Using eBPF for network traffic analysis

Samuele Sabella
Luca Deri
Network Monitoring

In the past few years the need to better understand what’s happening in our networks has lead from tools for volumetric data analysis to what is now called “deep packet inspection”
Containers are dynamical entities that provide microservices and can be contracted or expanded (i.e. elastic computing) on needs (e.g. kubernetes). Monitoring the system from the outside (i.e. looking at network packets), is no longer enough.
Monitoring Containers

The “photography” we take by looking at traffic can be out of date.

The information we gather looking at the packets only, are not complete:

- How much traffic belongs to user X?
- Calculating latency by looking at packets is reconstructing what we think is happening at a lower level

It is not an absolute view, we don’t know what is happening in what isn’t our domain!
Not a new idea: sysdig

Provides a way to observe the system at the kernel system call level.

We receive system call events, that however are difficult and time consuming to interpret:

- We work at a high level
- Runtime complexity, heavy load on CPU

It is a very good tool but with some limitations.
eBPF: what is and how can be used?

In 1997, it was introduced in Linux kernel as a technology for in-kernel packet filtering. The authors are Steven McCanne and Van Jacobson from Lawrence Berkeley Laboratory.

eBPF extended what bpf virtual machine could do, allowing it to run other kind of events and take several action other than filtering
**eBPF:** lot of very useful tools

- **tcplife:** Trace the lifespan of TCP sessions and summarize.
- **tcptop:** Summarize TCP send/recv throughput by host.
- **biolatency:** Summarize block device I/O latency as a histogram.
- **filetop:** file reads and writes by process.
A toolkit for ebpf: **bcc**

An easy to use toolkit to write eBPF programs that offers a front-end interface in different languages:

- Python
- Lua
- C++
- Rust

The repository offers a lot of examples on various topics
BPF: how it works

1. User writes C code.
2. C code is compiled to LLVM/clang bytecode.
3. Bytecode is compiled to native code by JIT + Verifier.
4. Native code is linked with bpf code.
5. Event triggers the code execution in the kernel.
eBPF/bcc: basic usage

Events
- Kprobes
- Kretprobes
- Uprobes
- Uretprobs

Output
- printk
- perf_output

Maps
- Hash tables
- Histograms
- Lru hash

Helper functions
- bpf_get_current_task
- bpf_ktime_get_ns
- bpf_get_current_comm
eBPF: limitations

- eBPF and bcc are not mature projects
- Difficult to use
- Kernel functions available to use are the one determined by the flag `prog_type`
- We can’t do cycles
- VM is read only in kprobes!
- To access data external to ebpf stack we must use `bpf_probe_read` (not always necessary, the compiler may provide us support)
ebpflow: our objectives

Create an in-kernel flow monitoring tool for traffic analysis which can observe the system and take actions within the kernel.

▸ Reliable and trustworthy information on the status of the system when events take place.
▸ Low overhead event-based monitoring
▸ Information on users, network statistics, containers and processes
ebpflow: basic tcp

```c
int tcp_v4_connect(struct sock* sk)
{
    sk_hash.update(tid, sk);
    connect execution
    ret = PT_REGS_RC(ctx);
    tid = get_curr_tid();
    if (!ret) {
        sk_hash.delete(tid);
    }
    sk_hash.lookup(tid);
    // get all info and push event on buffer
    continue execution
}
```
**ebpflow: how to spot containers?**

Container can be spotted by looking at `proc/cgroup`, however retrieving information from there is a too slow operation.

Containers are processes isolated with the use of namespaces and cgroups:

- We can navigate through kernel data structures and read the namespaces
- We extract information at kernel level
ebpflow: ntop integration

- Low overhead
- Faithful picture of the state of the system
- Per user flow informations
- We can build the process hierarchy for each flow
### Active Flows

<table>
<thead>
<tr>
<th>Application</th>
<th>L4 Proto</th>
<th>Client</th>
<th>Server</th>
<th>Duration</th>
<th>Breakdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Redis</td>
<td>TCP</td>
<td>localhost:42646</td>
<td>localhost:6379</td>
<td>00:22</td>
<td>Client</td>
</tr>
<tr>
<td>DNS</td>
<td>UDP</td>
<td>localhost:51259</td>
<td>localhost:domain</td>
<td>&lt;1 sec</td>
<td>Client</td>
</tr>
<tr>
<td>DNS.ntop</td>
<td>UDP</td>
<td>localhost:59163</td>
<td>localhost:domain</td>
<td>&lt;1 sec</td>
<td>Client</td>
</tr>
<tr>
<td>HTTP</td>
<td>TCP</td>
<td>localhost:39174</td>
<td>localhost:8086</td>
<td>&lt;1 sec</td>
<td>Client</td>
</tr>
<tr>
<td>HTTP</td>
<td>TCP</td>
<td>localhost:39172</td>
<td>localhost:8086</td>
<td>&lt;1 sec</td>
<td>Client</td>
</tr>
<tr>
<td>HTTP</td>
<td>TCP</td>
<td>localhost:39170</td>
<td>localhost:8086</td>
<td>&lt;1 sec</td>
<td>Client</td>
</tr>
<tr>
<td>DNS</td>
<td>UDP</td>
<td>localhost:40920</td>
<td>localhost:domain</td>
<td>&lt;1 sec</td>
<td>Client</td>
</tr>
<tr>
<td>Redis</td>
<td>TCP</td>
<td>localhost:42646</td>
<td>localhost:6379</td>
<td>&lt;1 sec</td>
<td>Client</td>
</tr>
</tbody>
</table>

Showing 1 to 8 of 8 rows. Idle flows not listed.
### TCP Flags

Client ➔ Server: FIN SYN PUSH ACK

This flow is completed and will expire soon.

### Flow Status

Normal

### Client Process Information

<table>
<thead>
<tr>
<th>User Name</th>
<th>root</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process PID/Name</td>
<td>21873/ntopng</td>
</tr>
</tbody>
</table>

### Server Process Information

<table>
<thead>
<tr>
<th>User Name</th>
<th>influxdb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process PID/Name</td>
<td>2741/influxd</td>
</tr>
</tbody>
</table>

### HTTP

<table>
<thead>
<tr>
<th>HTTP Method</th>
<th>GET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server Name</td>
<td>localhost</td>
</tr>
<tr>
<td>URL</td>
<td>localhost/query?db=ntopng&amp;q=5</td>
</tr>
<tr>
<td>Response Code</td>
<td>200</td>
</tr>
</tbody>
</table>
Demo
Future work

- In Linux Kernel version 4.16 a new functionality has been added: `bpf_override_return`
  - Provides a way to override the return value of functions
  - The kernel function has to be whitelisted to allow error injection with: `ALLOW_ERROR_INJECTION`
  - It is supported only by few function (e.g. `open_ctree`) but in the future we hope also the networking functions will be supported
eBPF and bcc are great and powerful tools. However, due to the fact that they are not mature project, they are not stable and lack of some basic features. Some workaround are often needed.

They can offer a different point of view from the one provided by looking only at traffic that goes through the system.
References

BCC github repository:
https://github.com/iovisor/bcc

Brendan Gregg blog
http://www.brendangregg.com/

Reading material
https://qmonnet.github.io/whirl-offload/2016/09/01/dive-into-bpf/
References

eBPF intro:

Cool blog by Julia Evans:
https://jvns.ca/blog/2017/07/05/linux-tracing-systems