Internet Research Task Force
Internet-Draft
Intended status: Experimental
Expires: September 15, 2011

J. Levine
Taughannock Networks

March 14, 2011

An efficient method to publish ranges of IP addresses in the DNS
draft-levine-iprangepub-02

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 September 15, 2011.

Copyright Notice

Table of Contents

1. Introduction

For many years, the Domain Name System[RFC1034] [RFC1035] 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]. 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 [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 DNSxL organized like an IPv4 DNSxL with a record per address could use wildcards or a specialized server such as rbldnsd [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 DNSxL'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 DNSxL lists the query's address, or a negative entry[RFC2308] 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 DNSxL query traffic will both flood the DNSxL'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 DNSxLs, so that, say, each /64 had a separate listing, and the queries only used the high 64 bits of each address. This arguably could limit the damage from DNSxL queries (although even a 64 address is plenty to swamp caches) 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]. The technique in this document stores ordered sets of IP address ranges in DNS records, using TXT records as a containers for binary data. The technique is a straightforward adaptation of the way that B-trees store ordered sets of key strings in disk blocks.

2. Assumptions and Goals

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

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

2. Handle overlapping ranges, and exception entries. (It is typical for a single instance of rbldnsd to serve copies of several DNSxLs and to return answers from all of the DNSxLs that have entries matching a query.) Each range can be tagged with an entry type, and the answer includes the tag.

3. 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 the DNSxL uses. I assume that client->cache queries are cheap, while cache->server queries are much more expensive, so it is a worthwhile tradeoff to increase the number of the former to decrease the number of the latter.

4. Don't assume senders will be well behaved. In particular, spammers may use a unique 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.

5. 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.)

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

7. Don't attempt to be compatible with existing DNSxLs at the query level, but do provide a client API similar to the one for existing DNSxLs. 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

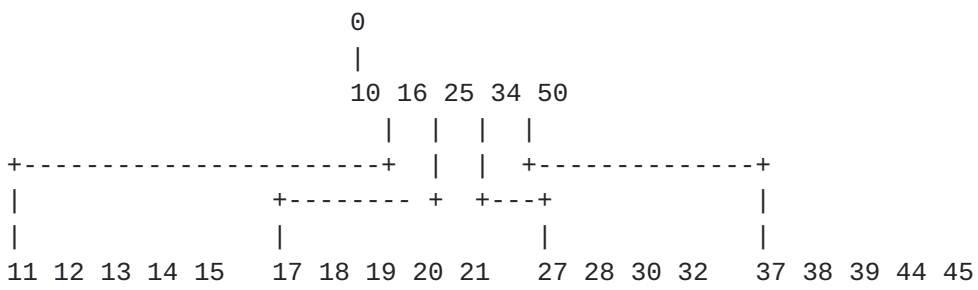
from a conventional DNS server such as BIND.

3. B-tree data structure

The goal is to see if a particular address is in one of the ranges. If we could retrieve the whole list in one query, it'd be easy to do a binary search of the list, but any useful DNSxL is much too big to store in one block. So a b-tree breaks the list into a tree of blocks, so you can do the binary search block by block.

As an example, the figure below shows simple two-level b-tree. It's a tree of blocks, where each block contains an ordered set of values. The number of ranges can vary from one block to another, since I use a variable length encoding.) The example below is two levels; DNSxLs represented as B-trees turn out to need between 2 and 5 levels.

In each non-leaf record, there is conceptually a lower level record between each pair of entries. So in the example below, if you were looking for 29, you'd look in the top record, see it's not there but it's between 25 and 34, so you look at the record after 25, then see it's between 28 and 30. If this weren't a leaf, there'd be another record under 28, but since it is a leaf, you know you didn't find it.



In a conventional B-tree, each record is a disk block, and the pointers are explicitly stored in each record, typically as a disk block number. But this b-tree is stored in the DNS, where each record has a name. Rather than invent pseudo-block numbers to use as names, I use the entry that points to the block. So the lower blocks in the example above would be named 10, 16, 25, and 34. The root block is always named 0.

Since the entries in a DNSBL are actually ranges of IP addresses, each one is represented as the base address of the range, and a bitmask length. For an IPv4 DNSxL, the ranges are CIDR ranges, and the bitmask length is the CIDR size, e.g., in 192.168.40.0/23 the base address is 192.168.40.0 and the mask length is 23. If the mask length is the length of the address (32 for IPv4 or 128 for IPv6, the

range is a single address.

4. DNS record format

A DNSxL is logically an ordered list of of IP address ranges. The ranges are in address order from lowest to highest. If one range encloses another, the ranges are ordered by the base aaddress of each range. Ranges with the base same adddress are ordered from shortest to longest mask length.

Each DNS record is a block of bytes representing an ordered sub-list of IP address ranges, Each range also includes a byte saying what kind of entry it is, i.e., what value to return to a client, and a one-bit exception saying that this entry is an exception to an enclosing range with the same value.

Since each DNS record has to have a name, the name is a text version of the IP address of the range immediately preceding the first range in the record. 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 block is the root of the B-tree, which is named with all zeros, such as 00000000000000000000000000000000.dnsxl.example or 00000000.dnsxl.example.

In order to improve performance, each block representing a list of ranges is stored in the DNS in a binary form using prefix and suffix compression. The simplest way to store a block of arbitrary bytes in the DNS is as a TXT record, with all the strings in the record concatenated. The more entries that fit in a block, the better this will work, so to make the best use of space, each of the strings other than the last should be the maxmum length, 255 bytes. Note that despite its name, the contents of the strings in a TXT record can be arbitrary binary data, so although the text form of the records may look rather ugly in a master file, they will work fine in the DNS. (If this design catches on, it might be worth defining an RR type for it to avoid scaring people who expect TXT records to be readable text.)

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

To minimize edge cases, the root block always contains the lowest and highest entries. In other blocks, there can be sub-blocks between

each pair of ranges, but not before the first range or after the last. In other words, the last range in a block is not used as the name of a sub-block.

5. Handling enclosing ranges

Since one range can enclose another, and some ranges enclose exception ranges, a search has to find not just one entry that contains an address, but all the ranges that contain the address. In order to avoid having to walk up and down the tree of blocks, each block also includes copies any ranges that enclose the first range in the block. This ensures that a search of a block will return all the values for ranges within the block.

Note that the copied ranges have addresses equal or less to the name of the block's record, while the other ranges in the block have addresses greater than the name of the block. Hence it is easy to identify copied enclosing ranges, and they need not be specially marked in the encoded version of the block.

6. Lookup algorithm

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

1. Make the root block the current block and fetch it. Note that there are no matches found so far.

2. Search through the current block for the highest-addressed range entry that contains the target IP address. Also note any preceding ranges that also contain the target address. If there are amy matches, discard any previous set of matches, and remember these matches.

3. If the IP address is lower than the first range in the current block (disregarding copied ranges), or higher than the last range, stop.

4. If this is a leaf block, stop.

5. Find the range in the current block whose address is just below the target IP. Use the address of that range as the name of new block, fetch that block, and make it the current one.

6. Go to Paragraph 2.

The result of this algorithm is a possibly empty set of matches.

Some of the matches may be exceptions; if so delete both the exception and the preceding non-exception match with the same value. If there are any remaining matches, the address is present in the DNSxL, so return all of the values for the remaining matches. If not, the address is not present in the DNSxL.

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

7. Details of block representation

The first byte of each block is a flag byte, with the bits interpreted as follows:

L P P P P P P P

The L bit is set if this is a leaf. The PPPPPPP is a seven-bit number which is the implicit prefix size. If all of the addresses in the block have the same initial bits as the name of the block does, which is fairly likely since DNSxL entries often occur in clusters, the implicit prefix size is the number of common initial bits. The common prefix bits are not stored in the block's entries, but are logically prefixed to each address in the block. An implicit prefix size of zero means no implicit prefix.

After that is some number of ranges:

X S S S S S S S (one byte)
Value (one byte)
Address (zero to 16 bytes)

The high bit first byte (X) X is the eXception flag, and means that this range is an exception entry. SSSSSSS is 0-127 (IPv6) or 0-31 (IPv4), the mask size minus one. The value is one byte, the details of which are discussed below. The Address is 128-P-S or 32-P-S bits long, rounded up to a byte. That is, it omits the implicit prefix bits which are obtained from the name of the block, and the bits beyond the prefix size. For copied ranges, the mask size may be the same as or less than the implicit prefix size, in which case there are no address bytes in the entry, and the address bits are all taken from the implicit prefix.

For example, say an entry is 2001:0DB8:5678:9ABC::/64, the implicit prefix size is 16 bits, and the value is hex 42. Then the entry would be (in hex)

3f 42 0D B8 56 78 9A BC

The 3f is the prefix size (64-1), 42 is the the 2001 is omitted since that's the common prefix, and the rest is the address.

Each block should be the as large as can 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 about 4000 bytes.

7.1. Return values

DNSBLs traditionally can return A and TXT records. The A records are the range 127.x.x.x, the TXT records can be arbitrary text that may be the same for all records, or customized to some extent per record. In practice, it is rare for a DNSxL to use a large number of different values. Hence the values in this encoding are a single byte, used to look up the values to return.

To look up a return value, do a DNS lookup of a record whose name is the letter V followed by the two-digit hex representation of the value, e.g., V00.dnsxl.example, for value zero. Use its A and TXT records as the return value. The most common TXT record customization is to insert the looked-up address. In keeping with common software practice, a dollar sign in the TXT record is replaced by the text form of the IP address before returning it to the client application.

Note: this is not entirely satisfactory. Some DNSBLs are keyed to a listing database, and use a database record ID rather than the IP address to customize the TXT record. It might be necessary to invent some sort of macro expansion scheme, with the macros in the TXT record and the values to be inserted included in each range's encoded entry.

8. Building and updating DNSxLs

DNSxLs are compiled as a list of ranges (prefix and length), and values. They must be turned into a tree of named blocks 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.

8.1. Building static DNSxLs

A static DNSxL should have the minimum number of blocks, 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].

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 tentative block out of them. Repeat saving an entry and creating a block until all entries are used. These will be the leaf blocks. Now, take the list of saved entries and repeat to create tentative blocks at the next level up. To preserve the identity that there are no entries covered by a block's range before the first entry in the block or after the last entry, before creating each block, it's necessary to promote the first and last entries covered by the block into the block A way to do this is to recursively "borrow" the first from the first sub-block, which in turn borrows the last entry from its first sub-block, all the way down to the leaf. Then do the same recursive procedure for the last entry in the last sub-block. Keep repeating to create each level of the tree until the process creates only one block. Eventually, a pass will create a single block, which is the root.

Before encoding each block, insert copies of the ranges that enclose the first range in the block. A way to do this is to make a linear pass through the entire sorted list of ranges before starting to create the blocks, and each entry inside an enclosing range making a link to the entry that encloses it. (This can be done in linear time by keeping a stack of currently "open" enclosing ranges and linking to the range at the top of the stack.) Then when encoding the block, following the links from the first range in the block will find the enclosing ranges that need to be copied.

When the list changes, rebuild it from scratch.

8.2. 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 block too big or makes it empty, rebalance the tree to fix the problem. If a block is too big, move entries into an adjacent block if possible, otherwise split the block. This will require updating the block 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 block becomes empty, move its pointer entry from the block above up into one of the adjacent blocks, then adjust the upper block as needed. Again, this might cause overflow in which case move entries between blocks or split the full one.

A tree in which each block 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 block, to limit the ripple effect from changes. The enclosing ranges also need to be updated.

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 (blocks) are locked while being changed, and the changes are immediately visible to all the clients. In this case, the clients cache each block for the DNS TTL. If updates change the entries in non-leaf blocks, they will break the links between blocks 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 blocks has expired, the CNAMEs can be deleted.

9. Estimated performance

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 block. A two-level tree could hold 160,000 entries, a three level tree 64 million entries, which would need 160,000 blocks. Large v4 DNSxLs like the CBL have about seven million entries now, so this should be adequate. If blocks have to fit in 512 byte responses, that would be about 40 entries per block. A five-level tree could hold 100 million entries in about 2.5 million blocks, 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 blocks, 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.

Some preliminary tests using traces of real mail server connection data and 15 minute TTLs suggest that hit rates will be about 80% for DNSxLs that include individual addresses, and close to 100% for DNSxLs that include large ranges.

10. Security considerations

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.

11. Topics for further consideration

- 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 blocks 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 blocks to maximize the size of the prefix in each block. If a block 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 block, another possibility would be to make the block 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.

12. IANA considerations

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

13. References

13.1. References - Normative

[RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, November 1987.

[RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, November 1987.

13.2. References - Informative

[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.

[RFC4862] Thomson, S., Narten, T., and T. Jinmei, "IPv6 Stateless Address Autoconfiguration", RFC 4862, September 2007.

[RFC5782] Levine, J., "DNS Blacklists and Whitelists", RFC 5782, February 2010.

[WPBTREE] Wikipedia, "B-tree", December 2010.

Appendix A. Change Log

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

A.1. Changes from -01 to -02

Fix typos

A.2. Changes from -00 to -01

Change CIDRs to prefixes. Allow for IPv4 addresses.

Add possible updates producing unbalanced trees.

Author's Address

John Levine
Taughannock Networks
PO Box 727
Trumansburg, NY 14886

Phone: +1 831 480 2300
Email: standards@taugh.com
URI: http://jl.ly