bpftrace: a path to the ultimate Linux tracing tool

FOSDEM 2025

Viktor Malík
Red Hat

Feb 1, 2025

Introduction

- Who am I?
  - Viktor Malík <vmalik@redhat.com>
  - Principal Software Engineer at Red Hat, Core Kernel Engineering
  - Upstream co-maintainer of **bpftrace**

Red Hat

- What am I doing here?
  - Introduce bpftrace as a tracing tool and language
  - Show what bpftrace can do for you
  - Tease what bpftrace could do for you in future

bpftrace introduction

bpftrace

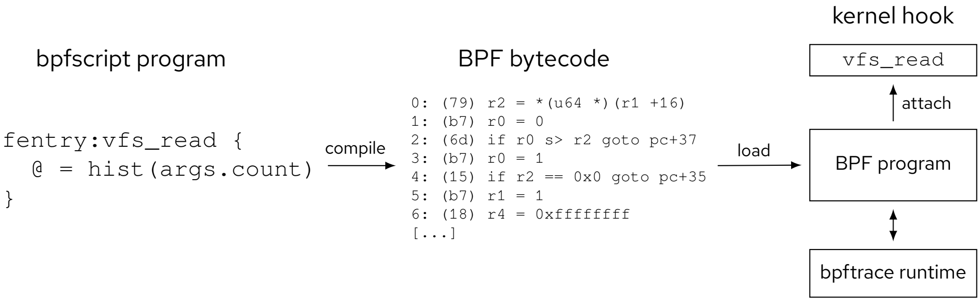
Quick introduction

- Tracing tool for Linux based on eBPF
- Comes with a domain-specific laguage, bpfscript

bpftrace

Quick introduction

- Tracing tool for Linux based on eBPF
- Comes with a domain-specific laguage, bpfscript
- Basic workflow:



bpfscript

Language overview

- Main building block is a **probe**
- Probes have 2 main parts:

`fentry:vfs_read`	attach point
`{ @ = hist(args.count) }`	action block

```
fentry:vfs_read
{
  @ = hist(args.count)
}
```


bpftrace

bpfscript

Language overview

- Main building block is a **probe**
- Probes have 2 main parts:

`fentry:vfs_read`	**attach point**
`{ @ = hist(args.count) }`	action block

```
fentry:vfs_read
{
  @ = hist(args.count)
}
```

bpfscript

Language overview

- Main building block is a **probe**
- Probes have 2 main parts:

```
fentry:vfs_read
{
  @ = hist(args.count)
}
```

attach point

action block

bpfscript

Language overview

- Main building block is a **probe**
- Probes have 2 main parts:

```
fentry:vfs_read              attach point
{
  @ = hist(args.count)       action block
}
```

- Main action block constructs:
  - Variables
  - Operators (arithmetic, logic, bitwise, struct/array member access, ...)
  - Control-flow statements (conditionals, loops)
  - Built-in variables and functions

bpfscript

Variables

- bpftrace provides two kinds of variables:
    - Scratch variables
        - Block-scoped (valid only inside the current lexical block)
        - Example: `$x = cpu`
    - Maps
        - Key-value pairs
        - Globally-scoped (each map is available from all probes)
        - Implemented using BPF hash maps
        - Example: `@start[pid] = nsecs`

bpftrace

bpfscript

Built-ins

- Built-ins are special variables and functions built into the language.
- Provide various functionalities such as access to kernel data, type conversions, printing, string manipulation, etc.

bpftrace strengths

bpftrace strengths

One-liners

- The terseness of the language allows to write powerful one-liner scripts that can be tailored to user's immediate needs.
- Great for on-the-fly debugging of production systems.

bpftrace strengths

One-liners

- The terseness of the language allows to write powerful one-liner scripts that can be tailored to user's immediate needs.
- Great for on-the-fly debugging of production systems.
- Example: list files opened by thread name

```
t:syscalls:sys_enter_open { printf("%s %s\n", comm, str(args.filename)) }
```

bpftrace strengths

Abstraction from BPF

- BPF has many powerful features but sometimes requires significant expertise to be used.
- bpftrace tries to eliminate this by abstracting the implementation details away from the user.
- This makes bpftrace a great choice as the entry point to the BPF world.

bpftrace strengths

Abstraction from BPF

- BPF has many powerful features but sometimes requires significant expertise to be used.
- bpftrace tries to eliminate this by abstracting the implementation details away from the user.
- This makes bpftrace a great choice as the entry point to the BPF world.
- Example:
  - BPF stack is only 512 B so it is often necessary to offload values to BPF maps or global variables.
  - Creating maps in BPF is not entirely easy:

```
struct {
  __uint(type, BPF_MAP_TYPE_ARRAY);
  __uint(max_entries, 1);
  __type(key, u32);
  __type(value, pid_t);
} my_map SEC(".maps");
```

  - bpftrace will **automatically offload** large scratch variables to maps

bpftrace

bpftrace weak spots

bpftrace weak spots

- BPF feature completeness
  - A number of BPF features is not exposed via bpftrace/bpfscript.
  - Missing are some helpers, pretty much all the kfuncs, map types, CO-RE, ...

bpftrace weak spots

- BPF feature completeness
  - A number of BPF features is not exposed via bpftrace/bpfscript.
  - Missing are some helpers, pretty much all the kfuncs, map types, CO-RE, …
- Complex scripts
  - bpftrace has traditionally been targeting one-liners.
  - Writing and maintaining larger scripts is often painful due to the lack of features – CLI options, functions (subprograms), …

bpftrace weak spots

- Tracing capabilities
  - Some events and environments are notoriously hard to trace
  - Examples:
    - inlined functions (both in kernel and userspace)
    - running the tracer in containers/namespaces
  - These problems are not specific to bpftrace

Recent, ongoing, and future work

Recent, ongoing, and future work

Better tracing capabilities

- Tracing inlined functions
  - Using lldb to resolve all locations of a function entry (including inlined) from DWARF
  - This also allows to place a probe after the function prologue (when the stack frame has been established) which will prevent missing entries when collecting stacks.

Recent, ongoing, and future work

Better tracing capabilities

- Tracing inlined functions
  - Using lldb to resolve all locations of a function entry (including inlined) from DWARF
  - This also allows to place a probe after the function prologue (when the stack frame has been established) which will prevent missing entries when collecting stacks.
- Running bpftrace inside a container with PID namespacing
  - In a PID namespace, `pid`, `tid`, and `ustack` do not work correctly
  - We must switch between different helpers (`bpf_get_current_pid_tgid`, `bpf_get_ns_current_pid_tgid`) depending on where bpftrace and the traced process are running.
  - Still not working for the case when bpftrace is in a child namespace while the target is in the root namespace.

Recent, ongoing, and future work

Variable/map declarations

- Automatic type inference is good for one-liners but can make scripts harder to maintain.
- Sometimes it is necessary to specify the variable type, especially for maps.

Recent, ongoing, and future work

Variable/map declarations

- Automatic type inference is good for one-liners but can make scripts harder to maintain.
- Sometimes it is necessary to specify the variable type, especially for maps.

- Scratch variable declarations:

```
let $x: uint8;
$x = 0;
```

Recent, ongoing, and future work

Variable/map declarations

- Automatic type inference is good for one-liners but can make scripts harder to maintain.
- Sometimes it is necessary to specify the variable type, especially for maps.

- Scratch variable declarations:

```
let $x: uint8;
$x = 0;
```

- Map declarations (not yet implemented):

```
let @hash = Hash<uint32, int64>;
@hash[pid] = nsecs;

let @array = Array<int64>;
@array[0] = 100;
```


bpftrace

Recent, ongoing, and future work

Functions

- Migration to libbpf enabled usage of BPF subprograms.
- In future, bpftrace should support calling 2 new kinds of functions:
  - Defined in bpfscript:

```
fn sum(a: int64, b: int64): int64 {
  return $a + $b;
}
```

  - Imported from external BPF programs/libraries
    - Useful e.g. for external stack walkers for Python
- Eventually, these should enable creating a bpftrace standard library

bpftrace

Recent, ongoing, and future work

Command line options

- Large scripts/tools intended for frequent reuse usually require configuration options.
- We introduce a new `opts` builtin which allows the script to define its options.

```
opts = [{
  type=int,
  short="i",
  long="interval",
  desc="Interval in seconds"
}]

interval:s:opts.interval {
  print(...)
}
```

Summary

- **bpftrace** is a powerful tool for tracing Linux systems which allows to leverage BPF without the need to understand its technicalities.
- There is a lot of active development to overcome the existing limitations and support creating and maintaining complex tools.

Summary

- **bpftrace** is a powerful tool for tracing Linux systems which allows to leverage BPF without the need to understand its technicalities.
- There is a lot of active development to overcome the existing limitations and support creating and maintaining complex tools.

- Visit our new website! *https://bpftrace.org*
- Do you have questions? Ideas for features? Did you find bugs?
  → reach out via *https://github.com/bpftrace/bpftrace/*

Summary

- **bpftrace** is a powerful tool for tracing Linux systems which allows to leverage BPF without the need to understand its technicalities.
- There is a lot of active development to overcome the existing limitations and support creating and maintaining complex tools.

- Visit our new website! *https://bpftrace.org*
- Do you have questions? Ideas for features? Did you find bugs?
  → reach out via *https://github.com/bpftrace/bpftrace/*

Thank you for the attention!

bpftrace