RPKI Tools Manual

http://rpki.net

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

At the moment, the only way to obtain the code is via subversion. At some point we should start producing snapshot tarballs.

To download the code, do:

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

Installation

See the installation instructions for how to install the code 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.

Installation

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, and MacOSX.

Prerequisites

Before attempting to build the tools, you 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 2.5, 2.6, or 2.7.

On some platforms (particularly MacOSX) the simplest way to install most of the Python packages may be the easy_install tool that comes with Python.

Packages you will need:

- We do not (yet) have binary packages for any platform, so you will need a C compiler. gcc is fine, others such as Clang may 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
    - Ubuntu:
        - python
        - python-dev

- http://codespeak.net/lxml/, a Pythonic interface to the Gnome LibXML2 libraries. lxml in turn requires the LibXML2 C libraries.
    - FreeBSD: /usr/ports/devel/py-lxml
    - Fedora: python-lxml.i386
    - Ubuntu: python-lxml

- 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
  - /usr/ports/databases/mysql55-client
- Ubuntu:
  - mysql-client
  - mysql-server

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

- http://www.djangoproject.com/, the Django web user interface toolkit. The GUI interface to the CA tools requires this.
  - FreeBSD: /usr/ports/www/py-django

- 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
  - Ubuntu: python-yaml

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

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.

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 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
    - 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

This collection of 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.

Overview of the tools

Here's a brief summary of the current relying party tools.

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.

See the instructions for setting up and running rcynic.

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 instructions for setting up rtr-origin for further details.

roa-to-irr

roa-to-irr is an experimental program for converting RPKI ROA data into IRR data. Some operators have established procedures that depend heavily on IRR, so being able to distribute validated RPKI data via IRR is somewhat useful to these operators.

Opinions vary regarding exactly what the RPSL corresponding to a particular set of ROAs should look like, so roa-to-irr is currently experimental code at best. Operators who really care about this may well end up writing their own ROA to IRR conversion tools.

roa-to-irr expects its output to be piped to the `irr_rpsl_submit` program.

roa-to-irr isn't really documented (yet?). If you care, see the code.

rpki-torrent

rpki-torrent is an experimental program for distributing unvalidated RPKI data over the BitTorrent protocol. Such data still needs to be validated by the relying party (rpki-torrent does this automatically), BitTorrent is just being used as an alternative transport protocol.

rpki-torrent isn't really documented yet.

Utilities

You may also find some of the RPKI utility programs useful.

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.

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 (the normal case) or not and on the platform on which you're running rcynic, as the chroot utilities on different platforms behave slightly differently.

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. At some future date we may provide some sort of wrapper script which handles this for you.

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
/usr/local/bin/python /var/rcynic/etc/rcynic.py /var/rcynic/data/rcynic.xml /var/rcynic/data/rc
cd /var/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/rpki-rtr
chown rcynic /var/rpki-rtr
```

On Linux, the script might look like this:

```
#!/bin/sh -
/usr/sbin/chroot --userspec rcynic:rcynic /var/rcynic /bin/rcynic -c /etc/rcynic.conf || exit
/usr/local/bin/python /var/rcynic/etc/rcynic.py /var/rcynic/data/rcynic.xml /var/rcynic/data/rc
cd /var/rpki-rtr
/usr/bin/su -m rcynic -c '/usr/local/bin/rtr-origin --cronjob /var/rcynic/data/authenticated'
```

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 $hosts
do
  /usr/sbin/chroot -u rcynic -g rcynic /var/rcynic /bin/rcynic -c /etc/rcynic.conf -u /data/una
  /usr/local/bin/python /var/rcynic/etc/rcynic.py /var/rcynic/data/rcynic.xml /var/rcynic/data/
  /bin/cp -p /var/rcynic/data/rcynic.html  /var/rcynic/data/rcynic.${host}.html
done
cd /var/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
  /usr/local/bin/python /var/rcynic/etc/rcynic.py /var/rcynic/data/rcynic.xml /var/rcynic/data/
```

```
  /bin/cp -p /var/rcynic/data/rcynic.html  /var/rcynic/data/rcynic.${host}.html
done
cd /var/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
```

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 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 run work on Ubuntu (8.10) 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 rcynic.conf file for you and set up the chroot jail.

See the instructions for setting up a cron job for details on how to run rcynic automatically and feed its output into other relying party tools.

Overview

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

All certificates, CRLs, and CMS objects are in DER format, with 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 of the 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 config file uses OpenSSL's config 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 progran, perhaps under cron, you'll want to set additional parameters to tell rcynic where to find its data and where to write its output. The configuration file itself, however, is not optional. In order for rcynic to do anything useful, your configuration file **MUST** specify the file name(s) of one or more RPKI trust anchors or trust anchor locators.

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.

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 global trust anchor for the RPKI system, so you have to specify separate trust anchors for each Regional Internet Registry (RIR) which is publishing RPKI data:

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 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.

rynic stores trust anchors specified via the trust-anchor-locator directive in the unauthenticated tree just like any other fetched object, and copies 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 xxxxxxxx.n.cer, where xxxxxxxx and n are the OpenSSL object name hash and index within the resulting virtual hash bucket (the same as the c_hash Perl script that comes with OpenSSL would produce), and ".cer" is the literal string ".cer". The reason for this 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.

As currently implemented, rcynic does not attempt to maintain an in-memory cache of objects it might need again later. It does keep an internal cache of the URIs from which it has already fetched data in this pass, and it keeps a stack (actually, multiple stacks, to support parallel operations) containing the current certificate chain as it does its validation walk. All other data (eg, CRLs) are freed immediately after use and read from disk again as needed. From a database design standpoint, this is not very efficient, but as the rcynic's main bottlenecks are expected to be crypto and network operations, it seemed best to keep the design as simple as possible, at least until execution profiling demonstrates a real issue here.

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

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.

Configuration variables:

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 seconds.

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. At least one RIR currently refuses service at settings above 4, and another RIR appears to be running some kind of firewall that silently blocks connections when it thinks decides that the connection rate is excessive.

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 RPKI rsync servers into the ground at midnight UTC.

Default: 600

lockfile::

Name of lockfile, or empty for no lock. If you run rcynic under cron, you should use this parameter to set a lockfile so that successive instances of rcynic don't stomp on each other.

Default: no lock

xml-summary::

Enable output of a per-host summary at the end of an rcynic run in XML format. Some users prefer this to the log_telemetry style of logging, or just want it in addition to logging. Value: filename to which XML summary should be written; "-" will send XML summary to stdout.

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.

Values: true or false

Default: true

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. Do not even consider enabling this unless you are intimately familiar with X.509 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 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

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, represented as a local file containing an rsync URI and the RSA public key of the X.509 object specified by the URI. First line of the file is the URI, remainder is the public key in Base64 encoded DER format. Value of this option is the pathname of the file.

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.py

Converts rcynic's XML output into a pretty (well, we like it) HTML page summarizing the results then providing detailed information on each object retrieved, with some color coding intended to highlight issues needing attention. The intent is that the top of this page provide an at-a-glance summary of the state of the world; in practice, the state of the world tends to be messy enough that this is less useful than one might hope, but it's a start.

Usage:

```
$ python rcynic.py rcynic.xml rcynic.html
```

rcynic.xsl

An earlier attempt at the same kind of HTML display as rcynic.py. XSLT was a convenient language for our initial attempts at this, but as the processing got more and more complex, the speed and flexibility restrictions of XSLT became prohibitive. 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 is significantly slower than rcynic.py.

Usage:

```
$ xsltproc -o rcynic.html rcynic.xsl rcynic.xml
```

show.xsl / show.sh

Provides a quick flat text summary of validation results. Useful primarily in test scripts (smoketest uses it).

Usage:

```
$ sh show.sh rcynic.xml
```

validation_status.xsl / validation_status.awk

Provides a flat text translation of the detailed validation results. Useful primarily for checking the detailed status of some particular object or set of objects, most likely using a program like grep or awk to filter the output.

Usage:

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

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 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. Here's how.

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.

You'll need a chroot 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 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 will likely not be a precise match for the preferred way of doing this on any particular platform. We have sample scripts that do the right thing for FreeBSD, feel free to contribute such 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 you've created a user "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,a nd /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.

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 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]

trust-anchor-locator.1  = /etc/trust-anchors/ta-1.tal
trust-anchor-locator.2  = /etc/trust-anchors/ta-2.tal
trust-anchor-locator.3  = /etc/trust-anchors/ta-3.tal

rsync-program           = /bin/rsync
authenticated           = /data/authenticated
unauthenticated         = /data/unauthenticated
```

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: just set the environment variable `LDFLAGS='-static'` before building the rsync port 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 your syslogd configuration, 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"
```

rpki-rtr

rtr-origin is an implementation of the rpki-rtr protocol.

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 setting up rcynic:

1. You need to set up post-processing of rcynic's output into the data files used by rtr-origin, and
2. You need to set up a listener for the rtr-origin server, using the generated data files.

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.

To set this up, add an invocation of `rtr-origin --cronjob` to the cron job you're already running to run rcynic. In `--cronjob` rtr-origin, needs write access to a directory where it can store pre-digested versions of the data it pulls from rcynic.

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.

So, assuming that rtr-origin is installed in /usr/local/bin, that rcynic writes its data files under /var/rcynic/data/authenticated, and you want rtr-origin to write its datafiles to /var/rpki-rtr, you'd add something like the following to your cronjob:

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

See the instructions for setting up a cron job for an example of how to run rcynic and rtr-origin together in a single cron job.

You should make sure that rtr-origin 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 so that you can give that user its own home directory and ssh configuration files.

`--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, 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/rpki-rtr
```

   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/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/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/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.
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.

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
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

This section describes the configuration file syntax and settings.

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 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.

rpki.conf

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, unless you use the `--sysconfdir` option to `./configure`, in which case the default location is whatever directory you gave `./configure` as the argument to this option.

You can override the build-time default filename at runtime 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 specifying the name of the configuration file; if both the command line option and the environment variable are set, the command line option wins.

Unless you really know what you're doing, you should start by copying the `rpki.conf` from the `rpkid/examples` directory and modifying it, as the sample configuration file already includes all the additional settings necessary to use the simplified configuration.

We really should have a configuration wizard script which leads you through the process of creating a basic `rpki.conf` file, but we haven't written it yet. Someday Real Soon Now.

```
[myrpki]
```

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

```
# Handle naming hosted resource-holding entity (<self/>) represented
# by this myrpki instance.  Syntax is an identifier (ASCII letters,
# digits, hyphen, underscore -- no whitespace, non-ASCII characters,
# or other punctuation).  You need to set this.

handle                          = Me
```

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.

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.

```
# Directory for BPKI files generated by rpkic and used by rpkid and pubd.
# Default is where we expect autoconf to decide that our data files
# belong, you might want or need to change this.   In the long term
# this should be handled by a setup wizard.

bpki_servers_directory          = /usr/local/share/rpki
```

You shouldn't need to change this unless you used the `--datarootdir` option to tell `./configure`; if you did, you'll need to adjust the setting of `bpki_servers_directory` to match whatever you told `./configure`.

```
# Whether you want to run your own copy of rpkid (and irdbd).  You
# want this on unless somebody else is hosting rpkid service for you.

run_rpkid                       = true
```

You probably don't need to change this.

```
# DNS hostname and server port numbers for rpkid and irdbd.  rpkid's
# server host has to be a publicly reachable name to be useful;
# irdbd's server host should always be localhost unless you really
# know what you are doing.  Port numbers can be any legal TCP port
# number that you're not using for something else.

rpkid_server_host               = rpkid.example.org
rpkid_server_port               = 4404
irdbd_server_host               = localhost
irdbd_server_port               = 4403
```

You'll need to set at least the `rpkid_server_host` parameter here. You may be able to use the default port numbers, or may need to pick different ones. Unless you plan to run irdbd on a different machine from rpkid, you should leave `irdbd_server_host` alone.

```
# Whether you want to run your own copy of pubd.  In general, it's
# best to use your parent's pubd if you can, to reduce the overall
# number of publication sites that relying parties need to check, so
# don't enable this unless you have a good reason.

run_pubd                        = false

# DNS hostname and server port number for pubd, if you're running it.
# Hostname has to be a publicly reachable name to be useful, port can
# be any legal TCP port number that you're not using for something
# else.

pubd_server_host                = pubd.example.org
pubd_server_port                = 4402

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

pubd_contact_info               = repo-man@rpki.example.org
```

The out of band setup protocol will attempt to negotiate publication service for you with whatever publication service your parent is using, if you let it, so in most cases you should not need to run pubd unless you need to issue certificates for private IP address space or private Autononmous System Numbers.

If you do run pubd, you will need to set `pubd_server_host`. You may also need to set `pubd_server_port`, and you should provide something helpful as contact information in `pubd_contact_info` if you plan to offer publication service to your RPKI children, so that grandchildren (or descendents even further down the tree) who receive referrals to your service will know how to contact

you.

```
# 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                       = false

# 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
```

You shouldn't run rootd unless you're the root of an RPKI tree. Who gets to be the root of the public RPKI tree is a political issue outside the scope of this document. For everybody else, the only reason for running rootd (other than test purposes) would be to support certification of private IP addresses and ASNs. The core tools can do this without any problem, but the simplified configuration mechanism does not (yet) make this easy to do.

```
# Root of local directory tree where pubd (and rootd, sigh) 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_directory      = publication/

# 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

# 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}
```

These parameters control the mapping between the rsync URIs presented by rsyncd and the local filesystem on the machine where pubd and rsyncd run. Any changes here must also be reflected as changes in `rsyncd.conf`. In most cases you should not change the value of `publication_rsync_module` from the default; since pubd can't (and should not) rewrite `rsyncd.conf`, it's best to use a static rsync module name here and let pubd do its work underneath that name. In most cases `publication_rsync_server` should be the same as `publication_rsync_server`, which is what the macro invocation in the default setting does. `publication_base_directory`, like other pathnames in `rpki.conf`, can be either a relative or absolute pathname; if relative, it's interpreted with respect to the directory in which the programs in question were started. It's probably better to use an absolute pathname, since this pathname must also appear in `rsyncd.conf`.

```
# Startup control.  These all default to the values of the
# corresponding run_* options, to keep things simple.  The only case
# where you would want to change these 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_*" controls whether
# the back-end code is doing things to manage the daemon in question,
# while "start_*" controls whether rpki-start-servers attempts to
# start the daemon in question.

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

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

You don't need to change these unless for some reason you need to run rpkid, pubd, or both on different machines from your back end code. In such cases, you can use these options to control which daemons start on which hosts, and to tell the back end code (rpkic and the GUI) that they're responsible for talking to rpkid and pubd even though those daemons are running on other hosts.

The main reason why you might want to do this would be cases where you might want to run rpkid and pubd in a DMZ while keeping all of the back end code behind a firewall.

```
# SQL configuration.  You can ignore this if you're not running any of
# the daemons yourself.

# If you're comfortable with having all of the databases use the same
# MySQL username and password, set those values here.  It's ok to
# leave the default username alone, but you should use a locally
# generated password either here or in the individual settings below.

shared_sql_username             = rpki
shared_sql_password             = fnord

# If you want different usernames and passwords for the separate SQL
# databases, enter those settings here; the shared_sql_* settings are
# only referenced here, so you can remove them entirely if you're
# setting everything in this block.

rpkid_sql_database              = rpkid
rpkid_sql_username              = ${myrpki::shared_sql_username}
rpkid_sql_password              = ${myrpki::shared_sql_password}

irdbd_sql_database              = irdbd
irdbd_sql_username              = ${myrpki::shared_sql_username}
irdbd_sql_password              = ${myrpki::shared_sql_password}

pubd_sql_database               = pubd
pubd_sql_username               = ${myrpki::shared_sql_username}
pubd_sql_password               = ${myrpki::shared_sql_password}
```

These settings control how rpkid, irdbd, and pubd talk to the MySQL server. At minimum, each daemon needs its own database; in the simplest configuration, the username and password can be shared, which is what the macro references in the default configuration does. If for some reason you need to set different usernames and passwords for different daemons, you can do so by changing the daemon-specific variables.

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.

Other configuration files and options

In most cases the simplified configuration in the `[myrpki]` section of `rpki.conf` should suffice, but in case you need to tinker, here are details on the the rest of the configuration options. In most cases the default name of the configuration file for a program is the name of the program followed by `.conf`, and the section name is also named for the program, so that you can combine sections into a single configuration file as shown with `rpki.conf`.

Next steps

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

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.

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.

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

- Copy the rpki.conf and bpki directories you just created on the backend host over to the rpkid and pubd hosts, but only copying the private key (.key file) for the service in question. So 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.

- Edit the rpki.conf files on all three servers to customize their roles:
  - `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 `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 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.

rpkid.conf

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

Options:

startup-message::

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

sql-username::

Username to hand to MySQL when connecting to rpkid's database.

sql-database::

MySQL's database name for rpkid's database.

sql-password::

Password to hand to MySQL when connecting to rpkid's database.

bpki-ta::

Name of file containing BPKI trust anchor. All BPKI certificate verification within rpkid traces back to this trust anchor.

rpkid-cert::

Name of file containing rpkid's own BPKI EE certificate.

rpkid-key::

Name of file containing RSA key corresponding to rpkid-cert.

irbe-cert::

Name of file containing BPKI certificate used by IRBE (rpkic, GUI) when talking to rpkid.

irdb-cert::

Name of file containing BPKI certificate used by irdbd.

irdb-url::

Service URL for irdbd. Must be a `http://` URL.

server-host::

Hostname or IP address on which to listen for HTTP connections. Default is the wildcard address (IPv4 `0.0.0.0`, IPv6 `::`), which should work in most cases.

server-port::

TCP port on which to listen for HTTP connections.

irdbd.conf

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 section "[irdbd]".

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.

Options:

startup-message::

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

sql-username::

Username to hand to MySQL when connecting to irdbd's database.

sql-database::

MySQL's database name for irdbd's database.

sql-password::

Password to hand to MySQL when connecting to irdbd's database.

http-url::

Service URL for irdbd. Must be a `http://` URL.

pubd.conf

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 section "[pubd]". Certificates and keys may be either DER or PEM format.

Options:

sql-username::

Username to hand to MySQL when connecting to pubd's database.

sql-database::

MySQL's database name for pubd's database.

sql-password::

Password to hand to MySQL when connecting to pubd's database.

bpki-ta::

Name of file containing master BPKI trust anchor for pubd. All BPKI validation in pubd traces back to this trust anchor.

irbe-cert::

Name of file containing BPKI certificate used by IRBE (rpkic, GUI) when talking to pubd.

pubd-cert::

Name of file containing BPKI certificate used by pubd.

pubd-key::

Name of file containing RSA key corresponding to `pubd-cert`.

server-host::

Hostname or IP address on which to listen for HTTP connections. Default is the wildcard address (IPv4 `0.0.0.0`, IPv6 `::`), which should work in most cases.

server-port::

TCP port on which to listen for HTTP connections.

publication-base::

Path to base of filesystem tree where pubd should store publishable objects. Default is `publication/`.

rootd.conf

Caveat

rootd is, to be blunt about it, a mess. 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. In particular, rootd does not speak the publication protocol, and requires far too many configuration parameters to work correctly. rootd is still useful as a test tool, where its shortcomings are largely hidden by automated generation of its configuration. Don't run rootd unless you're sure that you need to do so.

OK, with that out of the way....

Configuration

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 section "[rootd]". Certificates and keys may be in either DER or PEM format.

Options:

bpki-ta::

> Name of file containing BPKI trust anchor. All BPKI certificate validation in rootd traces back to this trust anchor.

rootd-bpki-cert::

> Name of file containing rootd's own BPKI certificate.

rootd-bpki-key::

> Name of file containing RSA key corresponding to rootd-bpki-cert.

rootd-bpki-crl::

> Name of file containing BPKI CRL that would cover rootd-bpki-cert had it been revoked.

child-bpki-cert::

> Name of file containing BPKI certificate for rootd's one and only child (RPKI engine to which rootd issues an RPKI certificate).

server-host::

> Hostname or IP address on which to listen for HTTP connections. Default is localhost; don't change this unless you really know what you are doing.

server-port::

> TCP port on which to listen for HTTP connections.

rpki-root-key::

Name of file containing RSA key to use in signing resource certificates.

rpki-root-cert::

Name of file containing self-signed RPKI certificate corresponding to rpki-root-key.

rpki-root-dir::

Name of directory where rootd should write RPKI subject certificate, manifest, and CRL. This needs to match pubd's configuration.

rpki-subject-cert::

Name of file that rootd should use to save the one and only certificate it issues. Default is "Child.cer".

rpki-root-crl::

Name of file to which rootd should save its RPKI CRL. Default is "Root.crl".

rpki-root-manifest::

Name of file to which rootd should save its RPKI manifest. Default is "Root.mft".

rpki-subject-pkcs10::

Name of file that rootd should use when saving a copy of the received `PKCS #10` request for a resource certificate. Default is "Child.pkcs10".

Creating a root certificate

rootd does not create the RPKI root certificate, you have to do that yourself. The usual way of doing this is to use the OpenSSL command line tool. The exact details will depend on which resources you want 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/
1.3.6.1.5.5.7.48.10;URI = rsync://example.org/rpki/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 -config root.conf -out root.req -key root.key
$ openssl x509 -req -sha256             \
        -signkey root.key -in root.req  \
        -outform DER -out root.cer      \
        -extfile root.conf -extensions x509v3_extensions
```

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".

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.

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 either rpkic or the GUI rpkic using the. 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. 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.

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).

RPKI CA Engine GUI Interface

Documentation not written yet.

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_manifest manifest [manifest...]
```

print_roa

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

Usage:

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

-b Brief mode (only show ASN and prefix)

-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
```