Falco Internals 101

Syscalls processing for security analysis

Lorenzo Fontana, Sysdig

Author of "Linux Observability with BPF"
Maintainer of the Falcosecurity project

Syscalls processing

Security
Anomaly detection
Forensics
Stability

https://twitter.com/fntlnz
https://fntlnz.wtf

lo@linux.com

The history of Falco

Created as engine to parse libscap and libsinsp

Donated to CNCF by Sysdig (the company) in October 2018

C++

Falco Internals 101

Syscalls are very interesting

But there are too many

Throwing syscalls in userspace is challenging

And scary

And challenging

And scary

Lorenzo Fontana <lo@linux.com>

twitter.com/fntlnz - https://fntlnz.wtf

Syscalls are hard and slow to manage in userspace

Don't believe me? Do that in a system with millions of syscalls in a second

IT'S NOT 1998 ANYMORE

I mean, to know if a syscall was called you need to do a syscall

And of course to do time operations on the syscalls you need to do a syscall
Or hack with things like rdtsc

Syscalls are NOT enough

Context is important

- Did this syscall originated in a container?
- What's the name of the container ?
- What's the container image ?
- In which cluster it is running ?

libscap

Main duties

- Event capture and control
- File format management (.scap)
- Read and write dumps

libsinsp

Main duties

- Event parsing (sinsp_event)
- Container runtime metadata
- Kubernetes metadata
- Filtering

Falco Internals 101

Essentially, Falco loops on sinsp events

```
# for slides reasons some glue parts have been omitted
# include <sinsp.h>
int main(int argc, char **argv)
{
 # event pointer
 sinsp_evt* ev;
 while(1)
 {
   # next event
   rc = inspector->next(&ev);
   # process the event against the rules
   unique_ptr<falco_engine::rule_result> res = engine->process_sinsp_event(ev);
   if(res)
   {
       # send to the output if there's a match
       outputs->handle_event(res->evt, res->rule, res->source, res->priority_num, res->format);
   }
 }
}
```

How to get syscalls to userspace?

- Implement your own kernel module

- Use eBPF raw tracepoints

Kernel module: Interaction diagram

Kernel module: Ring Buffer

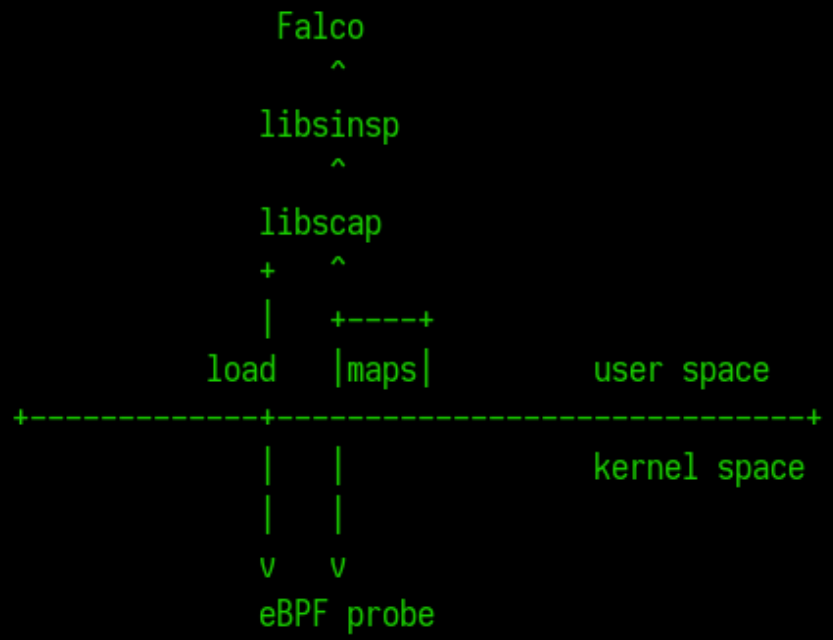
- 8 megabytes in memory
- event memory is dumped into it from the kernel
- one for each possible cpu
- userspace reads it from the /dev/falco0..N device

```
#ifndef PPM_RINGBUFFER_H_
#define PPM_RINGBUFFER_H_

#ifdef __KERNEL__
#include <linux/types.h>
#endif

static const __u32 RING_BUF_SIZE = 8 * 1024 * 1024;
static const __u32 MIN_USERSPACE_READ_SIZE = 128 * 1024;
struct ppm_ring_buffer_info {
 volatile __u32 head;
 volatile __u32 tail;
 volatile __u64 n_evts;   /* Total number of events that were received by the driver. */
 volatile __u64 n_drops_buffer;  /* Number of dropped events (buffer full). */
 volatile __u64 n_drops_pf;  /* Number of dropped events (page faults). */
 volatile __u64 n_preemptions;  /* Number of preemptions. */
 volatile __u64 n_context_switches; /* Number of received context switch events. */
};

#endif /* PPM_RINGBUFFER_H_ */
```

- Make sure that every matched syscall has consumable arguments
- There are generic ones
- Two for every syscall
- For every one of them a function is created at compile time

```
int f_sys_execve_e(struct event_filler_arguments *args)
{
 int res;
 unsigned long val;
 /*
  * filename
  */
 syscall_get_arguments_deprecated(current, args->regs, 0, 1, &val);
 res = val_to_ring(args, val, 0, true, 0);
 if (res == PPM_FAILURE_INVALID_USER_MEMORY)
  res = val_to_ring(args, (unsigned long)"<NA>", 0, false, 0);
 if (unlikely(res != PPM_SUCCESS))
  return res;
 return add_sentinel(args);
}
```

eBPF: interaction diagram



See driver/bpf/maps.h for FUN! Yes we load additional programs via maps.

eBPF: fillers

- Make sure that every matched syscall has consumable arguments
- There are generic ones
- Two for every syscall
- Based on data coming from syscalls static tracepoints (sys_*)

```
FILLER(sys_execve_e, true)
{
 unsigned long val;
 int res;
 val = bpf_syscall_get_argument(data, 0);
 res = bpf_val_to_ring(data, val);
 if (res == PPM_FAILURE_INVALID_USER_MEMORY) {
  char na[] = "<NA>";
  res = bpf_val_to_ring(data, (unsigned long)na);
 }
 return res;
}
```

Filtering

- Filtering is extremely important for performance reasons
- We only want to bring to userspace the syscalls we actually inspect
- Only when arguments are matching at kernel level already (still working on this!)

To summarize

- Getting tons of syscalls to userspace is heavy but optimizations can be made
- You can do it using an eBPF probe or a kernel module
- Good boundaries and well defined interfaces make life easier (libscap and libsinsp)

Join the community!

https://github.com/falcosecurity

https://falco.org

Lorenzo Fontana <lo@linux.com>