Replacing xt_qtaguid with an upstream eBPF implementation

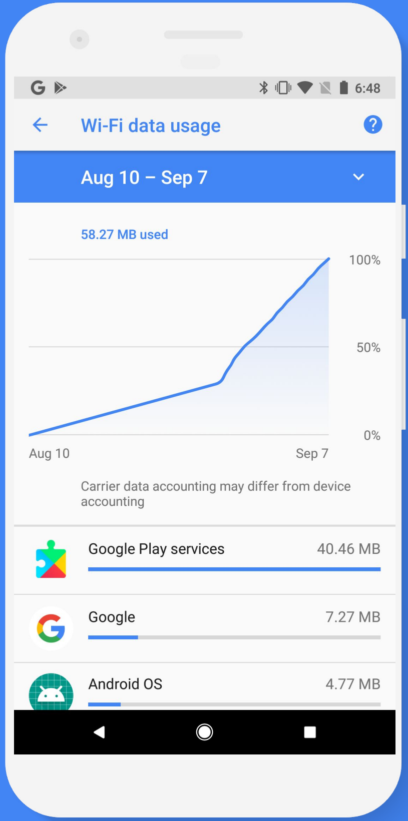
Linux Plumbers Android microconference
September, 2017

Lorenzo Colitti <lorenzo@google.com>
Chenbo Feng <fengc@google.com>

Background information

What is xt_qtaguid?

- Network traffic monitoring tool on Android devices
- Replaced the xt_owner module inside android device kernels
- Counting packet against the correct app uid.
- Filtering per-app traffic with socket owner match



Wi-Fi data usage

Aug 10 – Sep 7

58.27 MB used

Carrier data accounting may differ from device accounting

Google Play services 40.46 MB

Google 7.27 MB

Android OS 4.77 MB

Xt_qtaguid module

Problems with current module

  - Totally out of linux kernel tree and not upstreamable.
  - The version of this module varies with kernel version.
  - Stability, maintenance, and soon performance issues.

Goal

- developing a new tool to realize similar function as xt_qtaguid module with no out-of-tree code

Android socket tagging

- Semantics:
  - Counts packets and bytes on combination of app, app-defined tag, interface
  - Allows assigning 64-bit tag to every socket
    - Socket tags comprised of 32 bits UID (i.e., app) and 32 bits app-defined tag
    - Privileged UIDs may impersonate other UIDs (e.g., download manager billing traffic to app that requested the download)
- Userspace interface:
  - Apps tag their own sockets using /proc interface
  - System collects data by scraping /proc

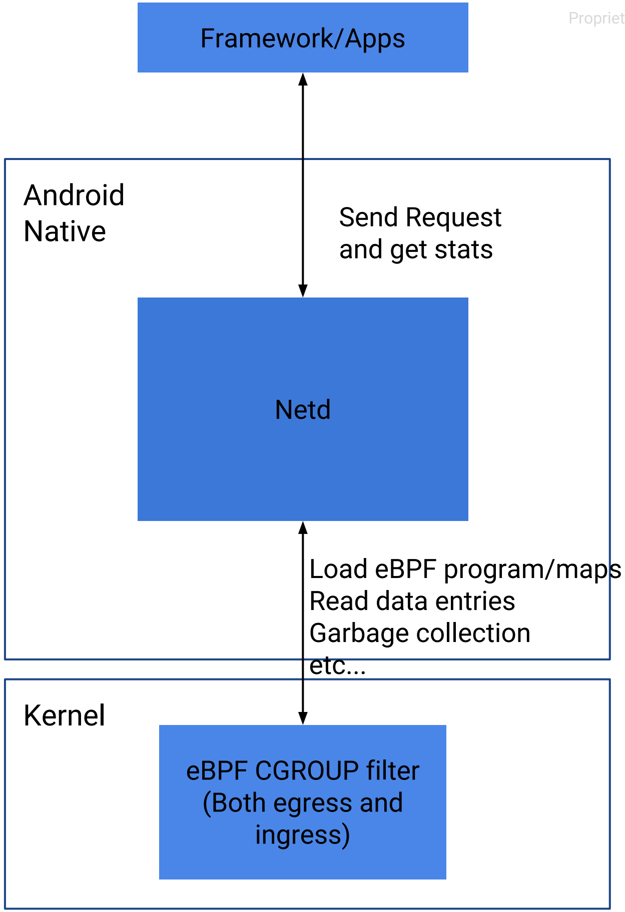
Design

Why use eBPF?

- Powerful way to apply policy from userspace
- In networking area, it can apply filters on socket, cgroup, iptables module (xt_bpf), tc-bpf, etc.
- Advantages:
  - Easier to upstream, since no custom code in kernel
  - Much less chance to cause kernel crash
  - Customizable eBPF program design
  - Multiple filter hook points in network stack.

Basic Design

- Per-cgroup eBPF program to perform accounting
  - Ingress: Transport layer (e.g. tcp_v4_rcv), same as eBPF socket filter
  - Egress: Network layer (eg. ip_finish_output)
- Stats received are stored in eBPF maps.
- Stats periodically retrieved by privileged process from eBPF map
- Apps tag sockets by sending fd using binder call to privileged process



Why cgroup filtering?

Following alternatives considered cannot fulfill our needs

xt_ebpf with pinned eBPF object

- skb->sk usually unavailable on ingress side

Per-socket eBPF filter

- Only does input packets
- Need to apply program to every fd individually
- Some sockets don't have an fd, so can't attach program to them

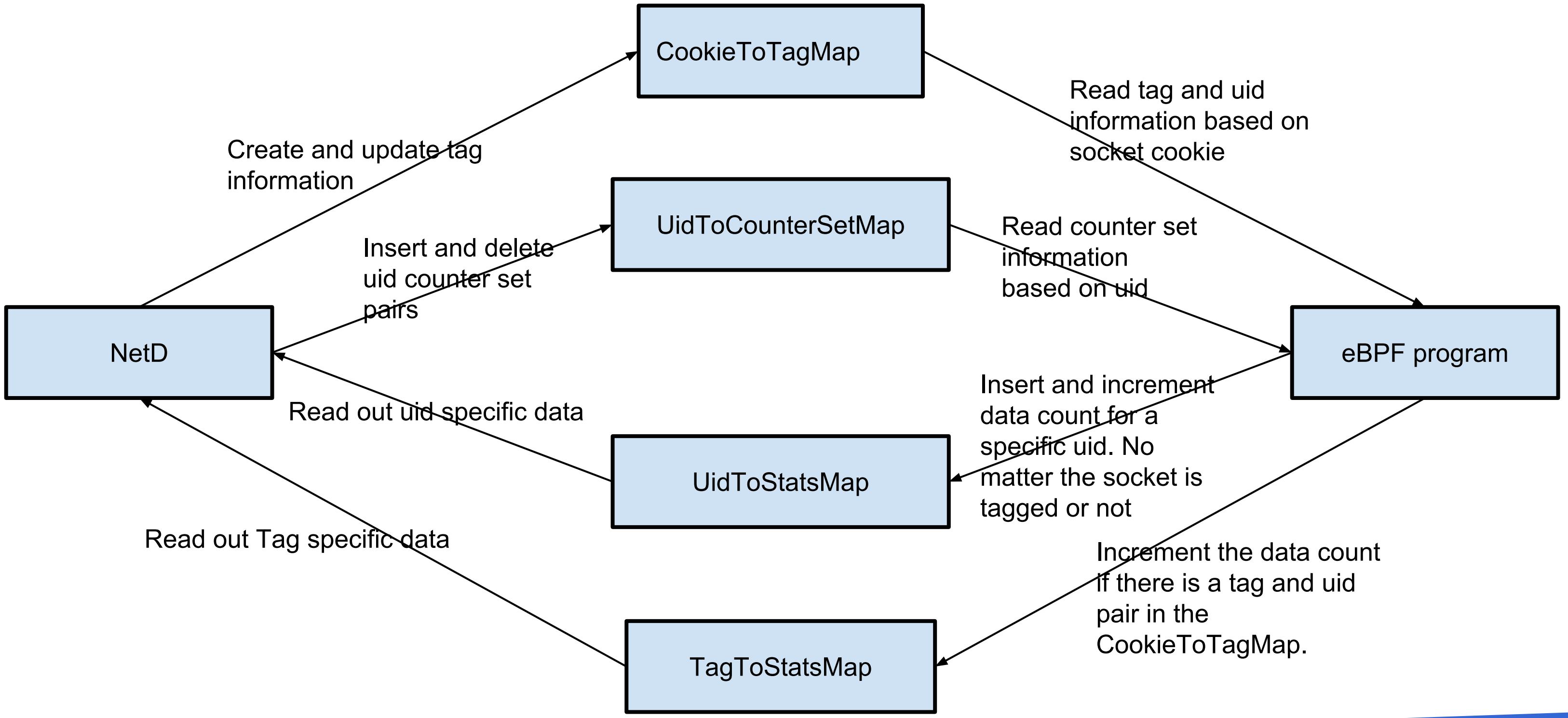
tc bpf

- Only does output packets

Data structures

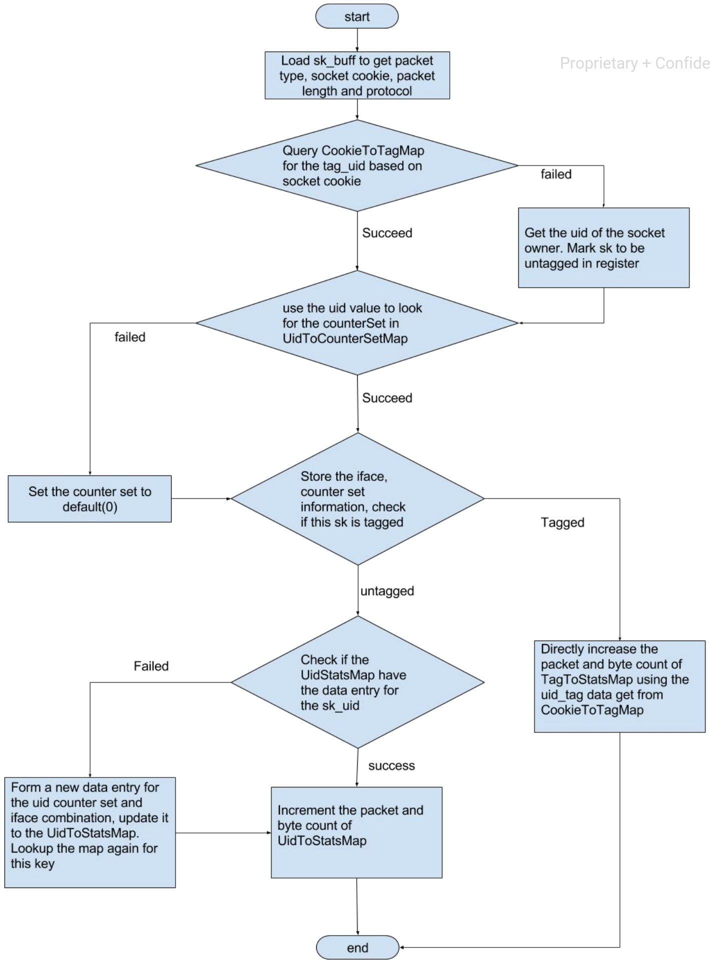
- Use sk_cookie to identify socket in various EBPF maps
  - If empty, cookie initialized by eBPF program when a packet is processed
- Cookies mapped to:
  - Socket IDs (uid | tag) if socket is tagged
- Stats entries are mapped with two struct
  - Key struct contains Socket ID | foreground state | interface
  - Value struct contains tx/rx packets number and tx/rx bytes
- Overall stats are in UidToStatsMap
- Tagged sockets stats are in TagToStatsMap

Userspace kernel interaction


CookieToTagMap
UidToCounterSetMap
NetD
eBPF program
UidToStatsMap
TagToStatsMap
Create and update tag information
Insert and delete uid counter set pairs
Read out uid specific data
Read out Tag specific data
Read tag and uid information based on socket cookie
Read counter set information based on uid
Insert and increment data count for a specific uid. No matter the socket is tagged or not
Increment the data count if there is a tag and uid pair in the CookieToTagMap.

Kernel Program

- Written in assembly like instruction arrays
  - Potentially allow creating eBPF program at run time.
- Loaded into the kernel on netd startup
- Packet information collected:
  - Socket uid
  - Packet type (tcp, udp, other)
  - Packet length
  - rx/tx interface



start

Load sk_buff to get packet type, socket cookie, packet length and protocol

Query CookieToTagMap for the tag_uid based on socket cookie

failed

Get the uid of the socket owner. Mark sk to be untagged in register

Succeed

use the uid value to look for the counterSet in UidToCounterSetMap

failed

Set the counter set to default(0)

Succeed

Store the iface, counter set information, check if this sk is tagged

Tagged

Directly increase the packet and byte count of TagToStatsMap using the uid_tag data get from CookieToTagMap

untagged

Check if the UidStatsMap have the data entry for the sk_uid

Failed

Form a new data entry for the uid counter set and iface combination, update it to the UidToStatsMap. Lookup the map again for this key

success

Increment the packet and byte count of UidToStatsMap

end

Userspace implementation

Userspace Service

- Netd is mainly responsible for managing the userspace service
- Init process:
  - Mount cgroup v2
  - Mount eBPF filesystem
- Netd Initialization
  - Create maps, load program into root cgroup
- Netd binder service:
  - Socket tag/untag
  - Periodically retrieve traffic statistics
  - Garbage collection

Userspace Initialization

- Init process:
  - Mount cgroup v2
  - Mount eBPF filesystem
- Netd startup:
  - map create and pinned to a specific location
  - Load the kernel filter program
  - Attach filter to the cgroup mounted
    - By default, the program is attached to the root cgroup so all processes will be contained

Userspace runtime services

- Socket tag/untag
- Request for stats
  - Request is passed by binder calls.
  - Netd read through the stats map and form the result
    - Combine result from uidToStatsMap and tagToStatsMap
    - App should not see the network stats of other apps
- Garbage collection
  - Scan for closed socket and clean the cookieToTag tables
  - After system server captured the stats snapshot, clean up the untagged socket stats in TagtoStatsMap
  - UidToStatsMap will never be cleaned until reboot or app uninstalled.

Netd crash recovery

- Pinned eBPF object will not be destroyed until the pinned file is deleted.
- When netd restart:
    - Scan for the pinned map file
    - Use sock_diag scan for any open sockets
    - Clean up the cookieToTagMap
    - TagToStatsMap will be garbage collected as usual when system server polls stats

Security Model

- Adding LSM hooks and selinux checks for eBPF operations in progress
- Selinux is responsible for restricting the access to eBPF object and cgroup.
    - Only allow netd to create eBPF maps, update element and load eBPF program
    - Only allow netd to access file under bpf filesystem
    - Only allow netd to access the root directory of cgroup v2
- May allow system server directly read maps to enhance performance

App compatibility

- Public APIs: TrafficStats and NetworkStatsManager,
    - System will use eBPF or xt_qtaguid depending on kernel version
- Some apps might be opening /proc/net/xt_qtaguid/... directly
    - No easy way to support this without xt_qtaguid module
        - Can’t just bound mount a file over /proc/net/xt_qtaguid/… as stats are per-UID
    - Disallow direct access as early as possible (e.g., in preview release)
- Some apps might be calling qtaguid_tagSocket, etc. directly
    - Might be able to turn these into these calls to netd
- Future implementation will switch to binder calls for all socket tagging and data retrieving processes.

Challenges

eBPF Challenges

- Memory management
  - xt_qtaguid can call kmalloc
  - eBPF maps cannot be resized, consume unswappable kernel memory
    - Tagging socket can fail, but not being able to account traffic to UID unacceptable
- Security model not fine-grained
  - Everyone can write to maps and load programs (bad)
  - Only CAP_NET_ADMIN can write to maps, so processes can't tag own sockets

Implementation Challenges

- Cgroup eBPF program call sites scattered around kernel
  - Needed several fixes to ensure different types of packets were counted [only] once
  - Still can’t count IPv6 SYN+ACKs
  - Not sure how to count IPsec packets yet
    - When applying per-socket policy, add estimated overhead to tag entry?
    - Need to avoid double-counting, deal with IPsec encapsulation, etc.
- Split user/kernel space solution
  - Many moving parts: kernel program, netd, init, …

Necessary kernel changes

- Fixes for accounting correct packets
- New getsockopt SO_COOKIE
- Helper functions to get UID and cookie
- All upstream as of 4.12, backported to android-4.9
- In progress: LSM hooks and selinux checks for eBPF operations

THANK YOU

Q & A