TraceLeft

A Configuration Driven eBPF Tracing Framework

Suchakra Sharma & Alban Crequy
All Systems Go, 29th September 2018, Berlin
Suchakra Sharma
Staff Scientist, ShiftLeft Inc.

Github: tuxology
Twitter: @tuxology
Email: suchakra@shiftleft.io

PhD, DORSAL Lab (Ecole Polytechnique de Montreal). Loves systems engineering, performance analysis, hardware tracing and runtime security.

Alban Crequy
CTO, Kinvolk GmbH.

Github: alban
Twitter: @albcr
Email: alban@kinvolk.io

Loves Kubernetes, networking, security, systemd and containers at the lower-levels of the system.
The Deep-stack Kubernetes Experts

Engineering services and products for Kubernetes, containers, process management and Linux user-space + kernel

Blog: kinvolk.io/blog
Github: kinvolk
Twitter: kinvolkio
Email: hello@kinvolk.io
Continuous Security for Cloud Native Apps

Secure applications by analyzing applications pre-emptively at build-time and carrying forward its security in production... seamlessly!

Blog: shiftleft.io/blog
Github: ShiftLeftSecurity
Twitter: ShiftLeftInc
Contact: shiftleft.io/contact
Agenda

- **Traceleft**
  - Background - Tracing 101
  - Architecture
  - Trace Configuration
  - JSON/Protobuf
  - Process/Store Trace Events

- **eBPF**
  - What is eBPF?
  - The eBPF programs & maps

- **Use cases**
  - Syscall monitoring example - demo ncurses demo
  - Event auditing - traceleft demo
Agenda

- Challenges
  - Recompilation
  - File operations
  - Network

- Future Work
  - Changes in recent kernel versions
  - Get rid of proc connector
Background

- **Tracing 101**
  - Low-impact recording on high frequency events such as syscalls, network events, scheduling, interrupts or process/container specific functions
  - Used for performance analysis and security
Think of your program as a bike with paint on tires, going down the street
System Tracing

- Tracing 101

Instrumentation
System Tracing

- Tracing 101
System Tracing

- Tracing 101
System Tracing

- **Examples**

  - *Static Tracing*: Kernel Tracepoints (Perf/Ftrace/eBPF), compile-time instrumentation (GCC/Clang), LTTng, USDT (Java, Python, Ruby)
  - *Dynamic Tracing*: Kprobes/Kretprobes (Ftrace/eBPF), Custom (Pin-tools, Dyninst) Uprobes (eBPF), Dtrace (BSD/MacOS)
System Tracing

- Code Instrumentation

<table>
<thead>
<tr>
<th>foo()</th>
<th>Observation Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>call_me_maybe()</td>
<td></td>
</tr>
<tr>
<td>bar()</td>
<td></td>
</tr>
<tr>
<td>baz()</td>
<td></td>
</tr>
<tr>
<td>call_me_maybe()</td>
<td>Collect Data</td>
</tr>
<tr>
<td></td>
<td>Fill Buffer</td>
</tr>
</tbody>
</table>

Event Event Event

Program Flow, Arguments, Latency, Data Flow
System Tracing - Kprobes

- Dynamic Instrumentation in Kernel

Supported by eBPF (stored in Maps → Perf Data), LTTng (CTF)
eBPF

Stateful, programmable, in-kernel decisions for networking, tracing and security
Berkeley Packet Filter

- **Classical BPF (cBPF)**
  - Network packet filtering [McCanne et al. 1993], Seccomp
  - Small, in-kernel VM. Register based, switch dispatch interpreter, few instructions

- **Extended BPF (eBPF)**
  - More registers, better verifier
  - Attach on Tracepoint/Kprobe/Uprobes/USDT
  - In-kernel trace aggregation & filtering
  - Control via `bpf()` , trace collection via BPF Maps/trace pipe
  - Upstream in Linux Kernel (`bpf()` syscall, kernel v3.18+)
  - Bytecode compilation upstream in LLVM/Clang
Berkeley Packet Filter

- eBPF Program
Berkeley Packet Filter

- eBPF + Kprobes
SEC("kprobe/tcp_set_state")
int kprobe__handle_tcp_set_state(struct pt_regs *ctx)
{
    u32 cpu = bpf_get_smp_processor_id();
    u64 pid_tgid = bpf_get_current_pid_tgid();
    u32 tgid = pid_tgid >> 32;
    int state = (int) PT_REGS_PARM2(ctx);

    tcp_event_t ev = {
        .timestamp = bpf_ktime_get_ns(),
        .tgid = tgid,
        .state = state,
        ...
    };

    bpf_perf_event_output(ctx, &events, cpu, &ev, sizeof(ev));
}
/* This is a key/value store with the keys being the cpu number
 * and the values being a perf file descriptor. */

struct bpf_map_def SEC("maps/events") events = {
    .type = BPF_MAP_TYPE_PERF_EVENT_ARRAY,
    .key_size = sizeof(int),
    .value_size = sizeof(__u32),
    .max_entries = 1024,
    .map_flags = 0,
    .pinning = PIN_GLOBAL_NS,
    .namespace = "traceleft",
};

typedef struct {
    uint64_t timestamp;
    int64_t  tgid;
    int64_t  state; ...
} tcp_event_t;
TraceLeft

https://github.com/ShiftLeftSecurity/traceleft
TraceLeft Overview

- **What’s TraceLeft?**
  - Framework to build syscall, network & file auditing or monitoring tools
  - eBPF+Kprobes based, supported from kernel v4.4+
  - Also a binary, *traceleft* that is a reference implementation
  - Can generate a single binary - with a modular trace *battery*
  - Everything is compiled based on detailed event configuration and platform information

- **Why?**
  - Configurable event tracing that *Just Works™* ...*coughs*
  - Programmable tracing, supported on older kernels
Components

![Diagram showing components and their interactions](image-url)
Components

- **Metagenerator**
  - Generated C and Go structures for each event to be received
  - Goes through `/sys/kernel/debug/tracing/events/syscalls/*` and generates structures

- **Generator**
  - Generates the eBPF handler program sources in C

- **Battery**
  - Compiled eBPF programs battery (a kernel \texttt{v4.4} pre-compiled battery has been tested to work till kernel \texttt{v4.16})
Components

- **Probe**
  - Responsible for registering and unregistering eBPF handlers.

- **Tracer**
  - Loads the probe, starts polling the events perf map and calls the callback for each received event.

- **Metrics Aggregator**
  - Experimental event aggregation code that allows processing of raw trace events generated by TraceLeft.
Components

- Configuration
  - A fine-grained per-event configuration that defines each BPF handler’s event structure
  - What all to collect from each probe along with type info, variable names
  - Can be eventually simplified to avoid duplication

```
"event": [
  {
    "name": "open",
    "args": [{
      "position": 1,
      "type": "char",
      "name": "filename",
      "hashFunc": "string",
      "suffix": "[256]"
    },
    {
      "position": 2,
      "type": "s64",
      "name": "flags"
    },
    {
      "position": 3,
      "type": "u64",
      "name": "mode"
    }
  ]
```
Components

- **Aggregation Spec**
  - Defines how each event collected should be aggregated, filtered and transmitted or stored
  - **Channels**: Where to store/send events,
  - **Function**: How to process input event stream),
  - **Rule**: Filter applied to event aggregation

```json
"channels": [
  {
    "id": "1",
    "type": "file",
    "path": "/tmp/traceleft.log"
  },
  {
    "id": "2",
    "type": "grpc",
    "path": "localhost:50051"
  }
],
"events": [
  {
    "name": "open",
    "channel": "1",
    "stream": "filesystem",
    "group": "system_metrics",
    "rule": "arg1 == '/tmp/a.txt'",
    "function": {
      "id": "sigma",
      "parameters": "frequency=100;threshold=0"
    },
    "output": {
      "metrics": "alerts_per_sec",
      "format": "collector_spec_pb"
    }
  }
]
```
Use Cases

- **traceleft CLI**
  - Simple syscall logging and auditing system

```plaintext
name open pid 5518 program id 0 return value 8 hash 3355305515321265881
Filename "/etc/passwd" Flags 524288 Mode 438

name open pid 5518 program id 0 return value 8 hash 3355305515321265881
Filename "/etc/passwd" Flags 524288 Mode 438

name open pid 5522 program id 0 return value 11 hash 10268694621493151422
Filename "/proc/sys/kernel/ngroups_max" Flags 0 Mode 0

name open pid 5522 program id 0 return value 11 hash 5259532013223916043
Filename "/etc/group" Flags 524288 Mode 438
```
Use Cases

- **Syscall Monitoring Agent**
  - Sample implementation for a ncurses based live syscall monitoring example using TraceLeft aggregation API
Challenges

Matching pids and applications
Matching pids and applications

- What’s an application?
  - One or more processes. Might be short-lived (shell scripts)

- Application running as a systemd unit
  - In a different cgroup
  - Maybe in different namespaces

- Application running in a container
  - In a different cgroup
  - In different namespaces
Matching pids and applications

<table>
<thead>
<tr>
<th>BPF helper function</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>bpf_get_current_pid_tgid()</td>
<td>4.2</td>
</tr>
<tr>
<td>bpf_get_cgroup_classid()</td>
<td>4.3 (network)</td>
</tr>
<tr>
<td>bpf_current_task_under_cgroup()</td>
<td>4.9</td>
</tr>
<tr>
<td>bpf_get_current_cgroup_id()</td>
<td>4.18 + cgroup-v2</td>
</tr>
<tr>
<td>bpf_get_current_pidns_info()</td>
<td>Future (4.20+?)</td>
</tr>
</tbody>
</table>

https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md
Using the Traceleft API

- Register handlers by PID
  
  ```go
  func (probe *Probe) RegisterHandlerById
  (programID uint64, pid int, hash string) error
  ```

- Matching the app and the pid externally
  - Using Linux’ proc connector
Proc connector

- Connector: sub-family of Netlink
  
  ```
  socket(AF_NETLINK, SOCK_RAW, NETLINK_CONNECTOR);
  ```

- Subscribe to proc events
  
  ```
  sendmsg(sockfd, ...PROC_CN_MCAST_LISTEN...);
  ```

- Receive notifications for fork, exec, exit
- Since Linux v2.6.15 (January 2006)
Proc connector limitations

- Only works in init users, pidns, with net privileges
- Can’t keep track of namespaces or cgroups
- Need to check in /proc, asynchronously
  - /proc/$PID/{exe,comm,cgroup,ns}
- Races
  - Short-lived processes: can’t read procfs fast enough
  - Missing early events that happened before the BPF handler was installed
Solutions

- Avoid
  - Procfs
  - Proc connector

- Using new BPF helpers
  - Add new BPF helpers upstream if needed
Challenges

Strings in eBPF
Reporting strings

- **Example with `open()` syscall**

  **In userspace:**
  ```c
  int open(const char *pathname, int flags);
  ```

  **In the eBPF kprobe:**
  ```c
  ret = bpf_probe_read(&evt.filename, sizeof(evt.filename),
                      (void *) PT_REGS_PARM2(args));
  ```

  **In kernel:**
  ```c
  len = strncpy_from_user(kname, filename, EMBEDDED_NAME_MAX);
  ```
Problems with strings

- **Time of check to time of use (TOCTOU)**
  - Buffer copied twice from userspace
  - Multithreaded programs could alter the buffer in the middle
  - Same issue as seccomp
Problems with strings

- **Cannot find the size of the string**
  - `probe_read_str()` only in Linux 4.11
  - TraceLeft copies 256 bytes
  - Might be too little

- **Danger of reading too much**
  - A page border might cause `EFAULT`
  - `open()` use NULL-terminated strings

```c
fd = open(ptr, flags);
```

```
<table>
<thead>
<tr>
<th>virtual memory of a process</th>
<th>mmap’ed region</th>
</tr>
</thead>
<tbody>
<tr>
<td>256 bytes</td>
<td></td>
</tr>
</tbody>
</table>
```
Challenges

Identifying files
File descriptors

```c
fd = open("/data/foo.txt", O_RDWR);
fd2 = dup(fd);
ret = write(fd2, buf, sz);
```

Keeping track of file descriptors per process
How processes receive a file descriptor

- open(), openat()...
- SCM_RIGHTS
- dup(), dup2(), dup3()
Path lookups

- All the string problems from before
- Path lookups depends on:
  - mntns
  - root, cwd, or dirfd with openat()
  - at every components, possible symlink
  - Cannot be evaluated atomically from eBPF

```c
fd = open("/data/foo.txt", O_RDWR);
```
Solutions?

- **Landlock-LSM?**
  - eBPF programs acting on kernel objects instead of strings
  - More programmable actions (resource control)
Challenges

Networking
Correlating IPs with services

- Destination IP visible at the syscall level

```c
ret = connect(sockfd, { IP: 192.168.0.40 } );
```

- But not the full connection tuple
- We add kprobes on `inet_csk_accept()`, `tcp_set_state()`, `tcp_close()`, `tcp_v4_connect()`
Challenges

Lost events: perf ring buffer and kretprobes
Losing events in the perf ring buffer

- **Events sent asynchronously**
  - BPF programs cannot sleep or wait
  - Ring buffer has limited size
  - Default in traceleft: 8 pages (32KiB) per cpu
  - `bpf_perf_event_output()` just overwrites previous entries
  - Counter of lost events
Missing kretprobes

- **How kprobes work**
  - Place break exception (or jump) on function entry

- **How kretprobes work**
  - Place break exception on function entry
  - Save the return address of function and replace it by a trampoline
  - The trampoline does its job and then return to the original address
- **Multiple CPUs, preemptible kernels**
  - There could be several function calls in parallel
  - Need to save several return addresses
  - Example: a synchronous `accept()` syscall

- **maxactive**
  - Default value:
    ```c
    rp->maxactive = max_t(unsigned int, 10, 2*num_possible_cpus());
    ```
  - Since Linux 4.12 (commit 696ced4fb1d7), configurable
  - In TraceLeft, we chose maxactive=16
Future work
Future Work

- Use tracepoints
  - Benefit from more stable API

- Use new BPF helper functions
  - bpf_get_current_cgroup_id
  - bpf_probe_read_str

- Use LLVM API directly
  - Avoid using clang, generation of sources etc.
References
Related Work

- **IOVisor/BPF**
  - BCC ([https://github.com/iovisor/bcc](https://github.com/iovisor/bcc))
  - bpfd ([https://github.com/genuinetools/bpfd](https://github.com/genuinetools/bpfd))
  - BPFd ([https://github.com/joelagnel/bpfd](https://github.com/joelagnel/bpfd)) [Deprecated]
  - BpfTrace([https://github.com/ajor/bpftrace](https://github.com/ajor/bpftrace))
  - Ply ([https://github.com/iovisor/ply](https://github.com/iovisor/ply))
  - Landlock LSM ([https://landlock.io/](https://landlock.io/))

- **Auditd**
  - Architecture ([https://goo.gl/zXdfsJ](https://goo.gl/zXdfsJ))
Documentation and Links

- **BPF Docs/Tutorials**
  - [https://github.com/zoidbergwill/awesome-ebpf](https://github.com/zoidbergwill/awesome-ebpf) (William Martin Stewart)
  - [http://www.brendangregg.com/ebpf.html](http://www.brendangregg.com/ebpf.html) (Brendan Gregg)
Research Papers

Research Papers

- [Borkmann 2016-1] On getting tc classifier fully programmable with cls bpf, NetDev 1.1 (2016), Seville