Merging packets with System Events using eBPF

Luca Deri <deri@ntop.org>, @lucaderi
Samuele Sabella <sabella@ntop.org>, @sabellasamuele
About Us

• Luca: lecturer at the University of Pisa, CS Department, founder of the ntop project.
• Samuele: student at Unipi CS Department, junior engineer working at ntop.
• ntop develops open source network traffic monitoring applications. ntop (circa 1998) is the first app we released and it is a web-based network monitoring application.
• Today our products range from traffic monitoring, high-speed packet processing, deep-packet inspection (DPI), IDS/IPS acceleration, and DDoS Mitigation.
• See http://github.com/ntop/
What is Network Traffic Monitoring?

• The key objective behind network traffic monitoring is to ensure availability and smooth operations on a computer network. Network monitoring incorporates network sniffing and packet capturing techniques in monitoring a network. Network traffic monitoring generally requires reviewing each incoming and outgoing packet.

https://www.techopedia.com/definition/29977/network-traffic-monitoring
ntop Ecosystem (2009): Packets Everywhere
ntop Ecosystem (2019): Still Packets [1/2]
What’s Wrong with Packets?

• Nothing in general but…
  ◦ It is a paradigm good for monitoring network traffic from outside of systems on a passive way.
  ◦ Encryption is challenging DPI techniques (BTW ntop maintains an open source DPI toolkit called nDPI).
  ◦ Virtualisation techniques reduce visibility when monitoring network traffic as network manager are blind with respect to what happens inside systems.
  ◦ Developers need to handle fragmentation, flow reconstruction, packet loss/retransmissions… metrics that would be already available inside a system.
From Problem Statement to a Solution

• Enhance network visibility with system introspection.
• Handle virtualisation as first citizen and don't be blind (yes we want to see containers interaction).
• Complete our monitoring journey and…
  ◦ System Events: processes, users, containers.
  ◦ Flows
  ◦ Packets
• …bind system events to network traffic for enabling continuous drill down: system events uncorrelated with network traffic are basically useless.
Early Experiments: Sysdig [1/3]

• ntop has been an early sysdig adopter adding in 2014 sysdig events support in PF_RING, ntopng, nProbe.
Early Experiments: Sysdig [2/3]
Early Experiments: Sysdig [3/3]

• Despite all our efforts, this activity has NOT been a success for many reasons:
  ◦ Too much CPU load (in average +10-20% CPU load) due to the design of sysdig (see later).
  ◦ People do not like to install agents on systems as this might create interferences with other installed apps.
  ◦ Sysdig requires a new kernel module that sometimes is not what sysadmins like as it might invalidate distro support.
  ◦ Containers were not so popular in 2014, and many people did not consider system visibility so important at that time.
How Sysdig Works

• As sysdig focuses on system calls for tracking a TCP connections we need to:
  ◦ Discard all non TCP related events (sockets are used for other activities on Linux such as Unix sockets)
  ◦ Track socket() and remember the socketId to process/thread
  ◦ Track connect() and accept() and remember the TCP peers/ports.
  ◦ Collect packets and bind each of them to a flow (i.e. this is packet capture again, using sysdig instead of libpcap).

• This explains the CPU load, complexity…
Welcome to eBPF

eBPF is great news for ntop as

- It gives the ability to avoid sending everything to user-space but perform in kernel computations and send metrics to user-space.
- We can track more than system calls (i.e. be notified when there is a transmission on a TCP connection without analyzing packets).
- It is part of modern Linux systems (i.e. no kernel module needed).
libebpfflow Overview [1/2]

// ------ ------ STRUCTS AND CLASSES ------ ------ //
struct ipv4_kernel_data {
    __u64 saddr;
    __u64 daddr;
    struct netInfo net;
};

struct ipv6_kernel_data {
    unsigned __int128 saddr;
    unsigned __int128 daddr;
    struct netInfo net;
};
typedef struct {
    __u64 ktime;
    char ifname[IFNAMSIZ];
    struct timeval event_time;
    __u8 ip_version:4, sent_packet:4;
} event;
union {
    struct ipv4_kernel_data v4;
    struct ipv6_kernel_data v6;
} event;

struct taskInfo {
    __u32 pid; /* Process Id */
    __u32 tid; /* Thread Id */
    __u32 uid; /* User Id */
    __u32 gid; /* Group Id */
    char task[COMMAND_LEN], *full_task_path;
};
libebpfflow Overview [2/2]

// Attaching probes ------ //
if (userarg_eoutput && userarg_tcp) {
    // IPv4
    AttachWrapper(&ebpf_kernel, "tcp_v4_connect", "trace_connect_entry", BPF_PROBE_ENTRY);
    AttachWrapper(&ebpf_kernel, "tcp_v4_connect", "trace_connect_v4_return", BPF_PROBE_RETURN);
    // IPv6
    AttachWrapper(&ebpf_kernel, "tcp_v6_connect", "trace_connect_entry", BPF_PROBE_ENTRY);
    AttachWrapper(&ebpf_kernel, "tcp_v6_connect", "trace_connect_v6_return", BPF_PROBE_RETURN);
}

if (userarg_einput && userarg_tcp)
    AttachWrapper(&ebpf_kernel, "inet_csk_accept", "trace_accept_return", BPF_PROBE_RETURN);

if (userarg_retr)
    AttachWrapper(&ebpf_kernel, "tcp_retransmit_skb", "trace_tcp_retransmit_skb", BPF_PROBE_ENTRY);

if (userarg_tcpclose)
    AttachWrapper(&ebpf_kernel, "tcp_set_state", "trace_tcp_close", BPF_PROBE_ENTRY);

if (userarg_einput && userarg_udp)
    AttachWrapper(&ebpf_kernel, "inet_recvmsg", "trace_inet_recvmsg_entry", BPF_PROBE_ENTRY);
    AttachWrapper(&ebpf_kernel, "inet_recvmsg", "trace_inet_recvmsg_return", BPF_PROBE_RETURN);

if (userarg_eoutput && userarg_udp) {
    AttachWrapper(&ebpf_kernel, "udp_sendmsg", "trace_udp_sendmsg_entry", BPF_PROBE_ENTRY);
    AttachWrapper(&ebpf_kernel, "udp_sendmsg", "trace_udpv6_sendmsg_entry", BPF_PROBE_ENTRY);
    }
Gathering Information Through eBPF

• In Linux every task has associated a struct (i.e. `task_struct`) that can be retrieved by invoking the function `bpf_get_current_task` provided by eBPF. By navigating through the kernel structures it can be gathered:
  ◦ uid, gid, pid, tid, process name and executable path
  ◦ cgroups associated with the task.
  ◦ connection details: source and destination ip/port, bytes send and received, protocol used.
Containers Visibility: cgroups and Docker

• For each container Docker creates a cgroup whose name corresponds to the container identifier.
• Therefore by looking at the task cgroup the docker identifier can be retrieved and further information collected.
TCP Under the Hood: accept

A **kprobe** has been attached to **inet_csk_accept**
- Used to accept the next outstanding connection.
- Returns the socket that will be used for the communication, NULL if an error occurs.
- Information is collected both from the socket returned and from the **task_struct** associated with the process that triggered the event.

In a similar fashion events concerning retransmissions and socket closure can be monitored.
TCP Under the Hood: connect

An hash table, indexed with thread IDs, has been used:

◦ When `connect` is invoked the socket is collected from the function arguments and stored together with the kernel time.
◦ When the function terminates the execution, the return value is collected and the thread ID is used to retrieve the socket from the hash table.
◦ The kernel time is used to calculate the connection latency.
Using libebpfflow from CLI

deri@ubuntu18 205$ sudo ./ebpflow
kProbes attached
Output buffer opened
[ktime: 0][pid: 11443][uid: 0][gid: 1000][sudo]
  __ parent: [pid: 11318][uid: 1000][gid: 1000][tcsh]
  __ netinfo: [UDP/snd][IPv4][addr: 127.0.0.1:56452 <-> 127.0.0.1:53]
  __ [minor_faults: 213][major_faults: 0]
[ktime: 1][pid: 10215][uid: 997][gid: 997][pihole-FTL]
  __ parent: [pid: 1][uid: 0][gid: 0][systemd]
  __ netinfo: [UDP/rcv][IPv4][addr: 127.0.0.1:56452 <-> 127.0.0.1:53]
  __ [minor_faults: 5849][major_faults: 0]
[ktime: 6][pid: 11443][uid: 0][gid: 1000][sudo]
  __ parent: [pid: 11318][uid: 1000][gid: 1000][tcsh]
  __ netinfo: [UDP/snd][IPv4][addr: 127.0.0.1:43457 <-> 127.0.0.1:53]
  __ [minor_faults: 216][major_faults: 0]
[ktime: 7][pid: 10215][uid: 997][gid: 997][pihole-FTL]
  __ parent: [pid: 1][uid: 0][gid: 0][systemd]
  __ netinfo: [UDP/rcv][IPv4][addr: 127.0.0.1:43457 <-> 127.0.0.1:53]
  __ [minor_faults: 5849][major_faults: 0]
[ktime: 31308][pid: 1136][uid: 114][gid: 117][chronyd]
  __ parent: [pid: 1][uid: 0][gid: 0][systemd]
  __ netinfo: [UDP/snd][IPv4][addr: 127.0.0.1:34324 <-> 127.0.0.1:123]
  __ [minor_faults: 147][major_faults: 2]
[ktime: 31437][pid: 1136][uid: 114][gid: 117][chronyd]
  __ parent: [pid: 1][uid: 0][gid: 0][systemd]
  __ netinfo: [UDP/rcv][IPv4][addr: 213.251.52.250:123 <-> 192.168.1.87:34324]
  __ [minor_faults: 147][major_faults: 2]
[ktime: 52712][pid: 1136][uid: 114][gid: 117][chronyd]
  __ parent: [pid: 1][uid: 0][gid: 0][systemd]
  __ netinfo: [UDP/snd][IPv4][addr: 127.0.0.1:34751 <-> 127.0.0.1:123]
  __ [minor_faults: 147][major_faults: 2]
Integrating eBPF with ntopng

• We have done an early integration of eBPF with ntopng using the libebpflow library we developed:
  ◦ Incoming TCP/UDP events are mapped to packets monitored by ntopng.
  ◦ We’ve added user/process/flow integration and partially implemented process and user statistics.

• Work in progress
  ◦ Container visibility (including pod), retransmissions… are reported by eBPF but not yet handled inside ntopng.
  ◦ To do things properly we need to implement a system interface in ntopng where to send all system events.
  ◦ Decide how/if netlink will be part of the equation.
ntopng with eBPF: Flows

Active Flows

<table>
<thead>
<tr>
<th>Application</th>
<th>L4 Proto</th>
<th>Client</th>
<th>Server</th>
<th>Duration</th>
<th>Breakdown</th>
<th>Actual Thpt</th>
<th>Total Bytes</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>Info</td>
<td>ICMP</td>
<td>217.28.76.4</td>
<td>pc-deri.nic.it</td>
<td>19:04:30</td>
<td>Client</td>
<td>0 bit/s</td>
<td>1.32 MB</td>
<td>Echo Reply</td>
</tr>
<tr>
<td>Info</td>
<td>IMAPS</td>
<td>TCP</td>
<td>pc-deri.nic.it</td>
<td>12:16:18</td>
<td>Client</td>
<td>0 bit/s</td>
<td>370.53 KB</td>
<td></td>
</tr>
<tr>
<td>Info</td>
<td>IMAPS</td>
<td>TCP</td>
<td>pc-deri.nic.it</td>
<td>04:47:03</td>
<td>Client</td>
<td>0 bit/s</td>
<td>407.59 KB</td>
<td></td>
</tr>
<tr>
<td>Info</td>
<td>SSL.Dropbox</td>
<td>TCP</td>
<td>pc-deri.nic.it</td>
<td>01:27:35</td>
<td>Client</td>
<td>0 bit/s</td>
<td>786.7 KB</td>
<td>bolt.dropbox.com</td>
</tr>
<tr>
<td>Info</td>
<td>SSL.Dropbox</td>
<td>TCP</td>
<td>pc-deri.nic.it</td>
<td>47:38</td>
<td>Client</td>
<td>0 bit/s</td>
<td>93.08 KB</td>
<td>bolt.dropbox.com</td>
</tr>
<tr>
<td>Info</td>
<td>MDNS</td>
<td>UDP</td>
<td>mesure.nic.it</td>
<td>224.0.0.281</td>
<td>Client</td>
<td>0 bit/s</td>
<td>7.24 KB</td>
<td></td>
</tr>
<tr>
<td>Info</td>
<td>MDNS</td>
<td>UDP</td>
<td>mesure.nic.it</td>
<td>01:37</td>
<td>Client</td>
<td>0 bit/s</td>
<td>1.21 KB</td>
<td></td>
</tr>
<tr>
<td>Info</td>
<td>SSLTelegram</td>
<td>TCP</td>
<td>pc-deri.nic.it</td>
<td>01:42</td>
<td>Client</td>
<td>0 bit/s</td>
<td>3.27 KB</td>
<td></td>
</tr>
<tr>
<td>Info</td>
<td>SSLntop</td>
<td>TCP</td>
<td>l7.ntop.org</td>
<td>00:06</td>
<td>Client</td>
<td>0 bit/s</td>
<td>6.3 KB</td>
<td>l7.ntop.org</td>
</tr>
<tr>
<td>Info</td>
<td>SSLntop</td>
<td>TCP</td>
<td>l7.ntop.org</td>
<td>00:06</td>
<td>Client</td>
<td>0 bit/s</td>
<td>6.29 KB</td>
<td>l7.ntop.org</td>
</tr>
</tbody>
</table>
ntopng with eBPF: Users + Processes
ntopng with eBPF: Processes + Protocols
Current eBPF Work Items: UDP

• Contrary to TCP, in UDP we need to handle packets. To avoid overloading the system we are using an in-kernel LRU to minimise load: is there a better option available that avoids us playing with packets at all?

• As in UDP each packet can have a different destination, intercepting up in the stack some metadata info are missing (local IP/Ethernet is computed after routing decision).

• Better multicast handling.
BCC/eBPF Pitfalls

- BCC (BPF Compiler Collection) has limitations in terms of:
  - Function complexity/length: memory/stack and loop unroll are limited and this might be a problem in some cases (e.g. decoding).
  - Sometimes its behaviour is non deterministic and the same code works with the dev but fails to compile with the stable version.
  - No ability to read the BCC API version (functions prototypes change cross versions).
- Inability to read message drops number.
- Packet decoding can be a nightmare due to restrictions on function calls.

Frame 1: 217 bytes on wire (1736 bits), 217 bytes captured (1736 bits) on interface 0
- Ethernet II, Src: 00:16:3e:6d:83:dc (00:16:3e:6d:83:dc), Dst: 00:16:3e:8f:4c:7e (00:16:3e:8f:4c:7e)
- Generic Routing Encapsulation (ERSSPAN)
- Encapsulated Remote Switch Packet ANalysis Type II
- Ethernet II, Src: 00:16:3e:6d:83:dc (00:16:3e:6d:83:dc), Dst: 00:16:3e:8f:4c:7e (00:16:3e:8f:4c:7e)
- 802.10 Virtual LAN, PRI: 0, DEI: 0, ID: 21
- User Datagram Protocol, Src Port: 64556, Dst Port: 3389
- Data (121 bytes)
Conclusions

• With eBPF it is now possible to have full system and network visibility in an integrated fashion.

• Contrary to Sysdig, eBPF load on the system is basically unnoticeable and no kernel module is necessary (i.e. issues of early work are now solved).

• Container/user/process information allows us to enhance network communications with metadata that is great not just for visibility but also for spotting malicious system activities.

• System visibility will be integrated in ntopng 4.x due later this year.