draft-gashinsky-v6nd-enhance-00

Igor Gashinsky, Yahoo!
igor@yahoo-inc.com
Warren Kumari, Google
warren@kumari.net
Joel Jaeggli, Zynga
jjaeggli@zynga.com

IETF 81

Motivation:

- IPv6 subnet of a /64 has trillions of addresses
  - Implementation of IPv6 NDP makes routers supporting such large subnets vulnerable to scan-based attacks
  - Same issues existed in IPv4, but subnets were significantly smaller, and had very little unused space
    - CPU resources could be exhausted performing resolutions for non-existant space
    - Memory resources could be exhausted for keeping a huge incomplete table

- Address the problem via:
  - Operational work-arounds
  - Implementation optimizations
  - Protocol enhancements

Operational workarounds (1):

- Filtering out unused address space
  - Pros:
    - Fast to deploy (on small scale)
  - Cons:
    - Hard to manage
    - Hard to scale
    - Adds to provisioning time
    - Doesn't work well with SLAAC

- Smaller Subnet sizing
  - Pros:
    - Common practice in Ipv4
    - Minimizes unused address space
  - Cons:
    - Makes it difficult to optimize address ranges
    - Doesn't work with SLAAC
    - Goes against the /64 per subnet convention
      - Too many implementations assume /64

Operational workarounds (2)

- Routing mitigation (ie route injection on the servers)
  - Pros:
    - It works :)
  - Cons:
    - 6k servers running routing...
- Tuning of NDP queue limits:
  - Pros:
    - May help in some case
      - Especially if there are seperate resolution queues
  - Cons:
    - Can “complete the attack”
    - May have negative implication on convergence

Recommendations for implementors:

1) NDP Prioritization

2) NDP Queue Tuning

Protocol Enhancements:

3) NDP Protocol Gratuitous NA

4) ND Cache Priming & Refresh

NDP Prioritization for a router:

1) Respond for your own addresses first

2) Prioritize refresh of known entries

3) Prioritize traffic sourced by the router (for control plane reasons)

4) RA related activity

5) Lookups for new/unknown destinations should have the lowest priority

NDP Queue Tuning:

- Control the size of the queue
- Control the rate of admission to the queue

Allow for gratuitous NA

- Current Standard:
  - Per RFC 4861, section 7.2.5 and 7.2.6 requires that unsolicited neighbor advertisements result in the receiver setting it's neighbor cache entry to STALE, kicking off the resolution of the neighbor using neighbor solicitation.

- Proposed Change:
  - If the link layer address in an unsolicited neighbor advertisement matches that of the existing ND cache entry, routers SHOULD/MAY? retain the existing entry updating it's status with regards to LRU retention policy.
  - NOT enabled by default
  - Use adequate safeguards

  - Rationale:
    - In datacenters/static environments, there is often only 1 way to get to an end host/default gateway (yes, there is redundancy, but often at a different layer), so no negative impact to disabling 1-way reachability detection

  - Pros:
    - 1 default gateway, 500 hosts behind it, 1 gratuitous NA populates all 500 hosts
      - Router spends less time answering for it's own NA
        - Router has more time to do other things
    - Easy for a host to “beacon” that it's alive to the router

  - Cons:
    - NDP doesn't confirm 2-way reachability (NUD suffers)

ND Cache Priming & Refresh

- Previous steps help withstand the attack, and de-prioritize new resolutions
- But, how do you add new hosts on the network while under attack?

- Solution:
  - Prime the cache via a “registration”
    - Host sends ping to all-routers multicast or unsolicited NA
  - Do everything possible to retain entry in cache
    - Be able to refresh despite brief “hick-ups” (STP, UDLD, etc)
      - Currently, the retry behavior is 3 retries, 1 second appart – Too aggressive!
      - Suggestion: implement exponential back-off for retry (see **draft-nordmark-6man-impatient-nud-00**)

      - Currently RFC4861 states:
        - **MAX_UNICAST_SOLICIT**=3 and **RETRANS_TIMER**=1second
      - Suggested algorithm :
        - operator configurable values for **MAX_UNICAST_SOLICIT**, **RETRANS_TIMER**, **BACKOFF_MULTIPLE**)
        - **next_retrans**=($**BACKOFF_MULTIPLE**^$**solicit_attempt_num**)*$**RETRANS_TIMER** + **jittered value**.
      - Recommended behavior = 5 attempts, with timing spacing of 0 (initial request), 1 second later, 3 seconds later, then 9, and then 27

      - Values:
        - MAX_UNICAST_SOLICIT=5
        - RETRANS_TIMER=1 (default)
        - BACKOFF_MULTIPLE=3
          - If BACKOFF_MULTIPLE=1 and MAX_UNICAST_SOLICIT=3, you would get the backwards-compatible RFC behavior

Next Steps?