connect() - why you so slow?
Frederick Lawler
Systems Engineer @ Cloudflare

- security_create_user_ns()
- CVE-2022-47929: traffic control noqueue no problem?
- pci_(alert|crit|dbg|emerg|err|info|notice|warn) printk macros
50k egress unicast connections to a single destination... Who does that?
CDN request flow for uncached assets
Who does that?

```bash
$ sysctl net.ipv4.ip_local_port_range
net.ipv4.ip_local_port_range = 9024 65535
```
bind() before connect()

sk = socket(AF_INET, SOCK_STREAM)
sk.setsockopt(IPPROTO_IP, IP_BIND_ADDRESS_NO_PORT, 1)
sk.bind((src_ip, 0))
sk.connect((dest_ip, dest_port))

How to stop running out of ephemeral ports and start to love long-lived connections
Who does that?
2 IPv4 addresses for this service
Who does that?

tcp_v4_connect() func latency 2 IPv4 address

- tcp_v4_connect() latency per connection for 2 IPv4 addresses

Latency per connection in nanoseconds:
- 10000 -> 999999
- 1000000 -> 9999999
- 100000000 -> 999999999

Number of connections
1 IPv4 addresses for this service
Who does that?

IPv4 sales data. Source: Hilco Streambank.
Who does that?

tcp_v4_connect() func latency 1 IPv4 address

Latency per connection in nanoseconds

Number of connections
Who does that?
tcp_v4_connect() func latency 3 IPv4 address (for fun)

tcp_v4_connect() latency per connection for 3 IPv4 addresses

Latency per connection in nanoseconds

<table>
<thead>
<tr>
<th>Latency</th>
<th>Number of connections</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000 -&gt; 99999</td>
<td>0</td>
</tr>
<tr>
<td>100000 -&gt; 99999999</td>
<td>20,000</td>
</tr>
<tr>
<td>100000000 -&gt; 999999999</td>
<td>60,000</td>
</tr>
<tr>
<td>1000000000 -&gt; 9999999999</td>
<td>80,000</td>
</tr>
</tbody>
</table>
Who does that?

This is fine for most workloads, but for Cloudflare...

- Customers largely still leverage IPv4
- Similar performance with 1 IPv4’s as we’d see with 3
- Leverage our infrastructure to lazily hand off excess connections ie. fail fast
Time to investigate: TCP connect() why you so slow?
Time to investigate: TCP connect() why you so slow?
inet_hash_connect() overview

- Called in both TCP IPv4 & IPv6 contexts; but we’ll be focusing on IPv4
- We assume the kernel has to pick a port
offset &= ~1U;

other_parity_scan:
    port = low + offset;
    for (i = 0; i < remaining; i += 2, port += 2) {
        if (unlikely(port >= high))
            port -= remaining;

        inet_bind_bucket_for_each(tb, &head->chain) {
            if (inet_bind_bucket_match(tb, net, port, l3mdev)) {
                if (!check_established(death_row, sk, port, &tw))
                    goto ok;
                goto next_port;
            }
        }
    }

offset++;
if ((offset & 1) && remaining > 1)
    goto other_parity_scan;

net/ipv4/inet_hashtables.c:__inet_hash_connect
__inet_hash_connect() overview: initial port selection

```c
offset &= ~1U;
other_parity_scan:
    port = low + offset;
    for (i = 0; i < remaining; i += 2, port += 2) {
        if (unlikely(port >= high))
            port -= remaining;
        inet_bind_bucket_for_each(tb, &head->chain) {
            if (inet_bind_bucket_match(tb, net, port, l3mdev)) {
                if (!check_established(death_row, sk, port, &tw))
                    goto ok;
            }
        }
    }
offset++;
if ((offset & 1) && remaining > 1)
    goto other_parity_scan;
```

- Offset is randomly generated
- Offset is set to an even number
- Picked port is either “even” or “odd” based on net.ipv4.ip_local_port_range’s low port eg. 9024

Time to investigate: TCP connect() why you so slow?
**Are we starting our loop at the same offset each connect()?**

<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>net: Compute protocol sequence numbers and fragment IDs using MD5. introduced <code>secure_ipv4_port_ephemeral()</code> with md5 hashing</td>
</tr>
<tr>
<td>2017</td>
<td>tcp: change source port randomization at connect() time table_perturb introduced for more randomization + fingerprint mitigation</td>
</tr>
<tr>
<td>2021</td>
<td>secure_seq: use SipHash in place of MD5</td>
</tr>
<tr>
<td>2022</td>
<td>tcp: resalt the secret every 10 seconds</td>
</tr>
</tbody>
</table>
Are we starting our loop at the same offset each connect()?

```bash
#!/usr/bin/env bpftrace

/*
 // net/ipv4/inet_hashtables.c:__inet_hash_connect
 <+211>:    and edx,0xfffffffe          // offset &= ~1U;
 <+214>:    mov DWORD PTR [rsp+0x28],edx
 // other_parity_scan:
 <+218>:    add r14d,DWORD PTR [rsp+0x28]  // port = low + offset;
 <+223>:    test r13d,r13d
*/

kprobe::__inet_hash_connect+223 { 
    $port = reg("r14");
    @port_buckets = lhist($port, 9024, 65535, 10000);
}
```

Time to investigate: TCP connect() why you so slow?
Are we starting our loop at the same offset each `connect()`?

Time to investigate: TCP `connect()` why you so slow?
offset &= ~1U;

other_parity_scan:
    port = low + offset;
    for (i = 0; i < remaining; i += 2, port += 2) {
        if (unlikely(port >= high))
            port -= remaining;
        inet_bind_bucket_for_each(tb, &head->chain) {
            if (inet_bind_bucket_match(tb, net, port, l3mdev)) {
                if (!check_established(death_row, sk, port, &tw))
                    goto ok;
                goto next_port;
            }
        }
    }

offset++;
if ((offset & 1) && remaining > 1)
    goto other_parity_scan;

- Check if the socket is unique
- check_established() == __inet_check_established()
Is \texttt{__inet_check_established()} a problem?

- Tested benchmarks on a quiet virtual machine
- No other connections were established for the same src/dest ip + dest port
- Therefore, negligible impact
- Bind buckets will fill up eventually!

\textit{The quantum state of a TCP port}
offset &= ~1U;

other_parity_scan:
    port = low + offset;
    for (i = 0; i < remaining; i += 2, port += 2) {
        if (unlikely(port >= high))
            port -= remaining;
        inet_bind_bucket_for_each(tb, &head->chain) {
            if (inet_bind_bucket_match(tb, net, port, l3mdev)) {
                if (!check_established(death_row, sk, port, &tw))
                    goto ok;
                goto next_port;
            }
        }
    }

offset++;
if ((offset & 1) && remaining > 1)
goto other_parity_scan;

• Loop through first half of the ephemeral range then second
• Every other port is tested in sequence
Time to investigate: TCP connect() why you so slow?

__inet_hash_connect() overview: the loop

#sk ≤ N/2

∅ 0 1 2 3 4 5 6 7

sk₀ 0 1 2 3 4 5 6 7

sk₁ 0 1 2 3 4 5 6 7

sk₂ 0 1 2 3 4 5 6 7

#sk > N/2

$\frac{1}{2}$ 0 1 2 3 4 5 6 7

sk₄ 0 1 2 3 4 5 6 7

sk₅ 0 1 2 3 4 5 6 7

sk₆ 0 1 2 3 4 5 6 7

sk₇ 0 1 2 3 4 5 6 7
Is the loop the problem?

- Via experimentation
- Counted the even ports green, odd ports red
- Our port range dictates we always loop through even ports first
Tracking port parity switches

#!/usr/bin/env bpftrace

kretfunc:vmlinux:inet_hash_connect /retval == 0/ {
    $port = args->sk->__sk_common.skc_num;
    @procs[comm,cgroup] += $port & 1;
}

rate(connect_port_parity_switches_total[1m])

Prometheus exporter for eBPF metrics
Our conclusion

- Exhausting half the net.ipv4.ip_local_port_range is fast
- The port looping appears to be our primary bottleneck
- Evidenced by a previous attempt [PATCH] tcp: avoid unnecessary loop if even ports are used up and was not merged
Some feasible, but not viable solutions for our case

1. Split egress unicast connections over 2..N IP addresses
2. Introduce a sysctl to manipulate connect
3. Pick a random port in userspace, and bind() with that
4. Leverage the new IP_LOCAL_PORT_RANGE socket option (v6.3.y)*
Split egress unicast connections over 2..N IP addresses

- Leaks networking configuration to user space
- No ability to tell the interface to balance between assigned IP’s or IP blocks
- Requires IP_BIND_ADDRESS_NO_PORT socket option + bind() before connect() pattern
- We do this strategy now, but want to reduce to 1 IP
Introduce a sysctl to manipulate connect

- Kernel modification
- [PATCH] tcp: avoid unnecessary loop if even ports are used up
Pick a random port in userspace, and bind() before connect()

- Requires bind() before connect()
- Syscall overhead and ~8-12 attempts per connect closer to exhaustion
- Good up to ~70-80% port range utilization

```python
sys = get_ip_local_port_range()
estab = 0
i = sys.hi
while i >= 0:
    if estab >= sys.hi:
        break
    random_port = random.randint(sys.lo, sys.hi)
    connection = attempt_connect(random_port)
    if connection is None:
        i += 1
        continue
    i -= 1
    estab += 1
```
Leverage the new IP_LOCAL_PORT_RANGE socket option (v6.3.y)

- Max # of connect() as range
- Pre-allocation of partitions to balance between
- Loop problem still persists

What do?

5k window @ 1.1 sec

- connection attempts: 5000 errored: 0 total: 5000
total time: 1.113s
latency min: 0.075ms max: 3.294ms avg: 0.224ms
even ports: 2500 latency min: 0.075ms max: 3.294ms avg: 0.125ms
odd ports: 2500 latency min: 0.231ms max: 1.054ms avg: 0.323ms
errored ports: 0 latency min: infms max: 0ms avg: 0.0ms

- linear regression
Leverage the new IP_LOCAL_PORT_RANGE socket option (v6.3.y)

- Lower range works better
- Overlapping ranges is possible
- Overlap is determined by implementation

What do?

1k window @ 0.1 sec
Leverage the new IP_LOCAL_PORT_RANGE socket option (v6.3.y) + random offset

What do?

connection attempts: 56512 erred: 0 total: 56512
total time: 130.494s
latency min: 0.011ms max: 23.551ms avg: 2.309ms
even ports: 28256 latency min: 0.011ms max: 1.416ms avg: 0.025ms
odd ports: 28256 latency min: 2.697ms max: 23.551ms avg: 4.593ms
erred ports: 0 latency min: 0ms max: 0ms avg: 0ms

2.1 min → 1.8 sec!

connection attempts: 56512 erred: 868 total: 55644
total time: 1.875s
latency min: 0.011ms max: 2.884ms avg: 0.033ms
even ports: 27843 latency min: 0.011ms max: 2.884ms avg: 0.033ms
odd ports: 27801 latency min: 0.011ms max: 1.161ms avg: 0.031ms
erred ports: 868 latency min: 0.068ms max: 0.759ms avg: 0.209ms
What do?

Implementation details

sys.lo = 9024;  sys.hi = 65535
Implementation details

window.lo = 0;  window.hi = 500
range = window.hi - window.lo
offset = randint(sys.lo, sys.hi - range)
window.lo = offset;  window.hi = offset + range
setsockopt(SOL_IP, IP_LOCAL_PORT_RANGE, window.lo | (window.hi << 16))
Implementation details

- Overlap is OK
- Reattempts may be necessary depending on use case
- Larger net.ipv4.ip_local_port_range is better with smaller selection window
In summary

- Leverages a random port offset + random low port in range to be even or odd
- Allows kernel to perform loop over a small + configurable local port range
- Overlaps windows on top of another

2.1 min → 1.8 sec @ 56k connections 500 window

<table>
<thead>
<tr>
<th>Type</th>
<th>Latency (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>even ports</td>
<td>0.011 ms max: 2.884 ms avg: 0.033 ms</td>
</tr>
<tr>
<td>odd ports</td>
<td>0.011 ms max: 2.884 ms avg: 0.031 ms</td>
</tr>
<tr>
<td>errored ports</td>
<td>0.011 ms max: 1.161 ms avg: 0.031 ms</td>
</tr>
<tr>
<td>Total time</td>
<td>1.875 s</td>
</tr>
</tbody>
</table>

connection attempts: 56512 errored: 868 total: 55644

500 window
Performance 56k unicast egress connections

What do?

2.1 min → 1.8 sec

500 window

connection attempts: 56512 errored: 868 total: 55644
total time: 1.875s
latency min: 0.011ms max: 2.884ms avg: 0.033ms
even ports: 27843 latency min: 0.011ms max: 2.884ms avg: 0.03ms
odd ports: 27601 latency min: 0.011ms max: 1.161ms avg: 0.031ms
errored ports: 868 latency min: 0.068ms max: 0.759ms avg: 0.209ms

2.1 min → 2.0 sec

1000 window

connection attempts: 56512 errored: 1129 total: 55383
total time: 2.073s
latency min: 0.012ms max: 0.935ms avg: 0.037ms
even ports: 27685 latency min: 0.012ms max: 0.771ms avg: 0.031ms
odd ports: 27828 latency min: 0.012ms max: 0.755ms avg: 0.03ms
errored ports: 1129 latency min: 0.144ms max: 0.935ms avg: 0.324ms
Performance 56k unicast egress connections

2.1 min → 6.7 sec

5k window

connection attempts: 56512 errored: 4037 total: 52475
total time: 6.723s
latency min: 0.011ms max: 14.158ms avg: 0.119ms
even ports: 26281 latency min: 0.011ms max: 3.687ms avg: 0.036ms
odd ports: 26194 latency min: 0.012ms max: 14.158ms avg: 0.04ms
errored ports: 4037 latency min: 0.639ms max: 7.368ms avg: 1.174ms

What do?

2.1 min → 17.7 sec

10k window

connection attempts: 56512 errored: 6695 total: 49817
total time: 17.749s
latency min: 0.012ms max: 39.609ms avg: 0.314ms
even ports: 24880 latency min: 0.012ms max: 6.345ms avg: 0.058ms
odd ports: 24937 latency min: 0.012ms max: 17.104ms avg: 0.06ms
errored ports: 6695 latency min: 1.258ms max: 39.609ms avg: 2.212ms
Leverage the new IP_LOCAL_PORT_RANGE socket option (v6.8.y)

connection attempts: 56512 errored: 0 total: 56512
total time: 130.494s
latency min: 0.011ms max: 23.551ms avg: 2.309ms
even ports: 28256 latency min: 0.011ms max: 1.416ms avg: 0.025ms
odd ports: 28256 latency min: 2.697ms max: 23.551ms avg: 4.593ms
errored ports: 0 latency min: infms max: 0ms avg: 0.0ms

What do?

2.1 min → 1.6 sec!
New in Linux 6.8.y

- Just requires IP_LOCAL_PORT_RANGE from userspace
- Faster performance with other 6.8.y features
- Guaranteed to find a port
- Patch: tcp/dccp: change source port selection at connect() time
>= Linux 6.8.y implementation

IP_BIND_ADDRESS_NO_PORT = 24
IP_LOCAL_PORT_RANGE = 51

sys = get_local_port_range()
sk = socket(AF_INET, SOCK_STREAM)

range = pack("@I", sys.lo | (sys.hi << 16))

sk.setsockopt(IPPROTO_IP, IP_LOCAL_PORT_RANGE, range)

sk.bind((src_ip, 0))
sk.connect((dest_ip, dest_port))
What about UDP?
Completely different algorithm!

- Still uses a tight loop
- Does not check one half of the range, then the next
- A port is randomly picked, a loop increments that port by a fixed-random number until integer overflow back to original port
- then, increment port by 1 and repeat until port is found

What about UDP?
Takeaways

- Current implementation guarantees a port is selected
- Current implementation is not great at extreme egress workloads
- Random offset + 500-1k window coupled with kernel random port picking ensures we start looping at both odd and even ports with small-N
- Backport patches or update to at least 6.8.y
- Purely user space implementation
Questions?

✉️ fred@cloudflare.com
📖 connect() - why are you so slow?
🔗 Minimal code that generated the charts