rtrlib-python Documentation

Release 0.1

Marcel Röthke

Mar 10, 2021

Contents

rtrlib-python is a cffi based binding for RTRlib.

The RTRlib implements the client-side of the RPKI-RTR protocol (RFC 6810) and BGP Prefix Origin Validation (RFC 6811). This release also supports Internet-Draft draft-ietf-sidr-rpki-rtr-rfc6810-bis, which enables the maintenance of router keys. Router keys are required to deploy BGPSEC.

Currently only basic validation against one cache is supported.

CHAPTER 1

Installation

1.1 Requirements

- python 2.7 or python 3
- C Compiler
- RTRlib

1.2 Python Requirements

If you are using virtualenv these are installed automatically during the install step, otherwise you have to use your platforms package management tool or just run `pip install -r requirements.txt`.

- cffi>=1.4.0
- enum34
- six

1.3 Download and Installation

- `git clone https://github.com/rtrlib/python-binding.git`
- `cd python-binding`
- `python setup.py build`
- `python setup.py install`

For usage examples see here *here* or in the tools dir of the repository.

Contents:

1.3.1 API Documentation

rtrlib-python - a cffi based rtrlib wrapper

license MIT, see LICENSE for more details.

rtrlib.rtr_manager

class rtrlib.rtr_manager.**RTRManager**(*host*, *port*, *refresh_interval=3600*, *expire_interval=7200*, *retry_interval=600*, *status_callback=None*, *status_callback_data=None*, *pfx_update_callback=None*, *pfx_update_callback_data=None*, *spki_update_callback=None*, *spki_update_callback_data=None*)

Wrapper around rtr_manager.

Parameters

- **host** (*str*) – Hostname or IP of rpki cache server
- **port** (*int*) – Port number
- **refresh_interval** (*int*) – Interval in seconds between serial queries that are sent to the server. Must be >= 1 and <= 86400s (one day).
- **expire_interval** (*int*) – Stored validation records will be deleted if cache was unable to refresh data for this period. The value should be twice the refresh_interval. The value must be >= 600s (ten minutes) and <= 172800s (two days).
- **retry_interval** (*int*) – This parameter specifies how long to wait (in seconds) before retrying a failed Query. The value must be >= 1s and <= 7200s (two hours).
- **status_callback** (*function*) – status callback, called on status changes of the rtr manager
- **status_callback_data** (*object*) – arbitrary data object passed to the callback.
- **pfx_update_callback** (*function*) – pfx update callback called every time a pfx update is received
- **pfx_update_callback_data** – data passed to the pfx update callback
- **spki_update_callback** (*function*) – spki update callback called every time a spki update is received
- **spki_update_callback_data** – data passed to the spki update callback

Raises ***RTRInitError*** –

for_each_ipv4_record(*callback*, *data*)

Iterate over all ipv4 records of the pfx table.

callback must take two arguments, the pfx_record and the data object.

For a more pythonic alternative see *ipv4_records()*

Parameters

- **callback** (*callable*) – called for every record in the pfx table
- **data** (*object*) – arbitrary data object that is passed to the callback function

for_each_ipv6_record(*callback*, *data*)
Iterate over all ipv6 records of the pfx table.

callback must take two arguments, the pfx_record and the data object.

For a more pythonic alternative see *ipv6_records()*

Parameters

- **callback** (*callable*) – called for every record in the pfx table
- **data** (*object*) – arbitrary data object that is passed to the callback function

ipv4_records()
Return iterator over all ipv4 records in the pfx table.

This iterator utilises threads to execute retrieve the records. If that is a problem for you take a look at *for_each_ipv4_record()*.

Return type Iterator

ipv6_records()
Return iterator over all ipv6 records in the pfx table.

This iterator utilises threads to execute retrieve the records. If that is a problem for you take a look at *for_each_ipv6_record()*.

Return type Iterator

is_synced()
Check if RTRManager is fully synchronized.

Return type bool

start(*wait=True*, *timeout=5*)
Start RTRManager.

Parameters

- **wait** (*bool*) – Wait for the manager to finish sync
- **timeout** (*int*) –

Raises ***SyncTimeout*** – Raised if timeout is reached, this does not mean that the sync failed, only that it did not finish in time.

stop()
Stop RTRManager.

validate(*asn*, *prefix*, *mask_len*)
Validate BGP prefix and returns state as PfxvState enum.

Parameters

- **asn** (*int*) – autonomous system number
- **prefix** (*str*) – ip address
- **mask_len** (*int*) – length of the subnet mask

Return type *ValidationResult*

wait_for_sync(*timeout=5*)
Wait until RTRManager is synchronized.

Parameters **timeout** (*int*) –

Raises ***SyncTimeout*** – Raise if timeout is reached, this does not mean that the sync failed, only that it did not finish in time.

class rtrlib.rtr_manager.**PfxvState**

Wrapper for the pfxv_state enum.

invalid

One or more records that match the input prefix exists in the pfx_table, but the prefix max_len or ASN doesn't match.

not_found

No certificate for the route exists

valid

A valid certificate for the pfx_record exists

class rtrlib.rtr_manager.**ValidationResult**(*prefix*, *prefix_length*, *asn*, *state*, *reason_records=None*, *reason_len=0*)

Wrapper class for validation result.

Parameters

- **prefix** (*str*) – The prefix that was validated
- **prefix_length** (*int*) – The length of the prefix
- **asn** – The ASN the prefix is supposed to be in.
- **asn** – int
- **state** (*enum pfxv_state **) – Validation state
- **reason_records** (*struct pfx_record ***) – Array of PFXRecords the decision is based on
- **reason_len** (*int*) – Length of reason_records

as_invalid

True if at least one matching record has a different as number and state is invalid.

is_invalid

Return true if prefix is invalid.

is_valid

True if prefix is valid.

length_invalid

True if at least one matching record has a miss matching prefix length and state is invalid.

not_found

True if prefix could not be found.

reason

List of *Reason* .

state

Validation state.

class rtrlib.rtr_manager.**Reason**(*prefix_length*, *asn*, *record*)

A Reason upon which a validation decision was made.

Parameters

- **prefix_length** (*int*) – Length of the validated prefix
- **asn** (*int*) – As number of the validated prefix

- **record** (PFXRecord) – PFXRecord

as_invalid
True is as is invalid.

as_valid
True if as is valid.

length_invalid
True if prefix length is invalid.

length_valid
True if prefix length is valid.

rtrlib.rtr_socket

class rtrlib.rtr_socket.**RTRSocket**(*socket*)
Wrapper around the rtr_socket struct

Parameters **socket** (*cdata*) – rtr_socket struct

expire_interval
Time period in seconds. Received records are deleted if the client was unable to refresh data for this time period. If 0 is specified, the expire_interval is twice the refresh_interval.

has_recieved_pdus
True, if this socket has already received PDUs

last_update
Timestamp of the last validation record update. Is 0 if the pfx_table doesn't stores any validation records from this rtr_socket.

refresh_interval
Time period in seconds. Tells the router how long to wait before next attempting to poll the cache, using a Serial Query or Reset Query PDU.

retry_interval
Time period in seconds between a failed query and the next attempt.

state
Current state of the socket.

version
Protocol version used by this socket

class rtrlib.rtr_socket.**RTRSocketList**(*sockets*, *length*)
List of RTRSockets. Can be accessed like any other list.

Read Only.

class rtrlib.rtr_socket.**RTRSocketState**
States of the RTR socket

CONNECTING
Socket is establishing the transport connection

ERROR_FATAL
Fatal protocol error occurred

ERROR_NO_DATA_AVAILABLE
No validation records are available on the RTR server

ERROR_NO_INCREMENTAL_UPDATE_AVAILABLE
Server was unable to answer the last serial or reset query

ERROR_TRANSPORT
Error on the transport socket occurred

ESTABLISHED
Connection is established and socket is waiting for a Serial Notify or expiration of the refresh_interval timer.

FAST_RECONNECT
Reconnect without any waiting period

RESET
Resetting RTR connection

SHUTDOWN
RTR Socket is stopped

SYNC
Receiving validation records from the RTR server

rtrlib.records

Collection of wrappers for *record structs of rtrlib

class rtrlib.records.**PFXRecord**(*record*)
Wrapper around the pfx_record struct.

asn
Origin AS number.

max_len
Maximum prefix length.

min_len
Minimum prefix length.

prefix
IP prefix.

socket
RTRSocket this record was received in.

class rtrlib.records.**SPKIRecord**(*record*)
Wrapper around the spki_record struct.

asn
Origin AS number.

ski
Subject Key Identifier.

socket
RTRSocket this record was received in.

spki
Subject public key info.

rtrlib.records.**copy_pfx_record**(*record*)
Copy a pfx record.

Parameters **record** (PFXRecord) – The record that should be copied

Return type *PFXRecord*

rtrlib.manager_group

class rtrlib.manager_group.**ManagerGroup**(*group*)
Wrapper around the rtr_mgr_group struct

Parameters **group** (*cdata*) – A rtr_mgr_group struct

preference
The preference value of the group

sockets
The socket list as RTRSocketList

sockets_len
The sockets_len value of the group

status
The group status as enum34

class rtrlib.manager_group.**ManagerGroupStatus**
Wrapper around the C enum rtr_mgr_status.

CLOSED
RTR sockets are disconnected

CONNECTING
RTR sockets trying to establish a connection

ERROR
Error occured on at least one RTR socket

ESTABLISHED
All RTR sockets of the group are synchronized with the rtr servers

rtrlib.exceptions

Module for all custom exceptions

exception rtrlib.exceptions.**IpConversionException**
An Error during str to address conversion or the reverse occurred.

exception rtrlib.exceptions.**PFXException**
An error during validation occurred.

exception rtrlib.exceptions.**RTRInitError**
An error during initialization of the RTR manager occurred.

exception rtrlib.exceptions.**RTRlibException**
rtrlib exception base class.

exception rtrlib.exceptions.**SyncTimeout**
The timeout was reached while waiting for sync.

1.3.2 Callbacks

Rtrlib provides 3 callbacks one for updates on the manager status, one for pfx_table and one for spki_table updates.

RTR Manager Status Callback

This callback is called when the RTR Managers status is changed. The callback function must take 4 arguments.

manager_status_callback(*rtr_mgr_group*, *group_status*, *rtr_socket*, *data*)

Parameters

- **rtr_mgr_group** – socket group where the status change originates
- **group_status** – the new status
- **rtr_socket** – the socket where the change originates
- **data** (*object or None*) – Data Object, if defined at manager initialization

This callback is registered at manager initialization using status_callback parameter. The data object may be passed with the status_callback_data parameter.

PFX iteration callback

This callback can be used to iterate over the entire pfx table.

pfx_for_each(*pfx_record*, *data*)

pfx_record is only guaranteed to be valid during this function call. If you want to store it somewhere e.g. in *data* than you have to copy it. you can use *rtrlib.records.copy_pfx_record()* for this.

Parameters

- **pfx_record** (PFXRecord) – Pfx record from the iterated pfx table
- **data** (*object or None*) – Arbitrary data object provided by the user

PFX update callback

This callback is called for any change to the Prefix validation table, it takes two arguments.

pfx_update_callback(pfx_record, added):

Parameters

- **pfx_record** (*rtrlib.records.PFXRecord*) – The affected pfx record
- **added** (bool) – Indicates whether the record was added or removed
- **data** – Data Object, if defined at manager initialization

This callback is registered at manager initialization using the pfx_update_callback parameter. The data object may be passed with the pfx_update_callback_data parameter.

SPKI update callback

This callback is called for any change to the Subject Public Key Info table, it takes two arguments.

> **Warning:** This callback is untested due to the lack of spki support in rtr cache server implementations.

spki_update_callback(spki_record, added):

Parameters

- **spki_record** (*rtrlib.records.SPKIRecord*) – The affected spki record
- **added** (bool) – Indicates whether the record was added or removed
- **data** – Data Object, if defined at manager initialization

This callback is registered at manager initialization using the spki_update_callback parameter. The data object may be passed with the spki_update_callback_data parameter.

1.3.3 Usage Examples

Validation

```
from rtrlib import RTRManager, PfxvState

mgr = RTRManager('rpki-validator.realmv6.org', 8282)
mgr.start()
result = mgr.validate(12345, '10.10.0.0', 24)

if result == PfxvState.valid:
    print('Prefix Valid')
elif result == PfxvState.invalid:
    print('Prefix Invalid')
elif result == PfxvState.not_found:
    print('Prefix not found')
else:
    print('Invalid response')

mgr.stop()
```

PFX Table iteration (with iterator)

```
from rtrlib import RTRManager, PfxvState

mgr = RTRManager('rpki-validator.realmv6.org', 8282)
mgr.start()
result = mgr.validate(12345, '10.10.0.0', 24)

for recordv4 in mgr.ipv4_records():
    print(recordv4)

mgr.stop()
```

PFX Table iteration (with callback)

```
from rtrlib import RTRManager, PfxvState

def callback(pfx_record, data):
    print(pfx_record)

mgr = RTRManager('rpki-validator.realmv6.org', 8282)
mgr.start()
result = mgr.validate(12345, '10.10.0.0', 24)
```

(continues on next page)

(continued from previous page)

```
mgr.for_each_ipv4_record(callback, None)

mgr.stop()
```

Print PFX updates

```
from rtrlib import RTRManager

def callback(pfx_record, added, data):
    print('%s %s' % ('+' if added else '-', pfx_record))

mgr = RTRManager('rpki-validator.realmv6.org', 8282, pfx_update_callback=callback)
mgr.start()

mgr.stop()
```

Advanced Usage

Note: This is by no means supposed to be a reference on cffi itself, if you want to do something like this please read the cffi Documentation.

I case you want to do something that is not (yet) supported by the binding you can access the c functions directly.

Let's say you implemented RFC6810 yourself but still want to use rtrlibs pfxtable.

```
# _rtrlib is the cffi object, it contains the actual bindings in lib
# and helper functions for allocation and
# other stuff that is not native to python
from _rtrlib import lib, ffi

# only imported for the pfx_table_callback
import rtrlib

# allocate pfx_table
pfx_table = ffi.new('struct pfx_table *')

# initialize it
lib.pfx_table_init(pfx_table, ffi.NULL)


def add_record(asn, ip, prefixmin, prefixmax):
    record = ffi.new('struct pfx_record *')
    prefix = ffi.new('struct lrtr_ip_addr *')
    lib.lrtr_ip_str_to_addr(ip.encode('ascii'), prefix)

    record.asn = asn
    record.min_len = prefixmin
    record.max_len = prefixmax
    record.socket = ffi.NULL
    record.prefix = prefix[0]
```

(continues on next page)

(continued from previous page)

```
    lib.pfx_table_add(pfx_table, record)

# add records
records = ((234, '22.45.66.0', 24, 24),
           (545, '9..0.0', 8, 8),
           (4545, '223.4.66.0', 24, 24),
           (5454, '120.6.47.0', 24, 24))

for record in records:
    asn, ip, min_len, max_len = record
    add_record(asn, ip, min_len, max_len)


# iterate over pfx_table to demonstrate its content

# since the callback from the rtrlib module is used record
# is automatically wrapped in a python class
def callback(record, notused):
    print(record)

# necessary because cffi new style callbacks are used,
# lib.pfx_table_callback is a wrapper that calls the actual callback
handle = ffi.new_handle((callback, None))

lib.pfx_table_for_each_ipv4_record(pfx_table, lib.pfx_table_callback, handle)

lib.pfx_table_free(pfx_table)
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index