Implementing a Prefix Manager ASA

Brian Carpenter

July 2017 (minor update July 2020)

This document is a quick report on implementing an Autonomic Service Agent (ASA) for prefix management, based on [draft-ietf-anima-prefix-management](). This is a demonstration ASA, i.e. it does not actually hand out real prefixes to real routers, but it is otherwise intended to be reasonably complete.

It is based on the Python 3 demonstration [prototype of GRASP (file graspy.pdf)]() (read that document first) and inherits its limitations, especially the absence of a real ACP and therefore the absence of security.

The implementation is called `pfxm3` and can be downloaded from [https://github.com/becarpenter/graspy](). You also need `acp.py` and `grasp.py` from the same repository and the Python CBOR module (`pip3 install cbor` should do it).

`Pfxm3` has two operating modes: *origin* and *delegator*. When running as an origin, it starts by creating two pools of prefixes (for IPv6 and IPv4), and it acts as a source of the `PrefixManager.Params` objective.

When running as a delegator, it starts with an empty prefix pool, it acquires the value of `PrefixManager.Params`, and it delegates IPv6 and IPv4 prefixes (currently all of the same length) to imaginary requesting routers. An extra feature is the delegated prefix length, a parameter entered by the user. It can be any reasonable length.

Both as an origin and as a delegator, it seeks prefixes from peer ASAs when the pool is low, and it hands out prefixes to peer ASAs when possible.

In theory a network could include any number of origins and any number of delegators, with the only condition being that each origin's initial prefix pool is unique. A realistic scenario is to have exactly one origin and as many delegators as you like. A scenario with no origin is useless.

A defect in the implementation is that the data are kept in volatile storage. If a delegator exits for any reason, all the prefixes it has obtained or delegated are lost. If

an origin exits, its entire spare pool is lost. In a real implementation, stable storage for these data is essential.

`Pfxm3` doesn't implement 'dry run' negotiation realistically. That would mean temporarily marking any prefix handed out in a dry run as reserved, until either the peer obtains it in a live run, or a suitable timeout expires.

The main data structures used are

- The prefix pool, an ordered list of available prefixes. Prefixes are split when a longer prefix is needed than is listed in the pool, and a background garbage collector recombines split prefixes if they are returned to the pool.

- The delegated list, where a delegator stores the prefixes it has given to (imaginary) routers.

The main logic flows and some more details are below.

My conclusions: Apart from figuring out the bit manipulations and eliminating a few fencepost errors, this was quite easy work. I think it shows that the whole mechanism is viable. With stable storage added, and a secure ACP, it should be safe for real world use.

<u>Main thread logic (IPv6 case only; IPv4 is similar)</u> :

Create empty prefix pool (and an associated lock)
Create empty list of delegated prefixes
Ask user whether to act as origin
if origin:
      Create initial prefix pools
else:
      Ask user for *subnet_length* to delegate (default /64)
Register ASA with GRASP
Register objectives `PrefixManager` and `PrefixManager.Params`
if origin:
      Create value of `PrefixManager.Params`
      Start thread to flood `PrefixManager.Params`
      Start synch listener for `PrefixManager.Params`
Start **main_negotiator** thread for `PrefixManager`
if not origin:
      Synchronize (obtain value of) `PrefixManager.Params`
      Start **delegator** thread
Start garbage collector (**compress**) thread for prefix pool
while true:
      if prefix pool is low:
            Calculate wanted prefix length L
            Discover peers
            Choose a peer (prefer good_peer if available)
            req_negotiate("`PrefixManager`", peer)
            if OK:
                  if offered prefix length <L+1:
                        Negotiation succeeded
                        good_peer = peer
                else:
                      Fail negotiation
      sleep(10s)

Main negotiator thread :

While true:
      listen_negotiate("PrefixManager")
      start a separate new negotiator thread

Negotiator thread:

Request prefix length L from pool
if not OK:
      while not OK and L<64:
            L = L+1
            Request prefix length L from pool
if OK:
      Offer prefix length L to peer
      if accepted:
            Negotiation succeeded
      else:
            Fail negotiation
else:
      Fail negotiation

Delegator thread:

while True:
      sleep(at least 1s) # in real world, wait for PD request
      get a prefix of length *subnet_length* from pool
      if OK:
            append it to list # in real world, respond to PD request
      else:
            signal main thread that pool is low

Compress thread:

while True:
      search pool for adjacent prefixesof length L
      if they match at length L-1:
            merge prefixes
      sleep(5s)

Important global variables

```
ppool = []              #prefix pool
                        #The format is an array of tuples, such that
                        # ppool[i] = [plen, prefix] where
                        # plen is the prefix length (1..128) and
                        # prefix is a bytes object of 128 bits

pool_lock               #lock for thread-safe access to pool

delegated = []          #list of delegated prefixes
                        #(same format as ppool)

need = 0                #needed prefixes (counted in /64s)

origin = False          #Boolean

good_peer = None        # where we remember a helpful peer ASA

obj1 = grasp.objective("PrefixManager")      #when acting as server

obj2 = grasp.objective("PrefixManager.Params")

want_obj = grasp.objective("PrefixManager") #when acting as client
```

Threads and Functions

Threads:

class flooder(threading.Thread):

   """Thread to flood PrefixManager.Params repeatedly"""

class main_negotiator(threading.Thread):

   """Main negotiator"""

class negotiator(threading.Thread):

   """Thread to negotiate PrefixManager as server"""

class delegator(threading.Thread):

   """Thread to delegate prefixes"""

class compress(threading.Thread):

   """Thread to compress pool"""

Functions (IPv4 versions are similar, where needed):

def endit(snonce, r):

"""Support function for negotiator"""

def make_mask(plen):

"""-> bytes object that is a mask of plen bits"""

def mask_prefix(plen,prefix):

"""-> packed prefix masked to length plen"""

def split_prefix(plen, prefix):

"""-> plen+1, prefix1, plen+1, prefix2"""

def create_pool():

"""makes a prefix pool, called only in origin"""

def get_from_pool(plen):

"""-> packed prefix of requested length, or None"""

def insert_pool(plen, prefix):

"""inserts in pool in canonical order"""

def sum_pool():

""" -> estimate of pool size in /64s"""

def nudge_pool(L):

"""signal need for prefixes of length L"""

Diagnostic functions:

dump_pool():

"""Print prefix pool"""

def dump_delegates():

"""Print delegated prefixes"""

def dump_some():

"""Print obj_registry and flood cache"""