Linux security event monitoring with osquery

QueryCon 2019

Alessandro Gario

Senior Security Engineer

alessandro.gario@trailofbits.com

www.trailofbits.com

Overview

1. Event-based tables on Linux
2. Audit 101
3. The next big thing
4. What's eBPF
5. Journey from zero to `process_events`

Disclaimer: I like Spaceballs

State of the event-based tables

hardware_events, syslog_events

Awesome!

- Low memory usage
- Not many events to process
- Low CPU usage

file_events

Kind of annoying:

- Watchers have to be updated as events come in
- Relies on (globbing) existing files
- Prone to losing events
- No way to know if events were lost

How to break file_events

Example

```
$ cd /monitored
$ mkdir -p 1/2/3/4/5 && \
  date > 1/2/3/4/5/hidden_file
```

How to make file_events lose changes

Audit tables

Interesting:

- Good insight on each event
- Can monitor most things

Not perfect:

- Uses a lot of memory
- Consumes a lot of CPU

Can we do better?

Data sources alone determine the fate of the table's quality, **not the actual code**:

- How much memory is used?
- How much processing is required?
- **Can events be trusted?**

Audit 101

What is Audit?

A system tracing utility

- Syscalls
- System events

Used by most event-based tables:

- process_events
- socket_events
- user_events
- selinux_events
- process_file_events

What is wrong with Audit?

NOTHING!
Teddy and I wrote it

If you don't like it, you are
WRONG

What is **actually** wrong with Audit?

- Only one Audit consumer*
- Text-based
- Multiple records need to be a aggregated to create event context
- High memory footprint
- High CPU usage

The next big thing

Finding the next big thing

What would we like?

- Event tracing
- Syscall tracing
- Context information for each event
- Binary data instead of text walls

I've heard about a thing called eBPF

AMAZING

- Tracepoints!
- More tracepoints! Kprobes! Uprobes!
- Not much context information!
- Binary data! Finally!

eBPF looks like a good candidate!

What's eBPF

eBPF 101

A technology to load arbitrary programs and have them run when a specific event occurs:

- Tracepoints: manually defined in the source, stable interface
- kprobes: basically code hooking

More data sources exist, but we are only interested in the first two

eBPF 102

- eBPF programs are:
  - compiled into bytecode
  - Sandboxed
  - Verified kernel-side upon load

Can be built:

- Manually, with raw BPF opcodes
- Official toolchain

BPF Compiler Collection (BCC)

A toolkit for creating and compiling eBPF programs:

- developed by IOVisor,
- offers kernel instrumentation in C,
- has front-ends in Python and Lua,
- built on top of LLVM and Clang

Journey from zero to process_events

What's inside `process_events` anyway?

Many fields, but let's start with the following ones:

- pid
- path
- cmdline

Our initial implementation

```c
#include <uapi/linux/ptrace.h>
#include <uapi/linux/limits.h>

typedef struct {
  u32 pid;
  char filename[NAME_MAX]; // 256 bytes
} ExecveData;

BPF_PERF_OUTPUT(events);

int sys_enter_execve(struct tracepoint__syscalls__sys_enter_execve *args)
{
  ExecveData execve_data = {};
  execve_data.pid = (u32) (bpf_get_current_pid_tgid() >> 32);

  // We can't directly access user memory
  bpf_probe_read(&execve_data.filename,
                 sizeof(execve_data.filename),
                 args->filename);

  events.perf_submit(args, &execve_data, sizeof(ExecveData));
  return 0;
};
```

Our initial implementation

```c
#include <uapi/linux/ptrace.h>
#include <uapi/linux/limits.h>

typedef struct {
  u32 pid;
  char filename[NAME_MAX]; // 256 bytes
} ExecveData;

BPF_PERF_OUTPUT(events);

int sys_enter_execve(struct tracepoint__syscalls__sys_enter_execve *args)
{
  ExecveData execve_data = {};
  execve_data.pid = (u32) (bpf_get_current_pid_tgid() >> 32);

  // We can't directly access user memory
  bpf_probe_read(&execve_data.filename,
                 sizeof(execve_data.filename),
                 args->filename);

  events.perf_submit(args, &execve_data, sizeof(ExecveData));
  return 0;
};
```

Our initial implementation

```c
#include <uapi/linux/ptrace.h>
#include <uapi/linux/limits.h>

typedef struct {
  u32 pid;
  char filename[NAME_MAX]; // 256 bytes
} ExecveData;

BPF_PERF_OUTPUT(events);

int sys_enter_execve(struct tracepoint__syscalls__sys_enter_execve *args)
{
  ExecveData execve_data = {}; // Declare a new struct on stack
  execve_data.pid = (u32) (bpf_get_current_pid_tgid() >> 32);

  // We can't directly access user memory
  bpf_probe_read(&execve_data.filename,
                 sizeof(execve_data.filename),
                 args->filename);

  events.perf_submit(args, &execve_data, sizeof(ExecveData));
  return 0;
};
```

Our initial implementation

```c
#include <uapi/linux/ptrace.h>
#include <uapi/linux/limits.h>

typedef struct {
  u32 pid;
  char filename[NAME_MAX]; // 256 bytes
} ExecveData;

BPF_PERF_OUTPUT(events);

int sys_enter_execve(struct tracepoint__syscalls__sys_enter_execve *args)
{
  ExecveData execve_data = {};
  execve_data.pid = (u32) (bpf_get_current_pid_tgid() >> 32);

  // We can't directly access user memory
  bpf_probe_read(&execve_data.filename,
                 sizeof(execve_data.filename),
                 args->filename);

  events.perf_submit(args, &execve_data, sizeof(ExecveData));
  return 0;
};
```

First challenge

The **filename** parameter is truncated at 256 bytes.

You **COULD** increase the array size, but here's the thing: stack is limited to 512 bytes.

Can we do better?

First workaround: PER-CPU maps to the rescue!

```
BPF_PERCPU_ARRAY(temp_execve_data,
                 ExecveData, 1);
...
int index = 0;

// Make sure to check for NULL values!
ExecveData *execve_data_ptr =
    temp_execve_data.lookup(&index);
```

Second challenge: other parameters?

We only have the binary name!
What about program arguments?

Let's take a look at two possible workarounds:

- Use a bigger map
- Create additional maps

Second workaround/a: Using bigger maps

```
typedef struct {
  u32 pid;
  char filename[512];

  char param1[512];
  char param2[512];
  char param3[512];
  ...
} ExecveData;
```

Too much space across perf_events. Will make it easy to lose events.

Second workaround/b: Using additional maps

Step one: data map

```
typedef struct {
  char bytes[2048];
} StringBuffer;

PER_CPU_ARRAY(
  string_data,
  StringBuffer,
  1000
);
```

Step two: index map

```
PER_CPU_ARRAY(
  string_data_index,
  int,
  1
);
```

Step three: event object

```
typedef struct {
  u32 pid;
  char filename[512];
  int parameters[20];
} ExecveData;
```

Challenge 3

We are still only getting N parameters!

String size is still limited!

Workaround 3

NONE :(

Additional eBPF limitations

- Jumps can only go forward
- Only 4096 instructions per program

Different approaches

- Dedicated tracepoints
- Deeper inspection with kprobes

Conclusions

Conclusions

- Audit is not that bad!
- eBPF is hard
- Using eBPF like we use Audit doesn't work
- Teddy is a super hero