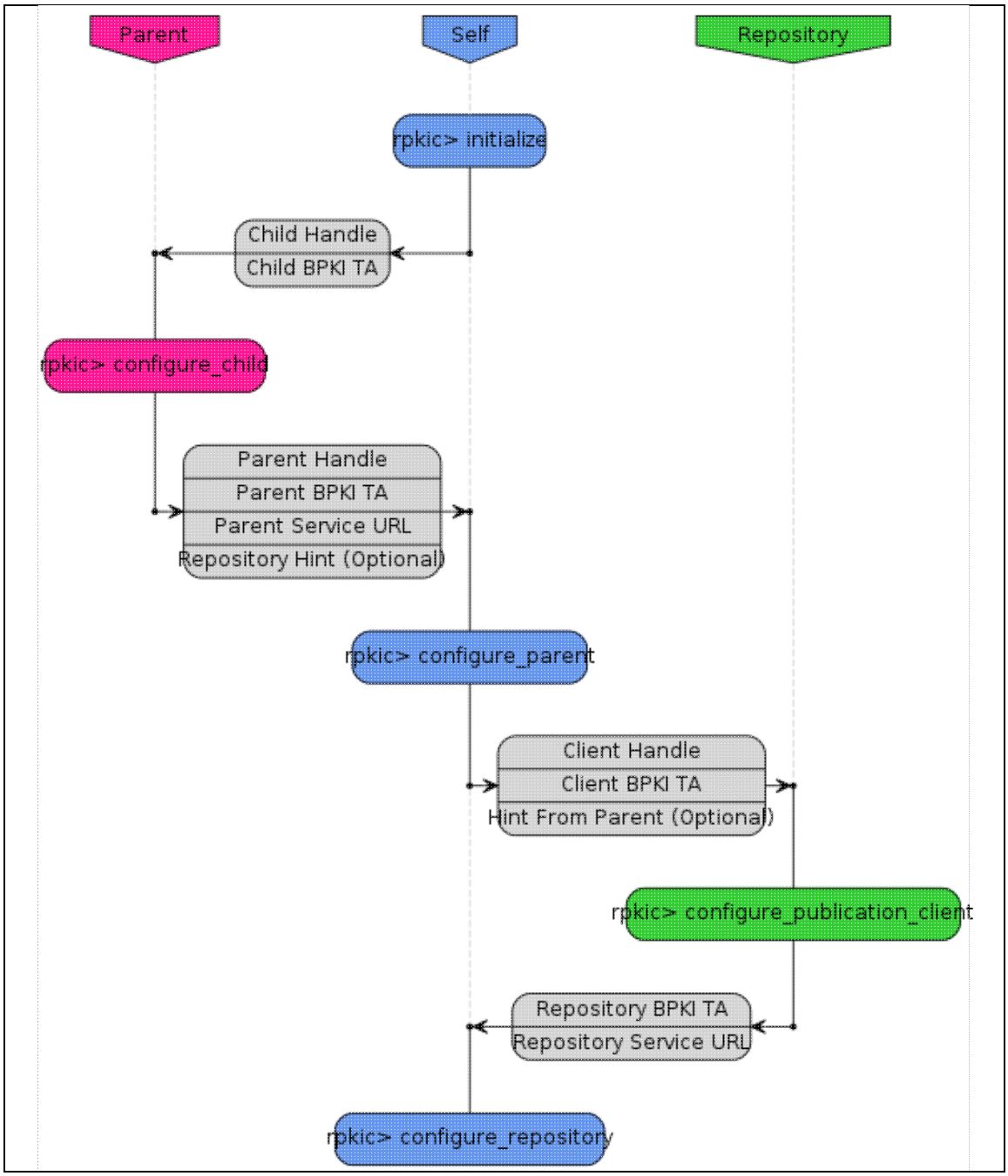
Brief overview of certain RPKI protocols. This is a work in progress.

- The out-of-band setup protocol
- The "Up-Down" provisioning protocol

The RPKI Out-Of-Band Setup Protocol

This protocol configures the initial URLs and BPKI certificates needed to operate the up-down and publication protocols. This is not an IETF standard of any kind. The rpki.net code is, as far as we know, the only complete implementation of this protocol, but other RPKI CA packages implement portions of it for interoperability.

In the long run we intend to clean this up and submit the cleaned-up version as a candidate for IETF standardization, since it seems to be in everyone's best interests, but we're not there yet.



RPKI "Up-Down" Provisioning Protocol

This is the provisioning protocol described in RFC-6492.