Designing a *gRPC* Interface for *Kernel Tracing* with *eBPF*

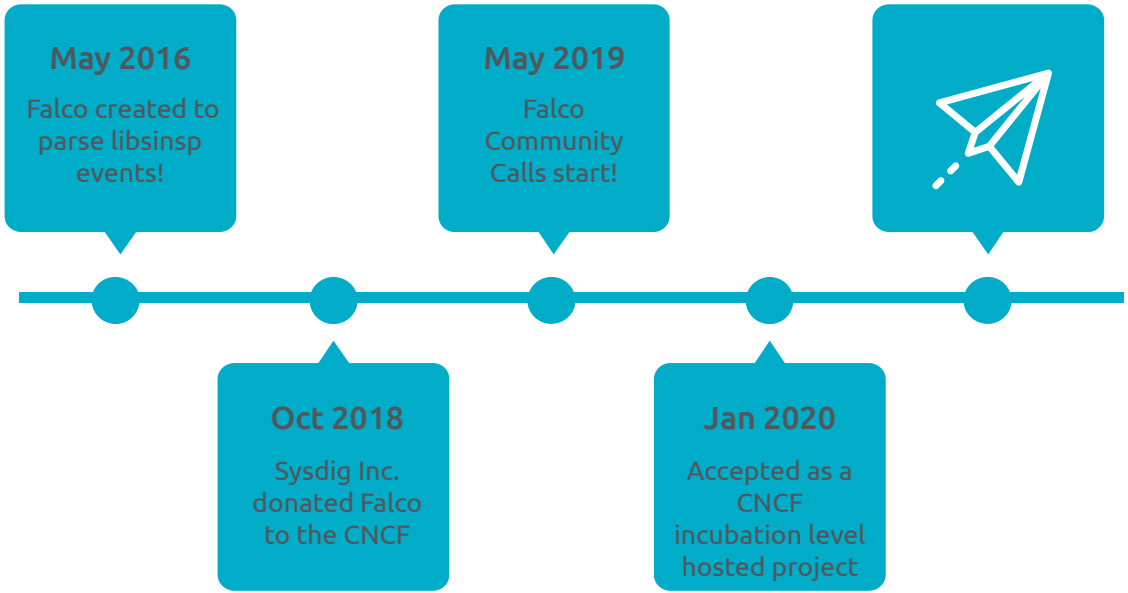
@leodido

KubeCon

CloudNativeCon

Europe 2020

A timeline always works fine



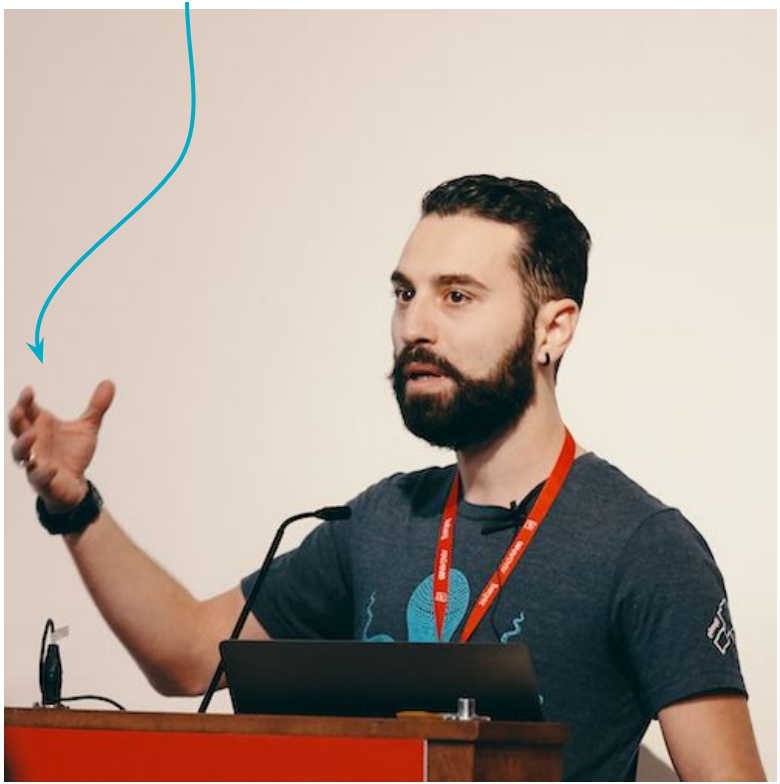
Whoami!

Leonardo Di Donato

Open Source Software Engineer
Falco Maintainer

@leodido

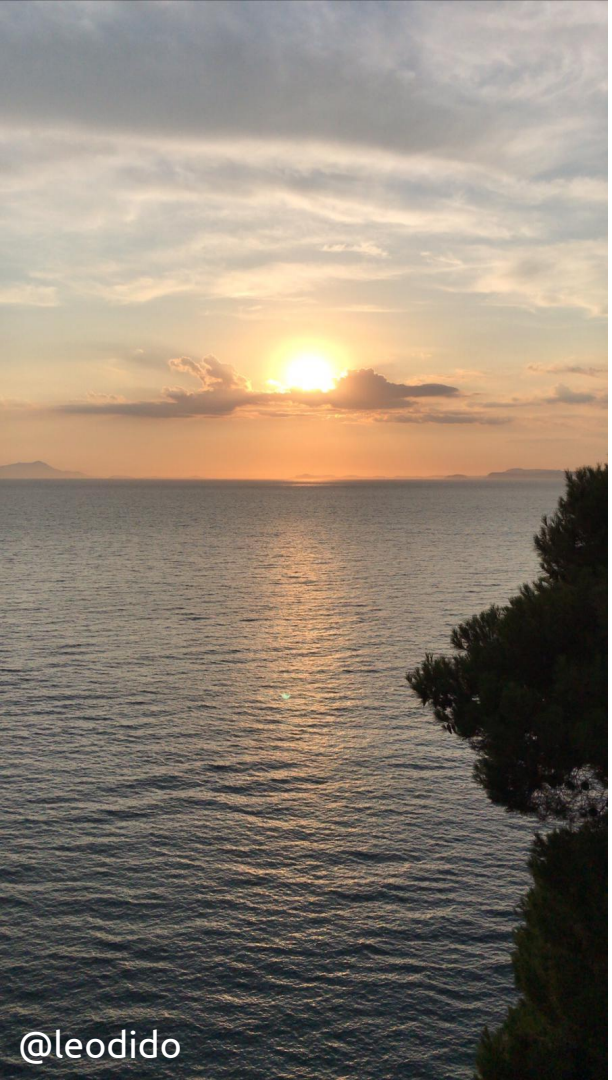
extra points to who spots the meaning of this Italian hand-gesture! 😄



sysdig

Falco

CLOUD NATIVE COMPUTING FOUNDATION



Contents

Security

Prevention

Use **policies** to *change the behavior* of a process by preventing syscalls from succeeding (also killing the process sometimes).

Detection

Use **policies** to *monitor the behavior* of a process and notify when its behavior steps outside the policy.

Security

Enforcement

sandboxing, access control

- ❏ seccomp
- ❏ seccomp-bpf
- ❏ SELinux
- ❏ AppArmor
- ❏ Cloud-Native Security
  - ❏ PSP
  - ❏ policy-based admission plugins
  - ❏ network policies
  - ❏ ...

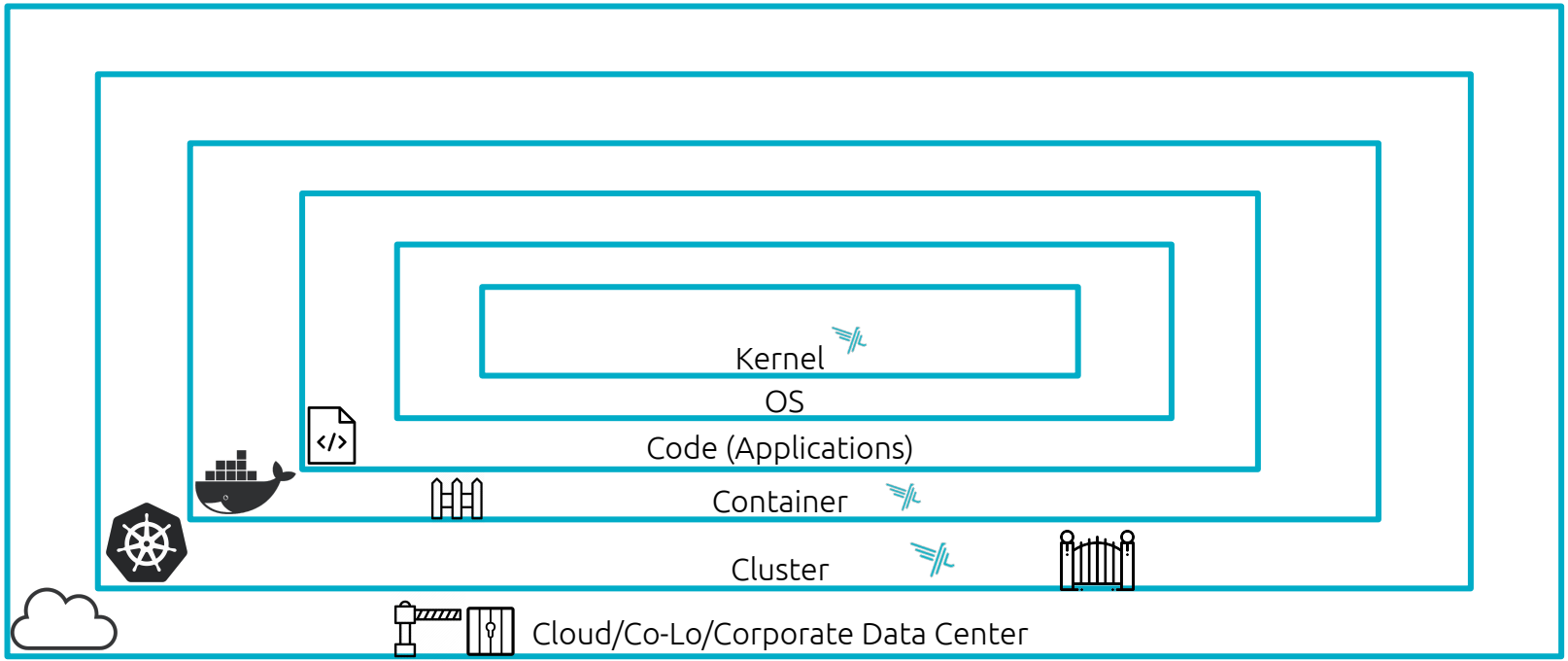
Auditing

behavioral monitoring, intrusion & anomaly detection, forensics

- ❏ auditd
- ❏ Falco
- ❏ ...
- ❏ *a lot still to be done in this space!*

Prevention is not enough.

Combine with runtime detection tools. Use a defense-in-depth strategy.



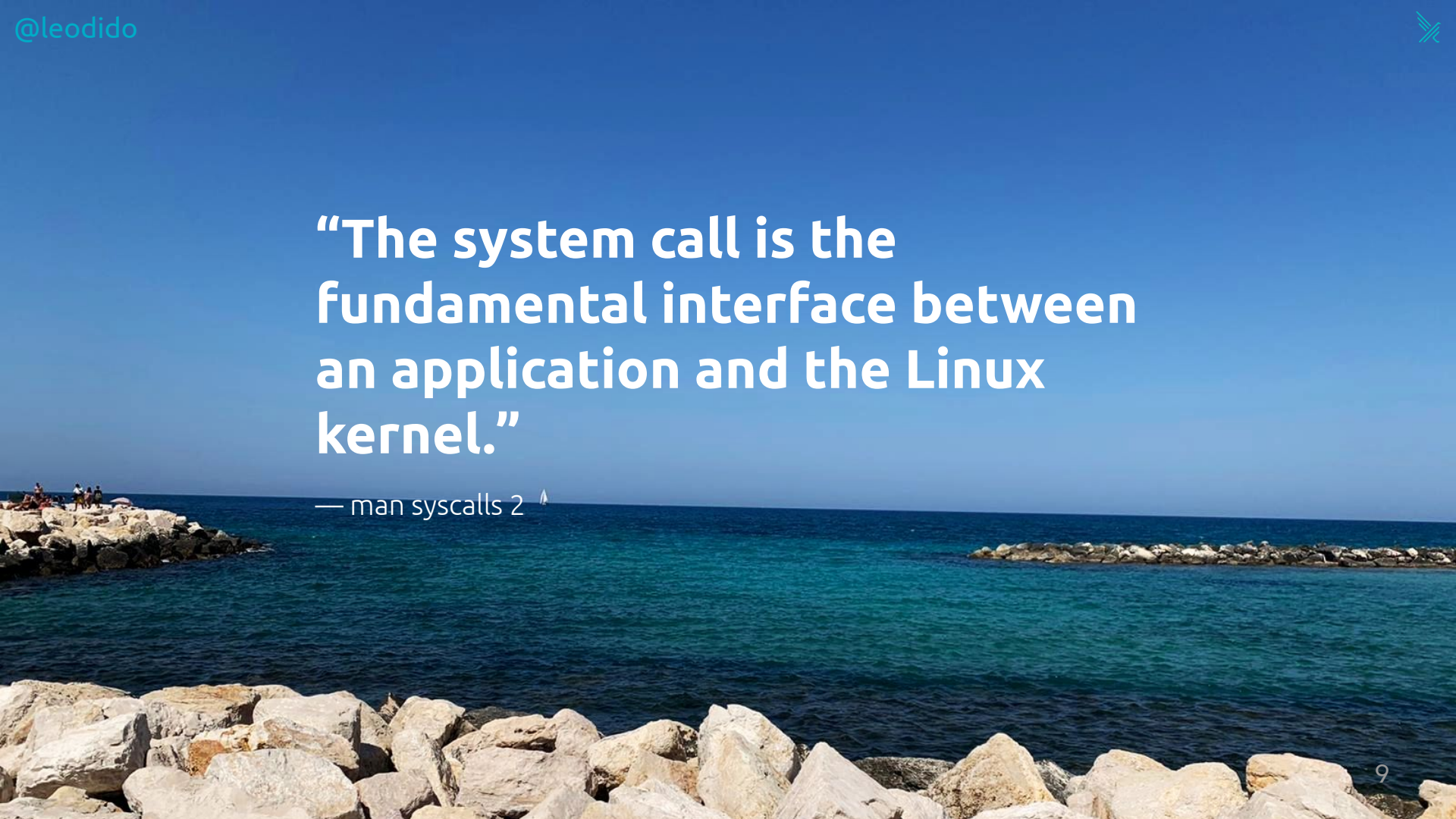
Runtime Security

She's Kelly. 💕

I have a lock on my front door and an alarm, but she alerts me when things aren't going right, when little bro is misbehaving or if there's someone suspicious outside or nearby.

She detects runtime anomalies in my life at home.

Thanks @ckranz for the inspiration!



“The system call is the fundamental interface between an application and the Linux kernel.”

— man syscalls 2

Syscalls only are not enough, too. 🤦

Containers

- ❏ Did the event originated in a container?
- ❏ What is the container name and ID?
- ❏ What is the container image?

Context

- ❏ timing
- ❏ arguments

Orchestrator

- ❏ In which cluster it is running?
- ❏ On which node?
- ❏ What is the container runtime interface in use?

Kernel module

Pros: very efficient, implement almost anything
Cons: kernel panics, not always suitable

eBPF probe

Pros: program the kernel without risking to break it
Cons: newer kernels

pdig

Pros: (almost) unprivileged
Cons: really hackish, ~20% slower

Other methods?

Future inputs/drivers?

eBPF

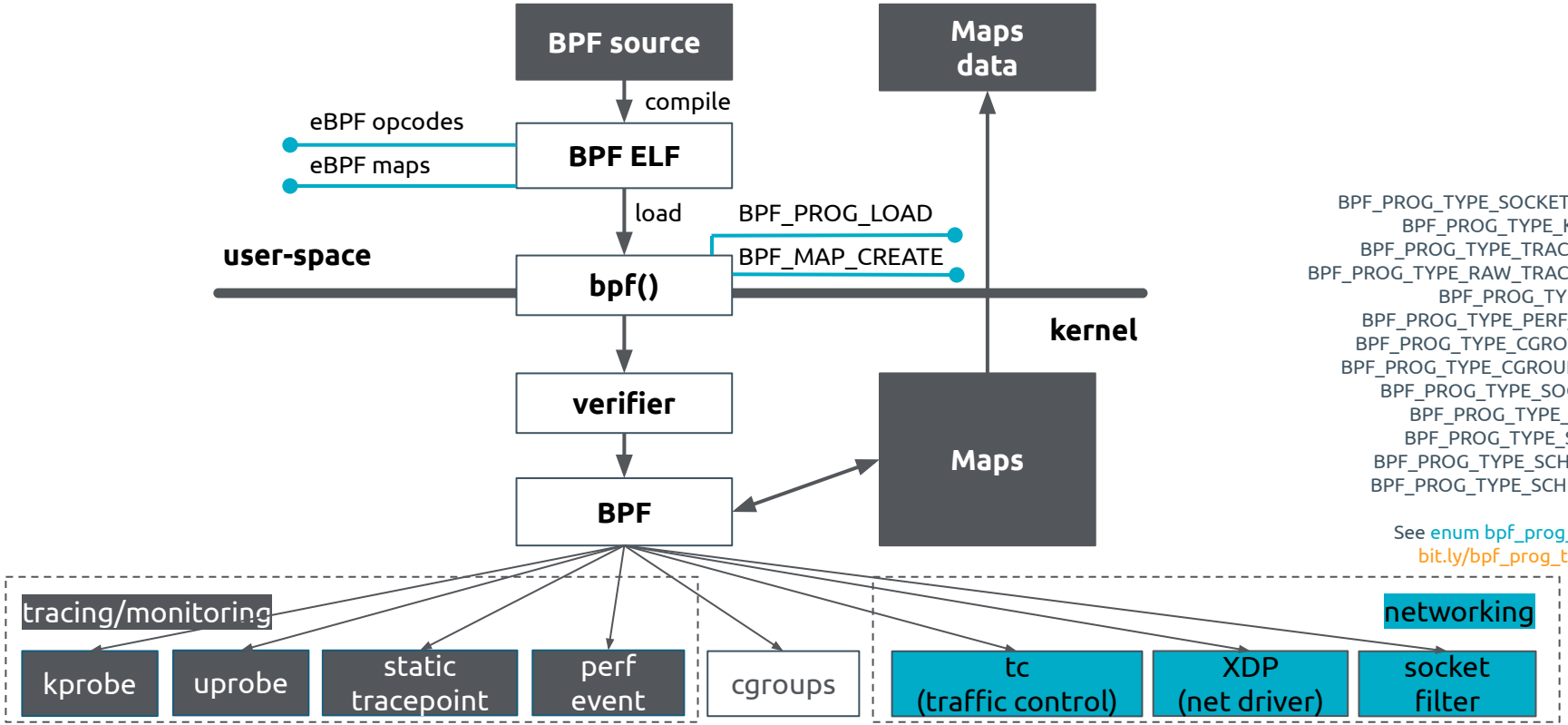
Not just packet filtering anymore.

You can now write mini programs that run on events (kernel routine execution, disk I/O, syscall) which are run in a **safe register-based VM using a custom 64 bit RISC instruction set** in the kernel.

The **In-kernel verifier** refuses to load eBPF programs with:

- ❏ invalid or bad pointer dereferences
- ❏ exceeding maximum call stack
- ❏ loops without an upper bound
- ❏ ...

Stable **A**pplication **B**inary **I**nterface (**ABI**).



BPF_PROG_TYPE_SOCKET_FILTER
BPF_PROG_TYPE_KPROBE
BPF_PROG_TYPE_TRACEPOINT
BPF_PROG_TYPE_RAW_TRACEPOINT
BPF_PROG_TYPE_XDP
BPF_PROG_TYPE_PERF_EVENT
BPF_PROG_TYPE_CGROUP_SKB
BPF_PROG_TYPE_CGROUP_SOCK
BPF_PROG_TYPE_SOCK_OPS
BPF_PROG_TYPE_SK_SKB
BPF_PROG_TYPE_SK_MSG
BPF_PROG_TYPE_SCHED_CLS
BPF_PROG_TYPE_SCHED_ACT

See enum bpf_prog_type at bit.ly/bpf_prog_types

How does eBPF work?

eBPF maps: sharing state between kernel and userspace

async in-kernel key-value store

Each map type has:

- a type
- a max number of elements
- key size (bytes)
- value size (bytes)

map operations

- BPF_MAP_CREATE
- BPF_MAP_LOOKUP_ELEM
- BPF_MAP_UPDATE_ELEM
- BPF_MAP_DELETE_ELEM
- BPF_MAP_GET_NEXT_KEY
- ...

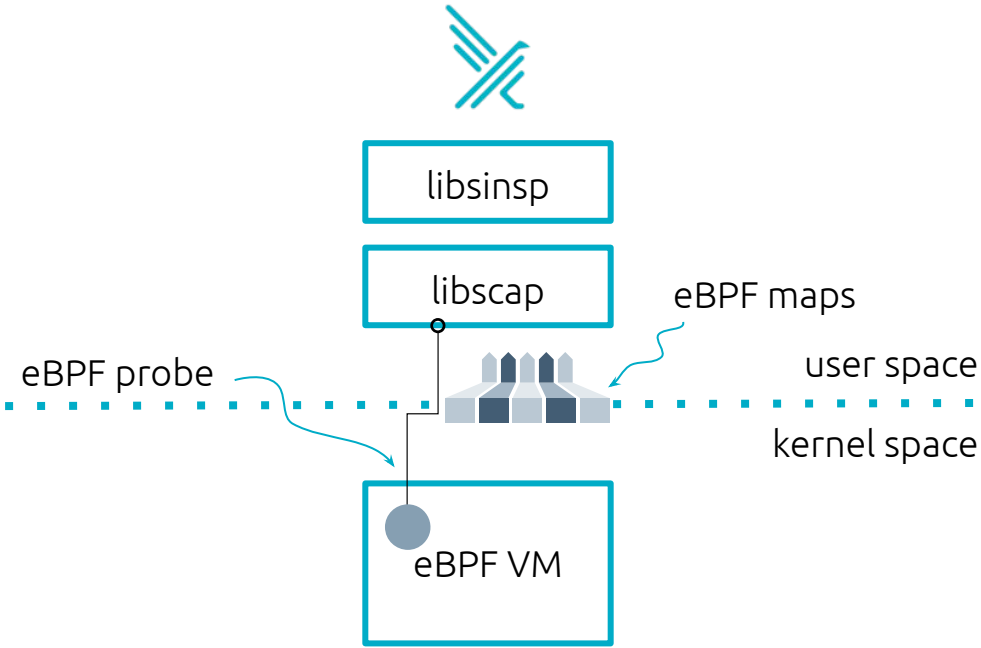
See enum bpf_cmd at bit.ly/bpf_map_commands

so many map types

- BPF_MAP_TYPE_HASH
- BPF_MAP_TYPE_ARRAY
- BPF_MAP_TYPE_PROG_ARRAY
- BPF_MAP_TYPE_PERF_EVENT_ARRAY
- BPF_MAP_TYPE_LPM_TRIE
- BPF_MAP_TYPE_PERCPU_HASH
  BPF_MAP_TYPE_PERCPU_ARRAY
- ...

See enum bpf_map_type at bit.ly/bpf_map_types

Syscalls from Falco eBPF probe



Build

Prerequisites:
clang, debugfs on /sys/kernel/debug, kernel headers...

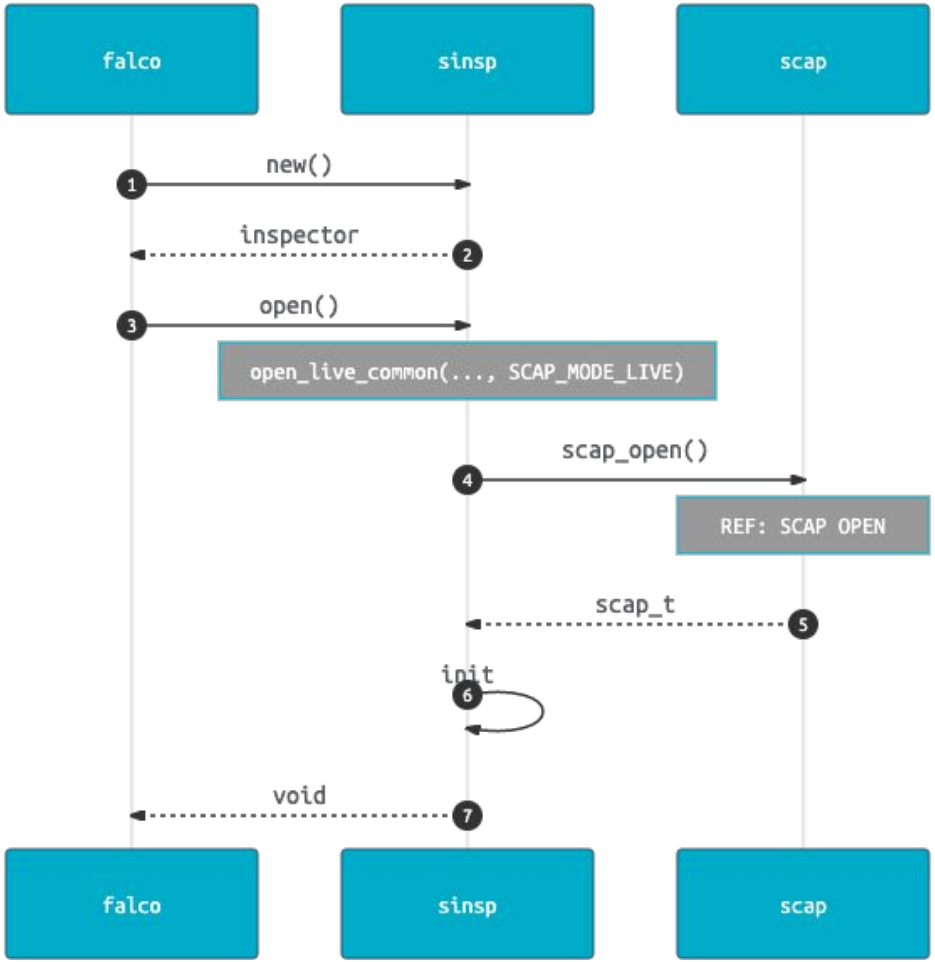
```
$ cmake -DBUILD_BPF=ON ..
> ...
$ make -j16 bpf
> Built target bpf
```

Load

It acts as the Falco inputs driver!

```
$ sudo FALCO_BPF_PROBE="" ./build/userspace/falco/falco -r rules/falco_rules.yaml
> ...
```

```
$ sudo FALCO_BPF_PROBE=build/driver/bpf/probe.o ./build/userspace/falco/falco -r
rules/falco_rules.yaml
> Thu Jul 23 19:45:27 2020: Falco version 0.24.0-4+4346e98f (driver version
85c88952b018fdbce2464222c3303229f5bfcfad)
> Thu Jul 23 19:45:27 2020: Falco initialized with configuration file
/home/vagrant/workspace/falcosecurity/falco/falco.yaml
> Thu Jul 23 19:45:27 2020: Loading rules from file rules/falco_rules.yaml:
> Thu Jul 23 19:45:28 2020: Starting internal webserver, listening on port 8765
```



When Falco starts...

Take a look at

- ❏ falco.cpp
- ❏ sinsp.cpp
- ❏ scap_open

libscap

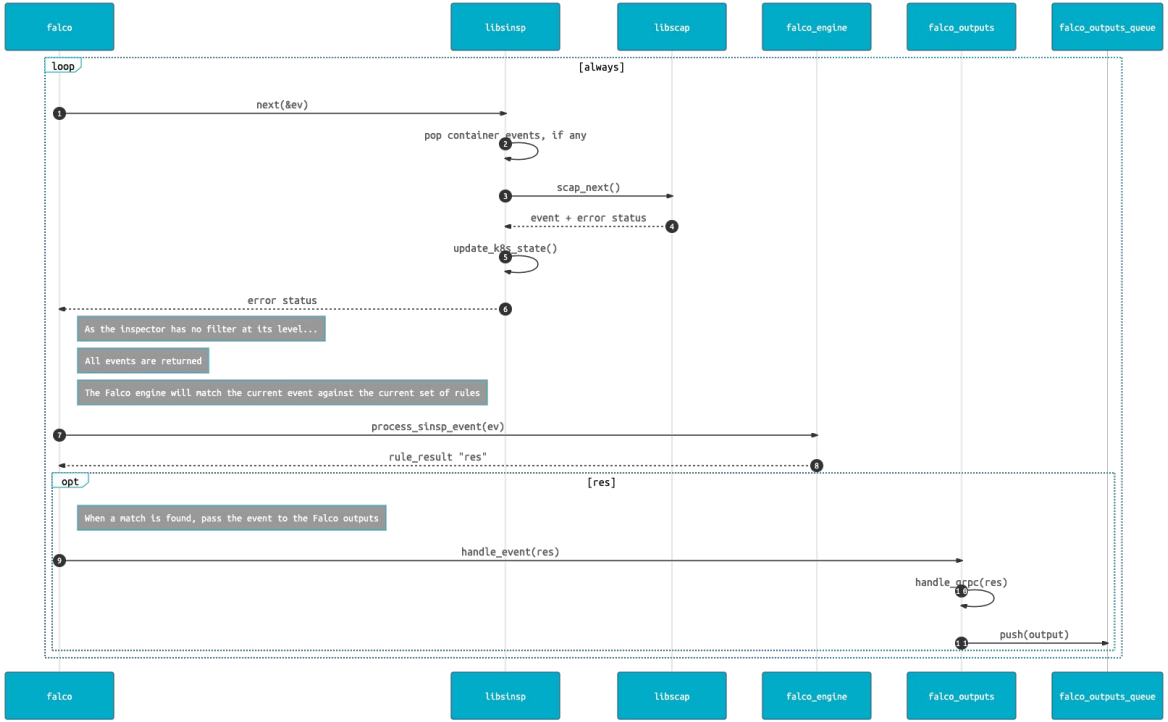
How actually loading an eBPF program looks like 👷

scap_open_live_int(), scap_bpf_load(), load_bpf_file(), load_elf_maps_section(), load_maps(), load_tracepoint(), populate_*_map() 🙏

1. collect machine info (# online cores), enable eBPF JIT, get iface, process and user list
2. parse the ELF of the eBPF object file
    a. check eBPF probe version matches Falco driver version
    b. look for “maps” sections and populate them
        i. SYSCALL_CODE_ROUTING_TABLE, SYSCALL_TABLE, EVENT_INFO_TABLE, FILLERS_TABLE, ...
    c. look for “tracepoint”, “raw_tracepoint” prefixed ELF sections
        i. load them: bpf() syscall (BPF_PROG_TYPE_TRACEPOINT or BPF_PROG_TYPE_RAW_TRACEPOINT)
        ii. attach them: open /sys/kernel/debug/tracing/events/<event>/id + ioctl(..., PERF_EVENT_IOC_SET_BPF), or bpf(BPF_RAW_TRACEPOINT_OPEN)
    d. “filler” prefixed ELF sections
        i. lookup FILLERS_TABLE and populate BPF_MAP_TYPE_PROG_ARRAY eBPF map
        ii. executed when corresponding syscall entry/exit (filler/<syscall-event>) get traced
3. scan “/proc” fs

Ready to start capturing!

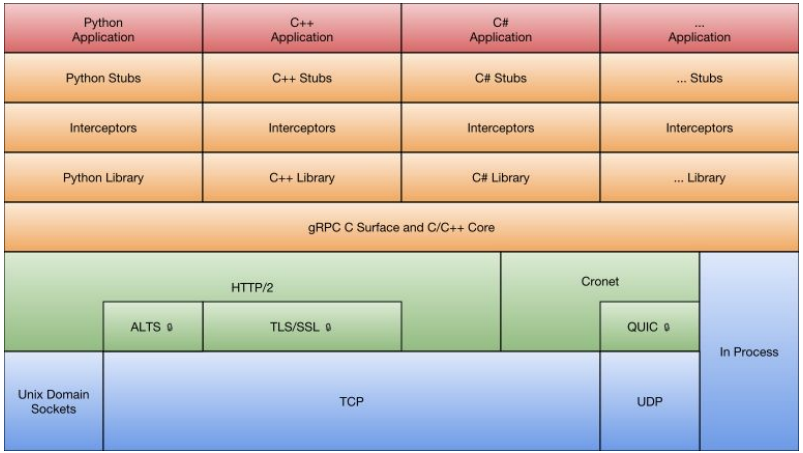
How the input events become alerts!



inspector->next(&ev), sinsp::next(), scap_next(), process_sinsp_event(), handle_grpc()

gRPC

- Working on top of HTTP2
  - stream multiplexing within single connection, …
- Streaming calls
  - client streaming, server streaming, both
- Implementations in many languages
- Authentication systems
- Strong protocol typing
  - protobuf, flatbuffer, …
- Many rich features
  - retries, flow control, cancellation, deadlines etc.



That's the question.

Sync

Pros
- ❏ simple to use and get going
- ❏ efficiency ok for most applications

Cons
- ❏ code can be called concurrently from multiple threads (same or different clients)
- ❏ worker thread occupied until RPC finishes
  - ❏ unary RPCs can block a thread for long time if handling requires blocking IO
  - ❏ long running streaming RPCs always block a thread

Async

man 7 epoll 📖

Pros
- ❏ bring your own **threading model**
- ❏ state-of-the-art at scaling
  - ❏ best performances for who's willing to go the extra mile

Cons
- ❏ implementing its API needs considerable boilerplate code
- ❏ some implicit behaviors not called out very well in the documentation

Falco gRPC Outputs API

falco.outputs.service/sub
Long-lived (bidi) streaming RPC

Get notified when some Falco rules violations happen and **wait**.

```
syntax = "proto3";

import "google/protobuf/timestamp.proto";
import "schema.proto";

package falco.outputs;

option go_package = "github.com/falcosecurity/client-go/pkg/api/outputs";

// This service defines the RPC methods
// to `request` a stream of output `response`s.
service service {
  // Subscribe to a stream of Falco outputs by sending a stream of requests.
  rpc sub(stream request) returns (stream response);
  // Get all the Falco outputs present in the system up to this call.
  rpc get(request) returns (stream response);
}

// The `request` message is the logical representation of the request model.
// It is the input of the `output.service` service.
message request {
  // TODO(leodido,fntlnz): tags not supported yet, keeping it for reference.
  // repeated string tags = 1;
}

// The `response` message is the representation of the output model.
// It contains all the elements that Falco emits in an output along with the
// definitions for priorities and source.
message response {
  google.protobuf.Timestamp time = 1;
  falco.schema.priority priority = 2;
  falco.schema.source source = 3;
  string rule = 4;
  string output = 5;
  map<string, string> output_fields = 6;
  string hostname = 7;
  // TODO(leodido,fntlnz): tags not supported yet, keeping it for reference.
  // repeated string tags = 8;
}
```

outputs.proto

Falco gRPC Outputs API

falco.outputs.service/get
Server streaming RPC

Get all the Falco rules violations happened and **stop**.

```
syntax = "proto3";

import "google/protobuf/timestamp.proto";
import "schema.proto";

package falco.outputs;

option go_package = "github.com/falcosecurity/client-go/pkg/api/outputs";

// This service defines the RPC methods
// to `request` a stream of output `response`s.
service service {
  // Subscribe to a stream of Falco outputs by sending a stream of requests.
  rpc sub(stream request) returns (stream response);
  // Get all the Falco outputs present in the system up to this call.
  rpc get(request) returns (stream response);
}

// The `request` message is the logical representation of the request model.
// It is the input of the `output.service` service.
message request {
  // TODO(leodido,fntlnz): tags not supported yet, keeping it for reference.
  // repeated string tags = 1;
}

// The `response` message is the representation of the output model.
// It contains all the elements that Falco emits in an output along with the
// definitions for priorities and source.
message response {
  google.protobuf.Timestamp time = 1;
  falco.schema.priority priority = 2;
  falco.schema.source source = 3;
  string rule = 4;
  string output = 5;
  map<string, string> output_fields = 6;
  string hostname = 7;
  // TODO(leodido,fntlnz): tags not supported yet, keeping it for reference.
  // repeated string tags = 8;
}
```

outputs.proto

Optimize

Tools + Benchmarks

- ❑ GRPC_TRACE
- ❑ gprof, pprof
- ❑ valgrind, mutrace
- ❑ experimental interceptors
- ❑ application benchmarks
- ❑ synthetic benchmarking

More at falco#1241

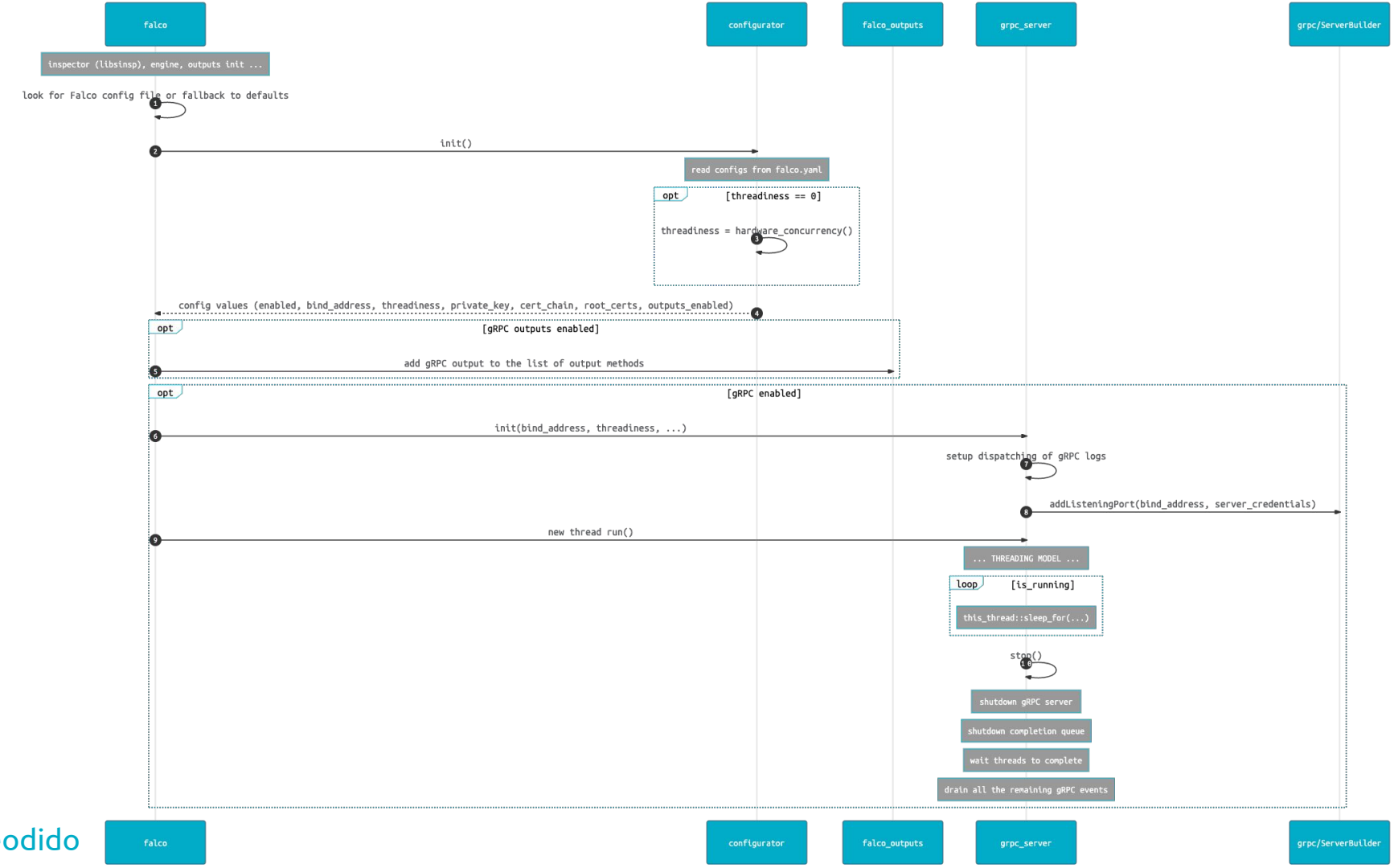
Suggestions

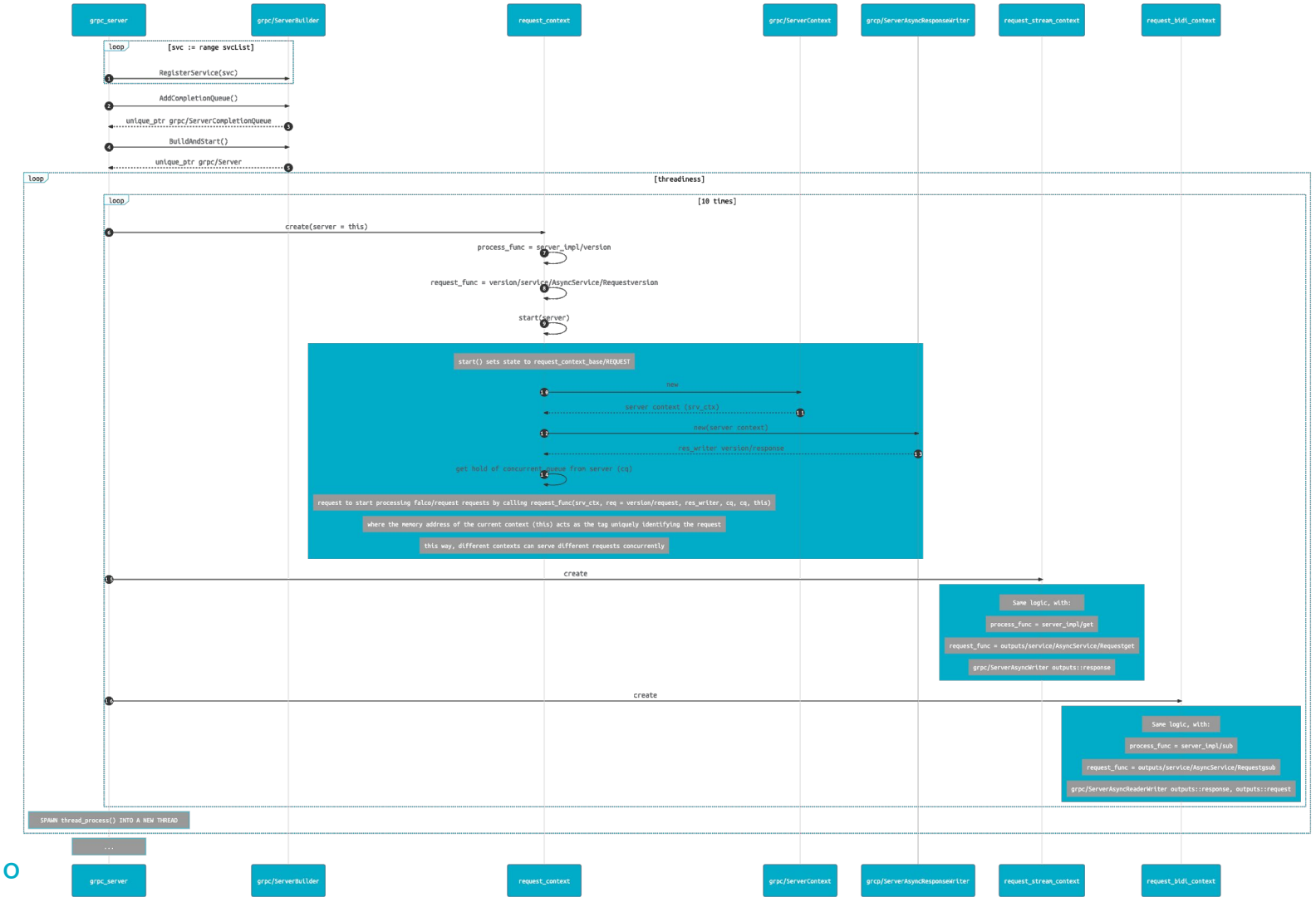
- ❑ Use the Async API
- ❑ Tune the threading model
- ❑ Tune the number of completion queues
- ❑ Reduce contention
- ❑ Reduce allocations
- ❑ Reduce copies
- ❑ Measure outstanding RPCs

Long running
bidirectional streaming.

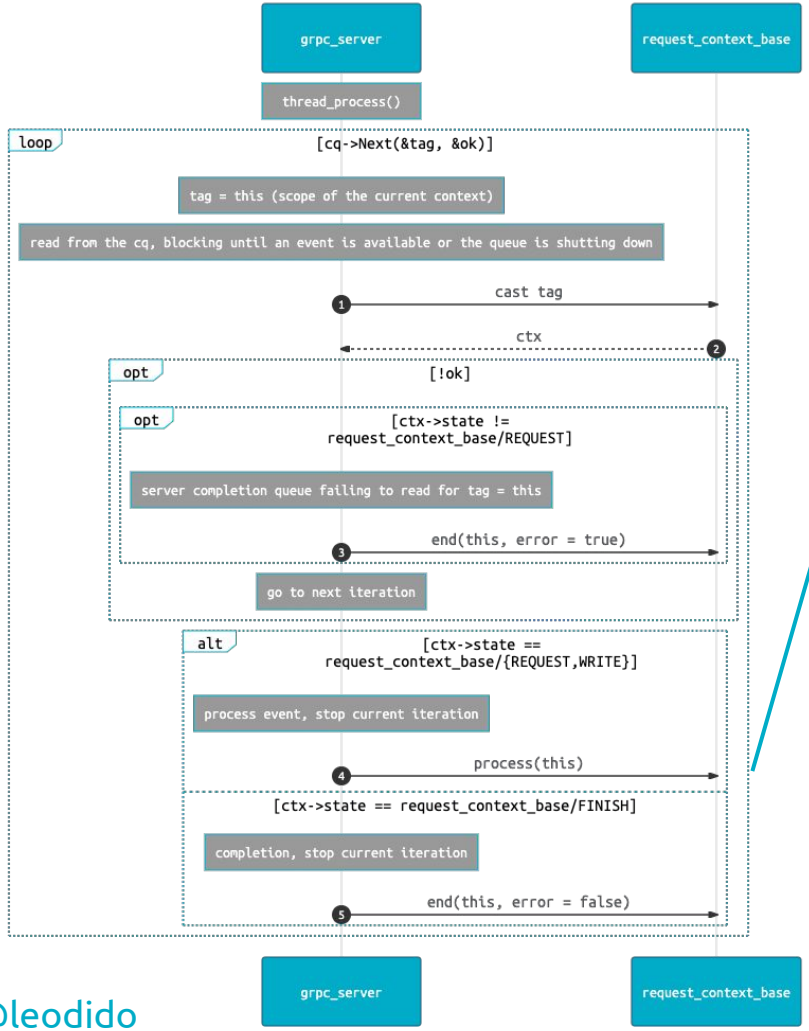
Or multiple unary RPC calls?

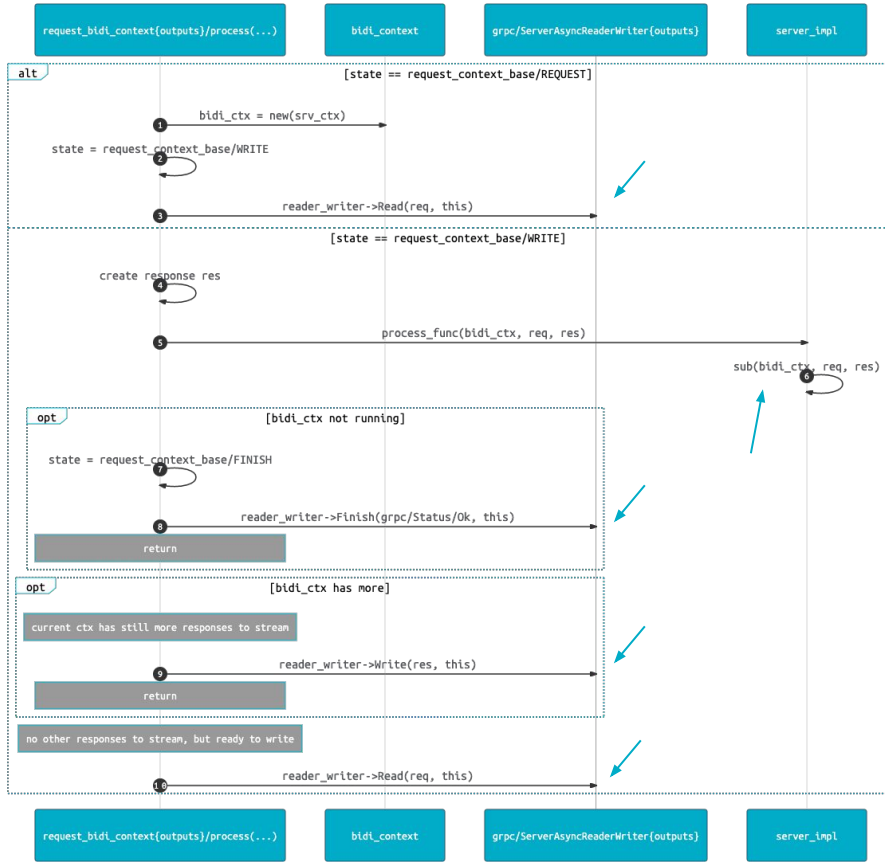
```
syntax = "proto3";

import "google/protobuf/timestamp.proto";
import "schema.proto";

package falco.outputs;

option go_package = "github.com/falcosecurity/client-go/pkg/api/outputs";

service service {
  rpc list(request) returns (response);
}

message request {}

message response {
  message output {
    google.protobuf.Timestamp time = 1;
    falco.schema.priority priority = 2;
    falco.schema.source source = 3;
    string rule = 4;
    string output = 5;
    map<string, string> output_fields = 6;
    string hostname = 7;
  }
  repeated output outputs = 1;
}
```


falco
configurator
falco_outputs
grpc_server
grpc/ServerBuilder
inspector (libsinsp), engine, outputs init ...
look for Falco config file or fallback to defaults
1
init()
2
read configs from falco.yaml
opt
[threadiness == 0]
threadiness = hardware_concurrency()
3
config values (enabled, bind_address, threadiness, private_key, cert_chain, root_certs, outputs_enabled)
4
opt
[gRPC outputs enabled]
add gRPC output to the list of output methods
5
opt
[gRPC enabled]
init(bind_address, threadiness, ...)
6
setup dispatching of gRPC logs
7
addListeningPort(bind_address, server_credentials)
8
new thread run()
9
... THREADING MODEL ...
loop
[is_running]
this_thread::sleep_for(...)
stop()
10
shutdown gRPC server
shutdown completion queue
wait threads to complete
drain all the remaining gRPC events
falco
configurator
falco_outputs
grpc_server
grpc/ServerBuilder


grpc_server
grpc/ServerBuilder
request_context
grpc/ServerContext
grcp/ServerAsyncResponseWriter
request_stream_context
request_bidi_context
loop
[svc := range svcList]
RegisterService(svc)
AddCompletionQueue()
unique_ptr grpc/ServerCompletionQueue
BuildAndStart()
unique_ptr grpc/Server
loop
[threadiness]
loop
[10 times]
create(server = this)
process_func = server_impl/version
request_func = version/service/AsyncService/Requestversion
start(server)
start() sets state to request_context_base/REQUEST
new
server context (srv_ctx)
new(server context)
res_writer version/response
get hold of concurrent queue from server (cq)
request to start processing falco/request requests by calling request_func(srv_ctx, req = version/request, res_writer, cq, cq, this)
where the memory address of the current context (this) acts as the tag uniquely identifying the request
this way, different contexts can serve different requests concurrently
create
Same logic, with:
process_func = server_impl/get
request_func = outputs/service/AsyncService/Requestget
grpc/ServerAsyncWriter outputs::response
create
Same logic, with:
process_func = server_impl/sub
request_func = outputs/service/AsyncService/Requestgsub
grpc/ServerAsyncReaderWriter outputs::response, outputs::request
SPAWN thread_process() INTO A NEW THREAD
...

@leodido


grpc_server
request_context_base
thread_process()
loop
[cq->Next(&tag, &ok)]
tag = this (scope of the current context)
read from the cq, blocking until an event is available or the queue is shutting down
cast tag
ctx
opt
[!ok]
opt
[ctx->state != request_context_base/REQUEST]
server completion queue failing to read for tag = this
end(this, error = true)
go to next iteration
alt
[ctx->state == request_context_base/{REQUEST,WRITE}]
process event, stop current iteration
process(this)
[ctx->state == request_context_base/FINISH]
completion, stop current iteration
end(this, error = false)
grpc_server
request_context_base

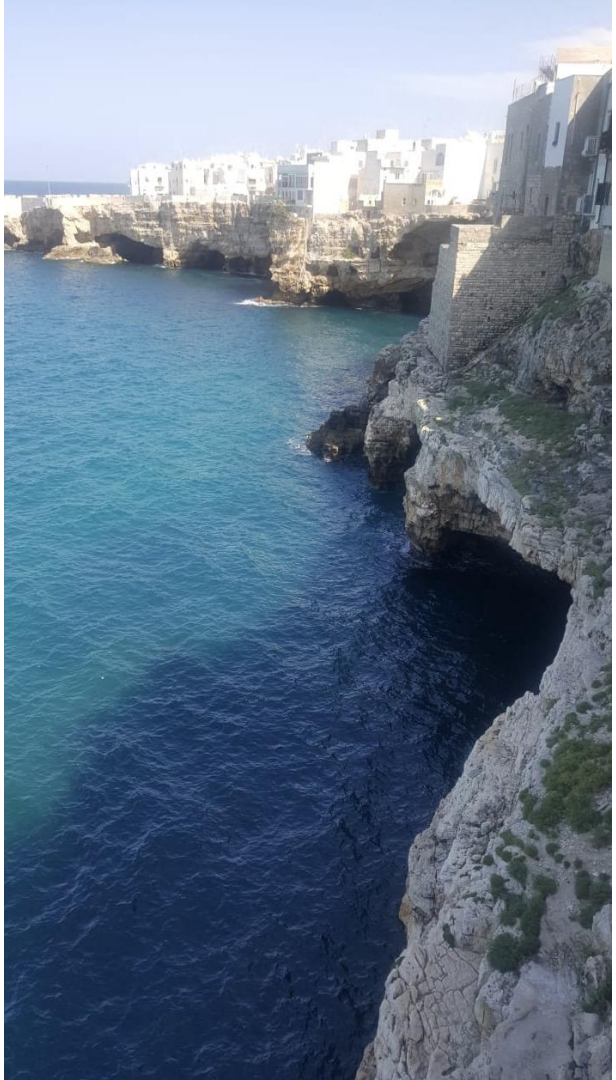

request_bidi_context{outputs}/process(...)
bidi_context
grpc/ServerAsyncReaderWriter{outputs}
server_impl
alt
[state == request_context_base/REQUEST]
bidi_ctx = new(srv_ctx)
state = request_context_base/WRITE
reader_writer->Read(req, this)
[state == request_context_base/WRITE]
create response res
process_func(bidi_ctx, req, res)
sub(bidi_ctx, req, res)
opt
[bidi_ctx not running]
state = request_context_base/FINISH
reader_writer->Finish(grpc/Status/Ok, this)
return
opt
[bidi_ctx has more]
current ctx has still more responses to stream
reader_writer->Write(res, this)
return
no other responses to stream, but ready to write
reader_writer->Read(req, this)
request_bidi_context{outputs}/process(...)
bidi_context
grpc/ServerAsyncReaderWriter{outputs}
server_impl

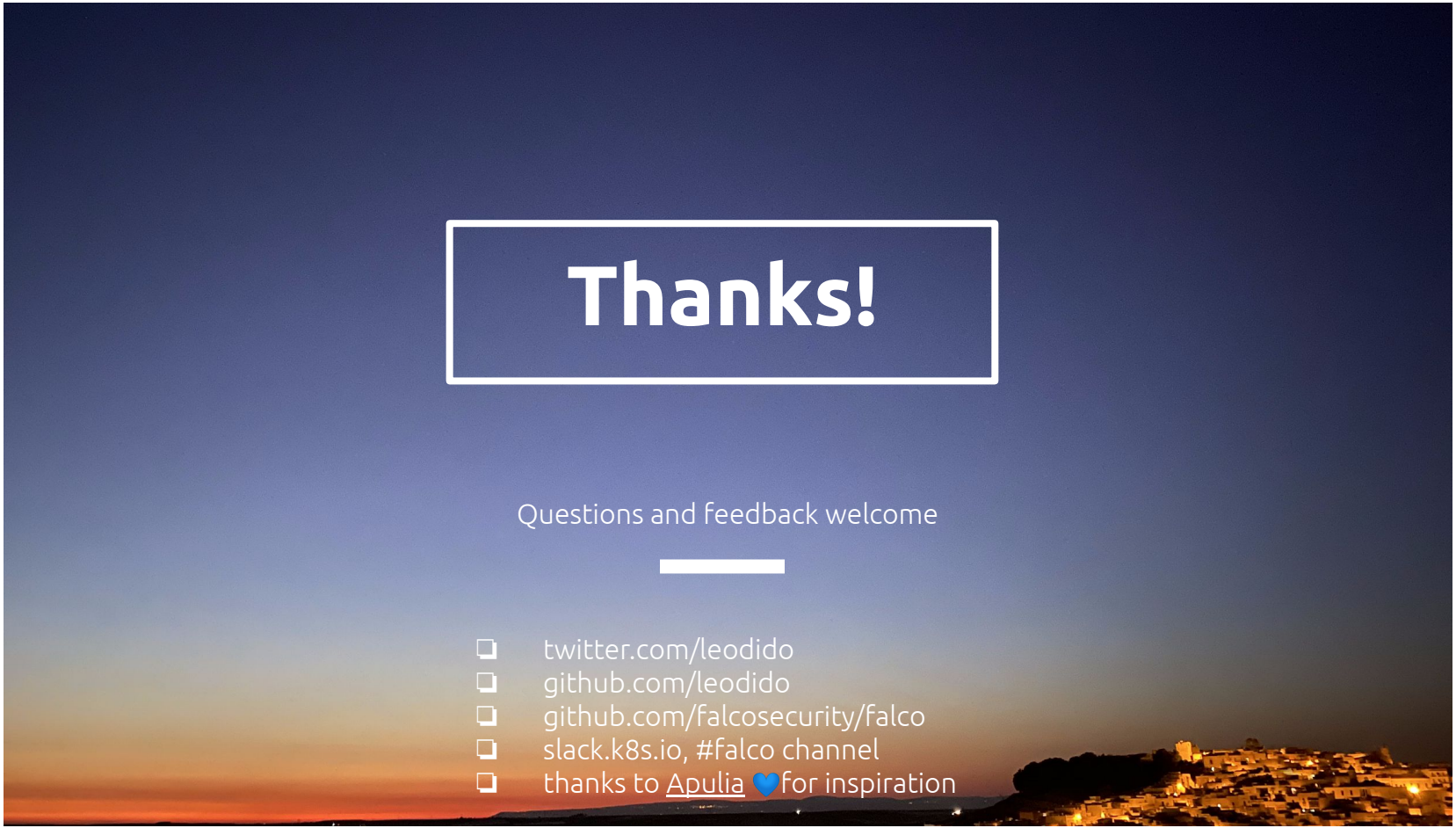
@leodido

Future work

- ❏ multiple unary RPCs rather than long running bidirectional?
- ❏ improve the half-duplexing of the existing bidirectional outputs RPC
- ❏ one output queue per session/context
- ❏ falcosecurity/client-go, client-py, client-rs
- ❏ go examples, asciinema py, asciinema rs

Contributors wanted! 🤗
Join the Falco community and help us! 💙




Thanks!
Questions and feedback welcome
twitter.com/leodido
github.com/leodido
github.com/falcosecurity/falco
slack.k8s.io, #falco channel
thanks to Apulia 💙for inspiration