IP Address Parsing for Humans

Jathan McCollum
SCaLE 21x - 2024-03-17
Agenda
Agenda you are here (1 min)
About Me & Intro (2 min)
What’s in a Network? (10 min)
Parsing Networks (5 min)
Compare & Contrast (20 min)
Bringing it Home (5 min)
This is my 2nd time* talking at SCaLE!

I love Python & networks

*https://youtu.be/7zZ9980X_bs
Core Dev for:
Trigger, NSoT, Nautobot

20+ years in InfraSec

#NetSecDevOps
In other words...

I love IPAM

(IP Address Management for those in the back)
Intro
IPAM is Hard

IPv4 is exhausted (except not really)
IPv6 is scary (but not really)
Spreadsheets make this infinitely worse
Manual allocation is the devil
Automation is the stairway to heaven
Before we jump in

All code is Python

>>> means we’re in a Python interactive shell

pip install is used to install libraries

This is dry content but we’ll have fun!
One Last Thing™

Lecture style talk so please ask questions

Leaving time at the end for demos and Q&A

Okay that was 2 things 3 if you count this one
What’s in a Network?
IP networks are trees

A 192.168.0.0
  └── B 192.168.1.0
      └── D 192.168.1.1
          └── E 192.168.1.2
     └── C 192.168.2.0
Tree Terminology

Ancestors are the parents of the parent

Parent is the direct parent network

Child is a direct child of a parent

Descendants are the children of children
Terminology

**Subnets** are descendent networks

**Supernets** are ancestor networks

**Prefixes** are networks (aka aggregates)

**IP addresses** are point locations on a network
Terminology (cont.)

**Prefix length** is the size of a prefix (in bits)

**Host addresses** are assigned to interfaces

**Gateway** is the entry point to a network

**Broadcast** is the end point of a network
Prefixes all the way down

Everything is a prefix

A host address is just the smallest prefix

A network contains other network prefixes

Subnet masks aren’t really used anymore
CIDR is the way

Classless Inter-Domain Routing

Prefix length is how you allocate & route

Larger number means smaller prefix (bits)

Zeros can be omitted or “compressed”
IPv4 anatomy

IPv4 address in dotted-decimal notation

172 . 16 . 254 . 1

\[
\begin{align*}
10101100 . 00010000 . 11111110 . 00000001 \\
\hline
\text{8 bits} \\
\hline
\text{32 bits (4 bytes)}
\end{align*}
\]
IPv4 networks

32-bit period-delimited; (4) 8-bit octets

All 0.0.0.0/0 (aka “quad zero”); or 0/0

Localhost 127.0.0.1/32

Link local 169.154/16

Private 10/8, 192.168/16, 172.16/12
IPv6 anatomy

An IPv6 address (in hexadecimal)

2001:0DB8:AC10:FE01:0000:0000:0000:0000

2001:0DB8:AC10:FE01::

Zeroes can be omitted

1000000000001:0000110110111000:10101100001000:1111110000000001:

00000000000000:00000000000000:00000000000000:00000000000000
IPv6 networks

128-bit colon-delimited; (8) 16-bit hextets

All ::/0 (aka “double colon zero”); or ::

Localhost ::1/128

Link local fe80::/10

Private fdaf:3c0f:118c:61ad::/64
IPv6 Networks (cont.)

IPv6 networks are *stupendously* large

/64 is the smallest prefix length you should use

/48 is the minimum size for BGP routing

Compressed form collapses zeros into ::
Compression

IPv4 trailing zeros can be omitted
192.168.0.0/24 -> 192.168/24
0.0.0.0/0 -> 0/0

IPv6 leading or consecutive zeros
21da:00d3:0000:0000:0000:0000:00ff:9c5a
   -> 21da:d3::ff:9c5a
fe80:0000:0000:0000:0000:0000:0000:0000/10
   -> fe80::/10
0000:0000:0000:0000:0000:0000:0000:0000/0
   -> ::/0
Parsing Networks
Pro Tips

**DO** use IP libraries to parse

**DO NOT** use spreadsheets; use IPAM

**DO NOT** cast IPv6 networks as lists (huge)

**DO NOT** use regex
DO NOT USE REGEX

Regex aka “regular expressions”

\d+.\d+.\d+.\d+ is “fine” for IPv4, but just no

There is nothing regular about IPv6

But if you insist…
J/K DO NOT DO THIS

"^\s((\[0-9A-Fa-f\]{1,4}:)\{7\}(\[0-9A-Fa-f\]{1,4}:)\{6\}:(\[0-9A-Fa-f\]{1,4}:)\{1,4\}):(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|1[0-9]?[0-9])\{3\}:1,2\})(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|1[0-9]?[0-9])\{3\}:1,2"}

Credit: http://stackoverflow.com/a/5567938/194311
import ipaddress

def parse_network(prefix):
    """Returns an IP network object or None if the prefix is invalid."""
    try:
        return ipaddress.ip_network(prefix)
    except ValueError:
        return None
Isn’t this nice?

```python
>>> parse_network("bogus")

>>> parse_network("192.168.0.0/24")
IPv4Network('192.168.0.0/24')

>>> parse_network("::/0")
IPv6Network('::/0')

>>> parse_network("fe80::/10")
IPv6Network('fe80::/10')

>>> parse_network("fe80::/10").version == 6
True
```
IPs are integers

>>> netaddr.IPAddress(0)
IPAddress('0.0.0.0')

>>> netaddr.IPAddress(0, version=6)
IPAddress('::')

>>> v4 = netaddr.IPNetwork("192.168.0.0/16")

>>> int(v4.ip)
3232235520

>>> int(netaddr.IPNetwork("fe80::1cbe:4216:28a4:ea7d"))
338288524927261089656090062382923311741
IPs are integers (cont.)

Binary integers represent each address

Bitwise math is used to calculate networks

This is out of scope for this talk!

See inet_pton, inet_ntop, et al. man(3) pages
One big IP family!

If IP $\geq$ your gateway address

And IP $\leq$ your broadcast address

Then IP is a member of the network

192.168.0.1 is a member of 192.168.0.0/24
Compare & Contrast
Library Overview

netaddr comprehensive IP manipulation

ipaddress bare bones IP manipulation

cidrize parses commonly used human inputs

ipparser simplifies parsing & DNS resolution
The Big Dogs

**netaddr** is the most advanced; but 3rd party

**ipaddress** is in the standard library

They share most features

**Low level** and missing user-friendly features
Common features
Membership (netaddr)

```python
>>> net = IPNetwork("192.168.0.0/24")

>>> "192.168.0.1" in net
True

>>> "192.168.1.1" in net
False

>>> IPAddress("192.168.0.1") > IPAddress("192.168.0.0")
True

>>> IPAddress("192.168.0.1") > IPAddress("192.168.0.255")
False
```
Sub/supernets (both)

```python
types = list(net2.subnets())
```

```python
types = list(net.subnet(25))
[IPNetwork('192.168.0.0/25'),
 IPNetwork('192.168.0.128/25')]
```

```python
types = net.supernet(23)
[IPNetwork('192.168.0.0/23')]
```

```python
types = net in IPNetwork("192.168.0.0/23")
True
```

```python
types = net2.supernet()
IPv4Network('192.168.0.0/23')
```

```python
types = net2 in ip_network("192.168.0.0/23")
False
```

```python
>>> net2.subnet_of(ip_network("192.168.0.0/23"))
True
```
Operations (both)

```python
>>> net[0]
IPAddress('192.168.0.0')

>>> next(net.iter_hosts())
IPAddress('192.168.0.1')

>>> net.broadcast
IPAddress('192.168.0.255')

>>> net.prefixlen
24

>>> net.size
256

>>> net2[0]
IPv4Address('192.168.0.0')

>>> next(net2.hosts())
IPv4Address('192.168.0.1')

>>> net2.broadcast_address
IPv4Address('192.168.0.255')

>>> net2.prefixlen
24

>>> net2.num_addresses
256
```
Other stuff (both)

```python
>>> net.is_
net.is_ipv4_compat()
net.is_loopback()
net.is_reserved()
net.is_ipv4_mapped()
net.is_multicast()
net.is_unicast()
net.is_link_local()
net.is_private()

>>> net.hostmask
IPAddress('0.0.0.255')

>>> net.version
4

>>> net2.is_
net2.is_global
net2.is_loopback
net2.is_private
net2.is_unspecified
net2.is_link_local
net2.is_multicast
net2.is_reserved

>>> net2.hostmask
IPv4Address('0.0.0.255')

>>> net2.version
4
```
netaddr
netaddr PROs

**Tons** of extra utilities

**MAC** (EUI) address support

**IPSet** object for doing advanced subnet math

**IPRange** object for calculating address ranges
netaddr CONs

**Slow** compared to *ipaddress*

**3rd party** so it must be installed using *pip*
IPNetwork objects

```python
>>> net = netaddr.IPNetwork("192.168.0.0/24")

>>> net.info
{'IPv4': [{'date': '1993-05',
            'designation': 'Administered by ARIN',
            'prefix': '192/8',
            'status': 'Legacy',
            'whois': 'whois.arin.net'}]}

>>> netaddr.IPNetwork("0/0")
IPNetwork('0.0.0.0/0')
```
IPAddress objects

>>> netaddr.IPAddress(0)
IPAddress('0.0.0.0')

>>> netaddr.IPAddress(0, version=6)
IPAddress('::')

>>> ip = netaddr.IPAddress("1.2.3.4")

>>> ip.info
{'IPv4': [{'date': '2010-01',
'designation': 'APNIC',
'prefix': '1/8',
'status': 'Allocated',
'whois': 'whois.apnic.net'}]}
>>> net_set = netaddr.IPSet(["192.168.0.0/24"])  

>>> slash26 = netaddr.IPSet(["192.168.0.128/26"])  

>>> net_diff = net_set - slash26  

>>> net_diff  
IPSet(['192.168.0.0/25', '192.168.0.192/26'])  

>>> net_diff ^ slash26  
IPSet(['192.168.0.0/24'])
IPRange objects

```python
>>> v4_range = netaddr.IPRange('192.168.0.1', '192.168.0.2')

>>> v4_range
IPRange('192.168.0.1', '192.168.0.2')

>>> list(v4_range)
[IPAddress('192.168.0.1'), IPAddress('192.168.0.2')]

>>> v6_range = netaddr.IPRange('fe80::1', 'fe80::2')

>>> list(v6_range)
[IPAddress('fe80::1'), IPAddress('fe80::2')]```
EUI (MAC) objects

```python
>>> mac = EUI('00-1B-77-49-54-FD')

>>> oui = mac.oui

>>> oui
OUI('00-1B-77')

>>> oui.registration().address
['Lot 8, Jalan Hi-Tech 2/3', 'Kulim Kedah 09000', 'MY']

>>> oui.registration().org
'Intel Corporate'
```
ipaddress
ipaddress PROs

In Python standard library since Python 3.3

Fast especially under large workloads

~3x faster than netaddr in my testing

Great for rapid prototyping and basic parsing
ipaddress CONs

**Basic** AF (but that is also a strength)

**Distinct classes** require you to be explicit
IPv4Address, IPv4Network, IPv6Address, IPv6Network

**Constructors** are limiting; can’t specify version
ip_address(), ip_network()
>>> ipaddress.ip_network("0/0")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    raise ValueError(f'\{address!r}\ does not appear to be an IPv4 or IPv6 network')
ValueError: '0/0' does not appear to be an IPv4 or IPv6 network

>>> ipaddress.ip_network("192.168.0.0/24")
IPv4Network('192.168.0.0/24')

>>> ipaddress.ip_network("fe80::/10")
IPv6Network('fe80::/10')
>>> ipaddress.ip_address(0)
IPv4Address('0.0.0.0')

>>> ipaddress.ip_address(0, version=6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: ip_address() got an unexpected keyword argument 'version'

>>> ipaddress.ip_address("1.2.3.4")
IPv4Address('1.2.3.4')

>>> ipaddress.ip_address("fe80::1")
IPv6Address('fe80::1')
Class objects

```python
>>> ipaddress.IPv4Address("192.168.0.1")
IPv4Address('192.168.0.1')

>>> ipaddress.IPv4Network("192.168.0.0/24")
IPv4Network('192.168.0.0/24')

>>> ipaddress.IPv6Address("fe80::1")
IPv6Address('fe80::1')

>>> ipaddress.IPv6Network("fe80::/10")
IPv6Network('fe80::/10')
```
>>> from ipaddress import collapse_addresses, summarize_network_range, IPv4Address, IPv4Network

>>> range = summarize_address_range(
    IPv4Address('192.0.2.0'), IPv4Address('192.0.2.128'))

>>> list(range)
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/32')]

>>> collapsed = collapse_addresses(
    [IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')])

>>> list(collapsed)
[IPv4Network('192.0.2.0/24')]
The Small Dogs

cidrize built on top of netaddr

ipparsen built on top of ipaddress

They share many features

High level adding user-friendly features
cidrize
cidrize PROs

Strict & loose parsing modes

Wildcards and bracket ranges

CLI utility included that is very handy

Returns IPNetwork objects (might be a con)
cidrize CONs

Slower than *ipparsen* because *netaddr*

Weird name from a weird guy

No URL or DNS support

Ranges limited to v4 addresses
Wildcards and more

>>> cidrize.cidrize("192.168.1.*")
[IPNetwork('192.168.1.0/24')]

>>> cidrize.cidrize("192.168.1.0-15")
[IPNetwork('192.168.1.0/28')]

>>> cidrize.cidrize("192.168.1.1[56]")
[IPNetwork('192.168.1.0/27')]

>>> cidrize.cidrize("any")
[IPNetwork('0.0.0.0/0')]

>>> cidrize.cidrize("::")
[IPNetwork('::/0')]
>>> cidrize.cidrize("192.168.1.*", strict=True)
[IPNetwork('192.168.1.0/24')]

>>> cidrize.cidrize("192.168.1.0-15", strict=True)
[IPNetwork('192.168.1.0/28')]

>>> cidrize.cidrize("192.168.1.1[56]", strict=True)
[IPNetwork('192.168.1.15/32'), IPNetwork('192.168.1.16/32')]

>>> cidrize.cidrize("192.168.0.254-192.168.1.3")
[IPNetwork('192.168.0.0/23')]

>>> cidrize.cidrize("192.168.0.254-192.168.1.3", strict=True)
[IPNetwork('192.168.0.254/31'), IPNetwork('192.168.1.0/30')]
$ cidr -v 192.160.0.0/24

Information for 192.160.0.0/24

IP Version: 4
Spanning CIDR: 192.160.0.0/24
Block Start/Network: 192.160.0.0
1st host: 192.160.0.1
Gateway: 192.160.0.254
Block End/Broadcast: 192.160.0.255
DQ Mask: 255.255.255.0
Cisco ACL Mask: 0.0.0.255
# of hosts: 254
Explicit CIDR blocks: 192.160.0.0/24
ipparser PROs

Very fast because `ipaddress`

Returns strings but this might be a con, too?

DNS, URL, port support with `resolve=True`

Nmap XML report parser
**ipparser CONs**

Only returns **IPs** and not networks

Advanced ranges are passed through

Ranges also limited to v4 addresses

Invalid inputs don’t raise errors :(  


>>> ipparser.ipparser("socallinuxexpo.org", resolve=True)
['23.21.71.118']

>>> ipparser.ipparser("socallinuxexpo.org,google.com", resolve=True)
['23.21.71.118', '142.250.176.14']

>>> ipparser.ipparser("192.168.1.0-15")

>>> ipparser.ipparser("192.168.0.254-192.168.1.3")
['192.168.0.254-192.168.1.3']
Bringing it Home
When to use netaddr

For advanced applications

Calculating subnet allocations with IPSets

Managing MAC addresses

Leveraging metadata such as designation
When to use ipaddress

When performance matters

For common use-cases

Quick & dirty solutions

If you can’t use 3rd party libraries
When to use cidrize

Parsing user inputs in apps and utilities

When you need max flexibility in user inputs

If you need a CLI utility out of the box
When to use ipparser

When performance matters

Parsing user input AND performance required

DNS resolution is required

Nmap support is required
Thank You!
Pick your poison

netaddr
https://netaddr.readthedocs.io/

ipaddress
https://docs.python.org/3/library/ipaddress.html

cidrize
https://cidrize.readthedocs.io/

ipparser
https://github.com/m8sec/ipparser
Stay in Touch

@jathanism

on ALL THE THINGS