Internet Research Task Force	J. Levine
Internet-Draft	Taughannock Networks
Intended status: Experimental	December 29, 2010
Expires: July 2, 2011	

TOC

An efficient method to publish ranges of IP addresses in the DNS
draft-levine-iprangepub-01

Abstract

The DNS has long been used to publish lists of IPv4 address ranges in blacklists and whitelists. The size of the IPv6 address space makes the entry-per-IP approach used for IPv4 lists impractical. A new technique for publishing IP address ranges is described. It is intended to permit efficient publishing and querying, and to have good DNS cache behavior.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.
Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.
Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as “work in progress.”
This Internet-Draft will expire on July 2, 2011.

Table of Contents

1. Introduction
2. Assumptions and Goals
3. DNS record format
4. Lookup algorithm
5. Details of blob representation
6. Building and updating DNSBLs
    6.1. Building static DNSxLs
    6.2. Building and updating dynamic DNSxLs
7. Estimated performance
8. Security considerations
9. Topics for further consideration
10. IANA considerations
11. References
    11.1. References - Normative
    11.2. References - Informative

Appendix A. Change Log
    A.1. Changes from -00 to -01

§ Author's Address

1. Introduction

TOC

For many years, the Domain Name System[RFC1034] (Mockapetris, P., “Domain names - concepts and facilities,” November 1987.) [RFC1035] (Mockapetris, P., “Domain names - implementation and specification,” November 1987.) has been the Internet's de facto distributed database. Blacklists and whitelists of IPv4 addresses have been published in the DNS using a simple system adapted from rDNS[RFC5782] (Levine, J., “DNS Blacklists and Whitelists,” February 2010.). A DNSxL (a DNSBL or DNSWL) is a DNS sub-tree, typically also a DNS zone, with each listed IP having an A and/or TXT record at a name corresponding to the IP address. While this publication method has worked well for IPv4 addresses, the size of the IPv6 address space makes an analogous approach unworkable.

In an IPv4 Internet, each network is typically limited to a few thousand or at most a few million addresses. A single host typically has a single address, or at most a few hundred addresses. The limited size of each network forces a host to use its assigned address or addresses. In IPv6 networks, hosts typically use Stateless Address Autoconfiguration (Thomson, S., Narten, T., and T. Jinmei, “IPv6 Stateless Address Autoconfiguration,” September 2007.) [RFC4862] to select an IP address, with the low 64 bits of the address being almost entirely arbitrary. A hostile host sending mail can easily switch to a

new address for every message it sends, never reusing an address, due to the vast size of the IPv6 address space.

An IPv6 DNS blacklist would use wildcards or a specialized server such as [rbldnsd (Tokarev, M., “rbldnsd: Small Daemon for DNSBLs,” .)] [RBLDNSD] to list entire /64 or larger ranges, but that does not help DNS performance. Since wildcards are expanded by the DNS server, every query for a unique IP address causes a unique query to the DNSBL's server. Moreover, every unique query will take up a cache entry in the client's local DNS cache, either a regular entry if the DNSBL lists the query's address, or a negative entry[RFC2308] (Andrews, M., “Negative Caching of DNS Queries (DNS NCACHE),” March 1998.) if it doesn't. In the event that hostile mailers (which we will call "spammers" for short) use a unique address per message, the normal DNSBL query traffic will both flood the DNSBL's server and fill local caches with useless single-use entries, forcing out other cached data and causing excess traffic to all the other servers the caches query as well.

For blacklists, an obvious approach would be to limit the granularity of DNSBLs, so that, say, each /64 had a separate listing, and the queries only used the high 64 bits of each address. While this might limit the damage from DNSBL queries, it is not helpful for DNS whitelists, which by their nature list individual IP addresses, and are likely to be far more popular with IPv6 mail than they have been with IPv4 mail.

The problem of looking up an address in a sorted list of IP addresses stored on a remote DNS server is not unlike the problem of searching for a key in a sorted list of keys stored on a disk, a problem that is common in database applications. An approach called *B-trees* has been widely used in databases since the 1970s, and is described in standard references such as [KNUTHV3] (Knuth, D., “The Art of Computer Programming: Volume 3, Sorting and Searching,” 1998.). The technique in this document stores ordered sets of IP address ranges in DNS records, using a straightforward adaptation of the way that B-trees store ordered sets of key strings in disk blocks.

---

2. Assumptions and Goals

TOC

This design is intended to meet this set of design goals:

1. Handle arbitrary mixtures of prefix ranges and individual IPs.

2. Work well with existing DNS servers and caches. The number of different queries should be limited, both to limit cache->server traffic and the number of cache entries it uses. I assume that client->cache queries are cheap, while cache->server queries are much more expensive.

3. Don't assume senders will be well behaved. In particular, bad guys may use a new IPv6 address for every message, hopping around within each /64 (or whatever the network size is) in which a zombie lives, and there will be many zombies active at the same time.

4. Don't assume MTAs remember query results from one lookup to the next, so the necessary info should be available in the DNS cache. (If they do, it doesn't hurt.)

5. Be compatible with DNSSEC, but don't depend on DNSSEC.

6. Don't attempt to be compatible with existing DNSBLs. I expect these lists will mostly be served from something like rbldnsd, although it would also be possible to have an external program create the zone files and serve them from a conventional DNS server such as BIND.

---

3. DNS record format

TOC

Each DNS record is a blob of bytes containing multiple self-describing logical entries, each of which is a an IP address prefix, which might be a single IP if the length of the prefix is 128 (IPv6) or 32 (IPv4). In each blob, the prefixes are in address order from lowest to highest. The address ranges of prefixes cannot overlap. The name of each record is the address entry that implicitly points to it, which is the entry immediately preceding it. In this context, there is no advantage to doing rDNS-style nibble reversed naming, so the name is just 32 ASCII characters for an IPv6 DNSxL or 8 characters for an IPv4 DNSxL, the address as a hex number. One blob is the tree root, which is named with all zeros, such as 00000000000000000000000000000000.dnsxl.example or 00000000.dnsxl.example.

The simplest way to represent a blob is as a TXT record, with all the strings concatenated. The more entries that fit in a blob, the better this will work, so although it would be possible for each logical entry to be a separate string in the TXT record, it doesn't to do that.

Each blob contains a set of the prefixes that are in the DNSxL. For blobs that are not leaves (identified by a per-blob flag, described below), the the entries in the blob also partition the address space, with the prefixes between the ones in the current blob in sub-blobs, each named by the entries in the current blob.

To minimize edge cases, the root blob always contains the lowest and highest entries.

---

4. Lookup algorithm

TOC

A client looks up an IP address in the DNSxL as follows:

1. Make the root blob the current blob and fetch it.

2. Search through the current blob for a prefix entry that contains the target IP address. If you find it, stop, the DNSxL contains the IP.

3. If the IP address is lower than the first entry in the current blob, or higher than the last entry, stop, the DNSxL does not contain the IP.

4. If this is a leaf blob, stop, the DNSxL does not contain the IP.

5. Find the entry in the current blob that is just below the IP. Use the address in that entry as the name of new blob, fetch that blob, and make it the current one.

6. Go to Paragraph 2.

It should be evident that this is analogous to a binary tree search, except that each node has a lot more than two descendants.

---

5. Details of blob representation

TOC

The first byte of each blob is a flag:

```
L P P P P P P P
```

The L bit is set if this is a leaf. The PPPPPPP is the implicit prefix size. If all of the addresses in the blob have the same initial bits as the name of the blob does, which is fairly likely due to the way that IPv6 addresses are allocated, this is the number of common initial bits. The common prefix bits are not stored in the blob's entries, but are logically prefixed to each address in the blob.
After that is some number of prefixes:

```
X S S S S S S S
Address
```

X is reserved for now. SSSSSSS is 0-127 (IPv6) or 0-31 (IPv4), the prefix size minus one. The Address is 128-P-S or 32-P-S bits long, rounded up to a byte. That is, it omits the common prefix bits which

are obtained from the name of the blob, and the suffix bits beyond the prefix size.
For example, say an entry is 2001:1234:5678:9ABC::/64, and the common prefix size is 16. Then the entry would be (in hex)

```
3f 12 34 56 78 9A BC
```

The 3f is the prefix size (64-1), the 2001 is omitted since that's the prefix, and the rest is the address.
Each blob is as big as will fit in a DNS answer. If you don't believe in EDNS0, the limit is about 450 bytes. If you do believe in EDNS0, it's whatever size you think clients ask for, probably in the 2K to 4K range.

---

6. Building and updating DNSBLs

TOC

[[Note: This section is somewhat speculative. I have experience with disk-based B-trees, but haven't implemented any of this yet.]]
DNSxLs are typically compiled as a list of prefixes and lengths. They must be turned into a tree of named blobs before being served in the DNS. The technique varies a little depending on whether the tree will be updated incrementally, or rebuilt from scratch if it changes.

6.1. Building static DNSxLs

TOC

A static DNSxL should have the minimum number of blobs, each of which should be as full as possible. The technique to build the tree is a direct adaptation of the B-tree building technique in [WPBTREE] (Wikipedia, “B-tree,” December 2010.).
Start with a sorted list of prefix entries. Save one entry for the next pass, then take as many entries as possible and make a blob out of them. Repeat saving an entry and creating a blob until all entries are used. These will be the leaf blobs. Now, take the list of saved entries and repeat to create the blobs at the next level up. Keep repeating to create each level of the tree until the process creates only one blob. Insert the one saved entry into that blob which will be the root. That might make the blob overflow, in which case split it in half and move the first, middle, and last entries into a new root blob.
When the list changes, rebuild it from scratch.

TOC

6.2. Building and updating dynamic DNSxLs

One of the reasons that B-trees are so widely used is that it is possible to update them efficiently without rebuilding them. The same should apply here.

The general approach to updates is to add or delete an entry, then if that makes a blob too big or makes it empty, rebalance the tree to fix the problem. If a blob is too big, move entries into an adjacent blob if possible, otherwise split the blob. This will require updating the blob above, which in turn might overflow if the update involves adding rather than replacing an entry. (It might overflow even with a replacement if it makes the compressible prefix shorter.) In that case, repeat the process, potentially all the way to the root. When deleting an entry, if the blob becomes empty, move its pointer entry from the blob above up into one of the adjacent blobs, then adjust the upper blob as needed. Again, this might cause overflow in which case move entries between blobs or split the full one.

A tree in which each blob is completely full is quite expensive to update. The first insertion will cause a leaf to overflow, with overflows rippling all way up the tree to the root. It would probably be a good idea when building a list that is intended to be updated to leave some slack space in each blob, to limit the ripple effect from changes.

A significant difference between this design and a conventional B-tree is the version skew due to DNS caching. In a normal B-tree, the pages (blobs) are locked while being changed, and the changes are immediately visible to all the clients. In this case, the clients cache each blob for the DNS TTL. If updates change the entries in non-leaf blobs, they will break the links between blobs since they use the entries as pointers. A possible band-aid is to add temporary CNAME records at the former names pointing to the closest new name, so most (admittedly not all) of the entries can still be located. Once the TTL on the old blobs has expired, the CNAMEs can be deleted.

---

7. Estimated performance

TOC

The size of entries varies depending on the length of the prefixes and the amount of common prefix compression. A /64 with no common prefix would take 9 bytes, so I'll use 10 bytes as an estimate of average entry size. With EDNS0 and 4K records, that would allow 400 entries per blob. A two-level tree could hold 160,000 entries, a three level tree 64 million entries, which would need 160,000 blobs. Large v4 DNSBLs like the CBL have about seven million entries now, so this should be adequate. If blobs have to fit in 512 byte responses, that would be about 40 entries per blob. A five-level tree could hold 100 million entries in about 2.5 million blobs, still adequate.

The number of queries for any particular lookup is the number of levels, which is unlikely to be more than five in a DNSxL of plausible size. The cache behavior obviously depends on both the layout of the entries and the query pattern, but this design avoids some obvious worst cases. If a /64 is either entirely listed, not listed at all, or just has a single /128 listed, all queries for addresses in that /64 will refetch the same four or five records. If a large range of addresses is either listed in one prefix, or not listed at all, all queries will refetch the same set of blobs, which would be likely to be cached.

The total number of DNS records used is always less than the number of records for a traditional entry-per-IP DNSxL for the same set of entries. Since all the DNS queries are made by following the tree of entries, clients shouldn't make queries that fail, so there will be no negative cache entries. (This isn't quite true due to version skew in updated DNSxLs, but it's hard to imagine a plausible scenario in which there would be a lot of different failing queries.) This suggests that the overall cache behavior will be no worse than, and quite possibly much better than the behavior of traditional IPv4 DNSxLs.

---

8. Security considerations

TOC

Semantically, there is little difference between a DNSxL published using this scheme and one published using the traditional entry per IP approach, since both publish the operator's opinion about some subset of the IP address space.

One significant practical difference is that it is much easier for clients to obtain copies of all or part of the database. For a traditional DNSxL, the only way to determine its contents is to query the entire address space (or at least the active part of it) one address at a time, which would require several billion queries for IPv4, and is deterred by rate limiting the queries. In this scheme, the names of all of the DNS records are easy for clients to determine, so they can efficiently walk the tree. While rate limiting is possible, it is less effective since clients fetch more data with each query. It is also easy for a client to fetch all the entries for a particular IP range, such as the range of a network the client controls to see what parts of it are blacklisted.

---

9. Topics for further consideration

TOC

* Conventional IPv4 DNSBLs generally can return an A record or a TXT record. The A record often has information coded in the low byte or two that identifies the type of listing, or which of

several sub-lists the entry is on. If it's important to return a byte or two of result, it would be straightforward to add the byte or two to each entry, at a modest increase in entry size and hence some loss in performance. The TXT records are typically either all the same, or there's one per A value, perhaps with the IP address interpolated into the string to provide a URL to look up. Those strings could be constructed in client libraries, with templates stored in the DNS, e.g. the string for code 42 might be in a TXT record at 42._strings.dnsxl.example.

*There might be better ways to do prefix compression, e.g., a per-entry field that says how many bits are the same as the previous entry. Entries in blobs could be bit rather than byte aligned, although I expect that would be a lot more work for minimal extra compression. There may be clever tricks to allocate entries into blobs to maximize the size of the prefix in each blob. If a blob consists entirely of /128's it might be worth a special case, leaving out the length byte on each entry.

*When adding a new entry to a full leaf blob, another possibility would be to make the blob a non-leaf, and create two new leaves below it. This would make updates faster and less disruptive, at the cost of possibly slower lookups since some parts of the tree will be deeper. Perhaps a hybrid approach would make sense, rebuild or rebalance the tree when it gets too ragged, with more than a one-level depth difference between the deepest and shallowest leaves.

---

10. IANA considerations

TOC

This document makes no requests to IANA. All data are stored and queried using existing DNS record types and operations.

---

11. References

TOC

11.1. References - Normative

TOC

[RFC1034]	Mockapetris, P., “Domain names - concepts and facilities,” STD 13, RFC 1034, November 1987 (TXT).
[RFC1035]	

	Mockapetris, P., “Domain names - implementation and specification,” STD 13, RFC 1035, November 1987 (TXT).

11.2. References - Informative

TOC

[KNUTHV3]	Knuth, D., “The Art of Computer Programming: Volume 3, Sorting and Searching,” 1998.
[RBLDNSD]	Tokarev, M., “rbldnsd: Small Daemon for DNSBLs.”
[RFC2308]	Andrews, M., “Negative Caching of DNS Queries (DNS NCACHE),” RFC 2308, March 1998 (TXT, HTML, XML).
[RFC4862]	Thomson, S., Narten, T., and T. Jinmei, “IPv6 Stateless Address Autoconfiguration,” RFC 4862, September 2007 (TXT).
[RFC5782]	Levine, J., “DNS Blacklists and Whitelists,” RFC 5782, February 2010 (TXT).
[WPBTREE]	Wikipedia, “B-tree,” December 2010.

Appendix A. Change Log

TOC

NOTE TO RFC EDITOR: This section may be removed upon publication of this document as an RFC.

A.1. Changes from -00 to -01

TOC

Change CIDRs to prefixes. Allow for IPv4 addresses.
Add possible updates producing unbalanced trees.

Author's Address

TOC

	John Levine
	Taughannock Networks
	PO Box 727
	Trumansburg, NY 14886
Phone:	+1 831 480 2300
Email:	standards@taugh.com
URI:	http://jl.ly