Visual eBPF

Live Programming
Observability on Linux

@nbaksalyar • Tracing Summit 2022

Adventures in tracing

- I want to dynamically observe the state of the system
- Kernel knows everything
- How to find the answers?
- Recompiling the kernel it is not an option

eBPF to the rescue

- … obviously!
- Part of the Linux kernel
- Dynamically attaching programs to trace points

Ways of using eBPF

○ Writing your own programs using libbpf

Ways of using eBPF

- Writing your own programs using libbpf
  - … is not ideal

Ways of using eBPF

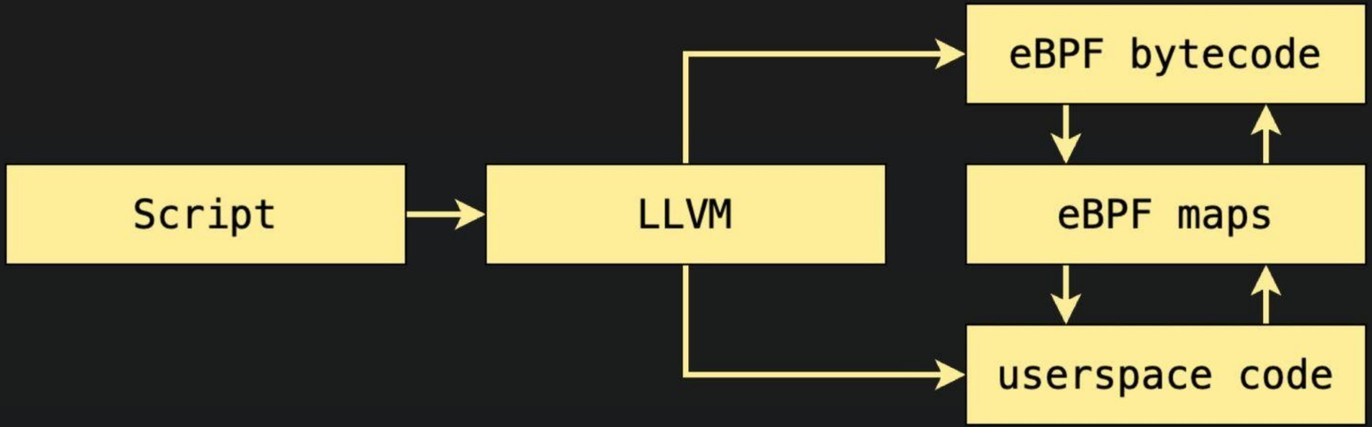
- Writing your own programs using libbpf
    - … is not ideal
- **bpftrace**: a simple scripting language

How bpftrace works

Attach to the scheduler and count the number of new processes:

```
tracepoint:sched:sched_process_fork
{
    @ = count();
}
```

How bpftrace works



Script

LLVM

eBPF bytecode

eBPF maps

userspace code

Is it possible to improve bpftrace?

- Command-line tools are…
  - Not interactive
  - Not visual enough
  - Limited by a single input method

Taking inspiration from databases

○ SQL is a domain-specific programming language

○ SQL is *declarative*

○ But SQL is for *static data* only

○ … or is it?

Streaming databases

- Apache Flink/Spark
    - the *streaming* abstraction
    - querying data in motion using CQL
- Is Linux kernel a database?
    - It can be!

Querying the kernel

Attach to the scheduler and count the number of new processes:

```
SELECT COUNT(*)

FROM "sched:sched_process_fork"

WHERE process_name = "Python"
```

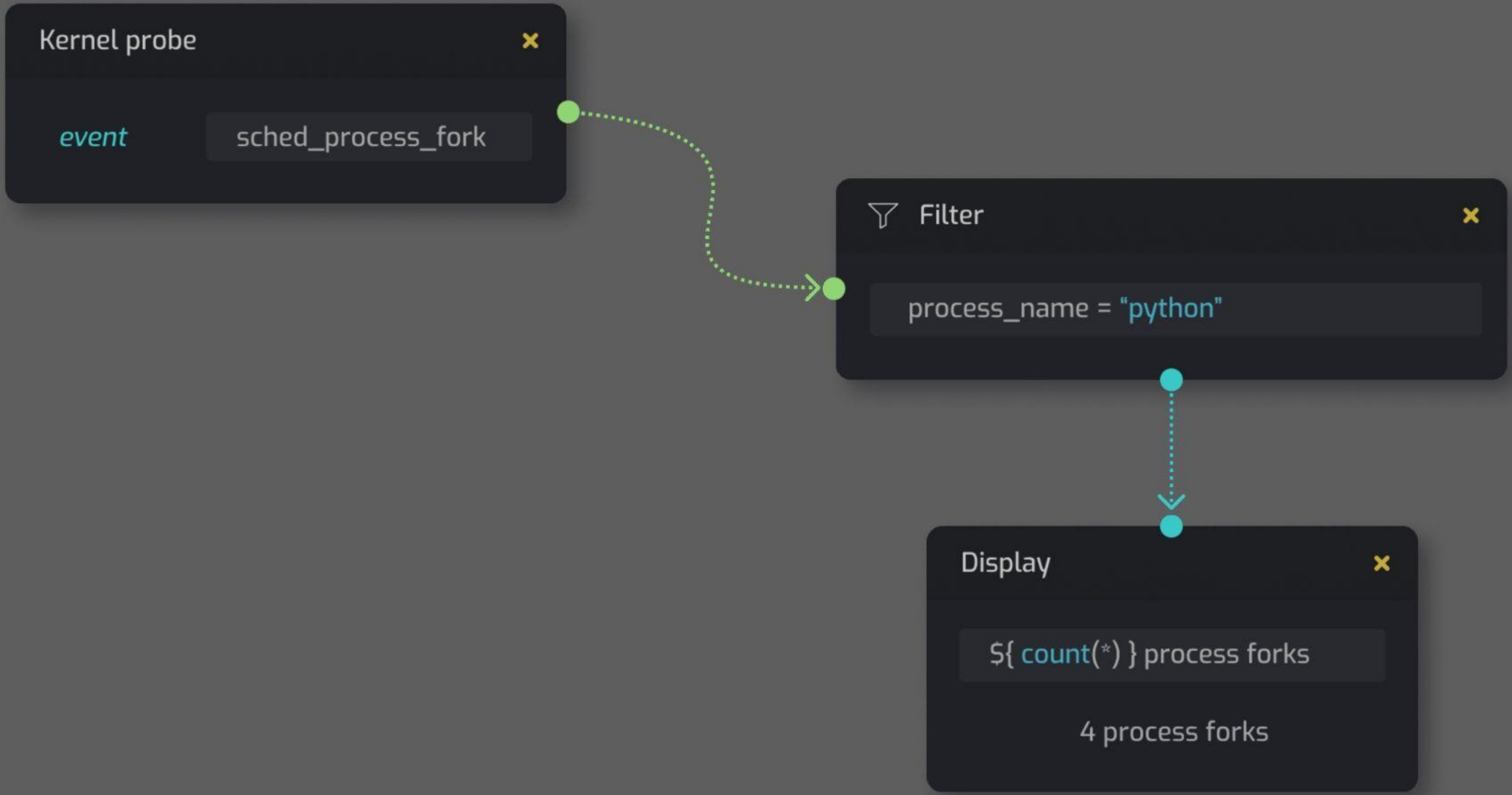
How is it different from bpftrace?

○ Declarative thinking

○ Can be translated into visual representation

○ Visual programming!

```
SELECT COUNT(*) FROM "sched_process_fork" WHERE process_name = "python"
```



Kernel probe

event sched_process_fork

Filter

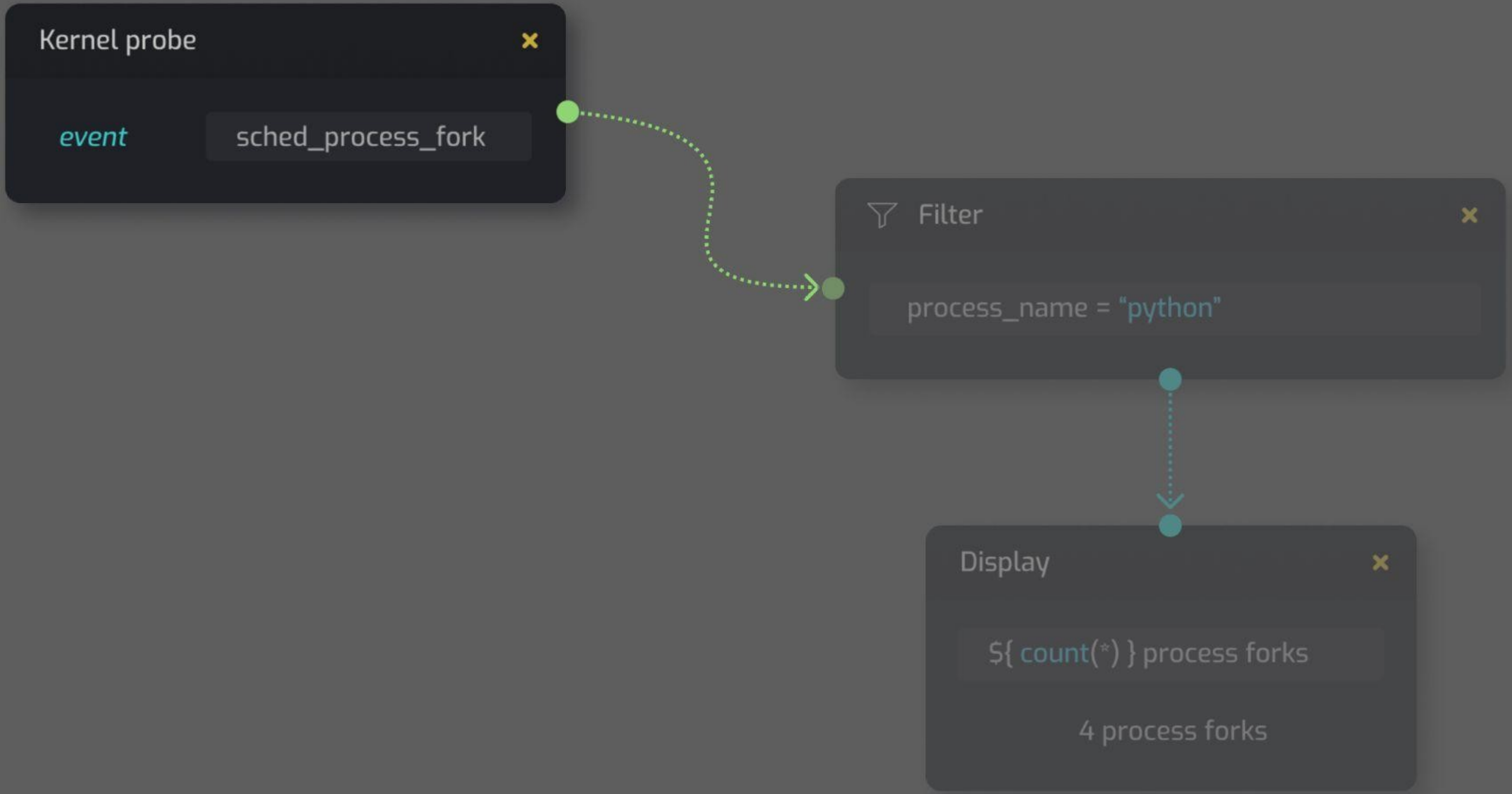
process_name = “python”

Display

${ count(*) } process forks

4 process forks

```
SELECT COUNT(*) FROM "sched_process_fork" WHERE process_name = "python"
```



Kernel probe

event sched_process_fork

Filter

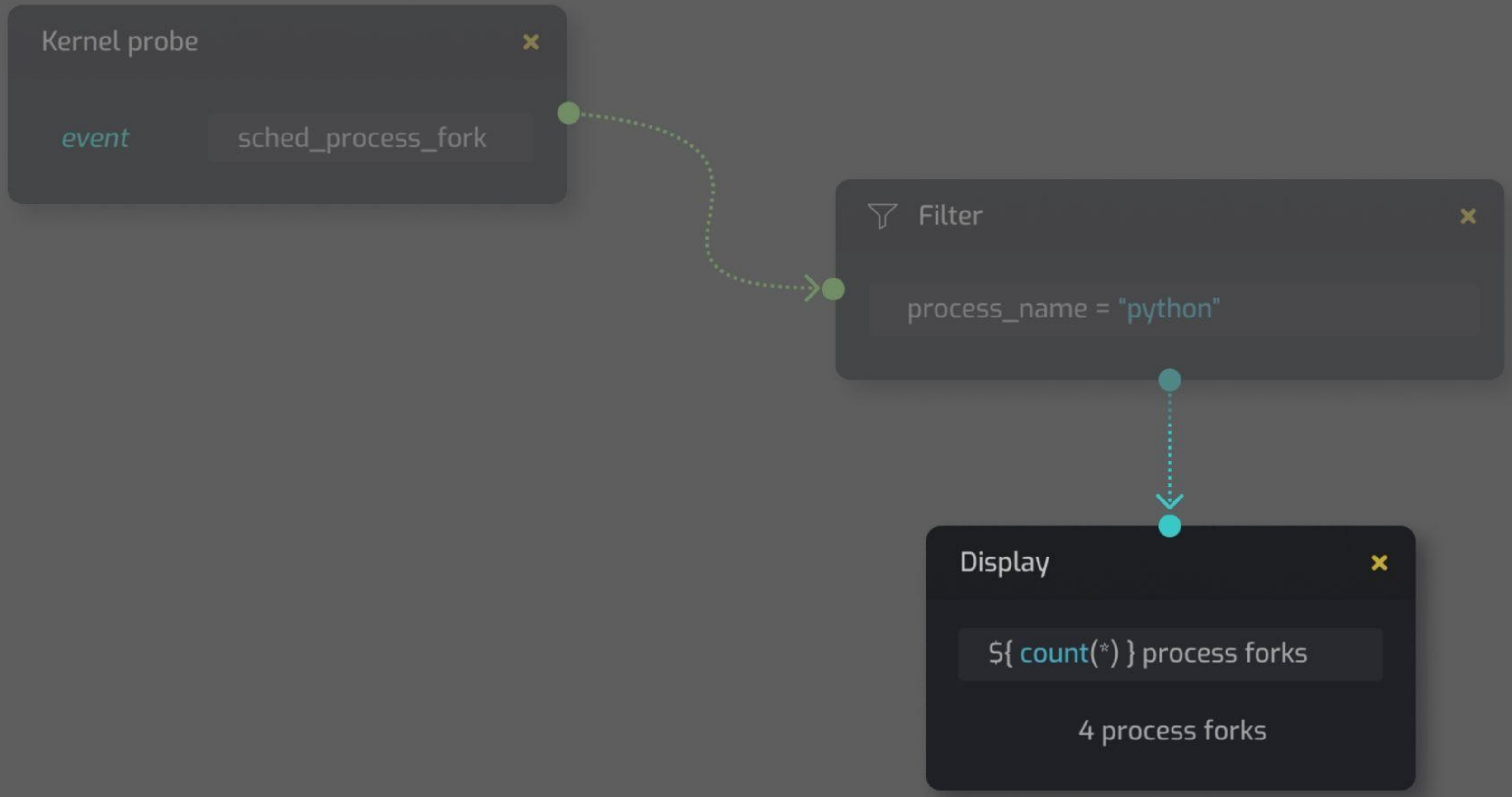
process_name = “python”

Display

${ count(*) } process forks

4 process forks

```
SELECT COUNT(*) FROM "sched_process_fork" WHERE process_name = "python"
```



Kernel probe

event sched_process_fork

Filter

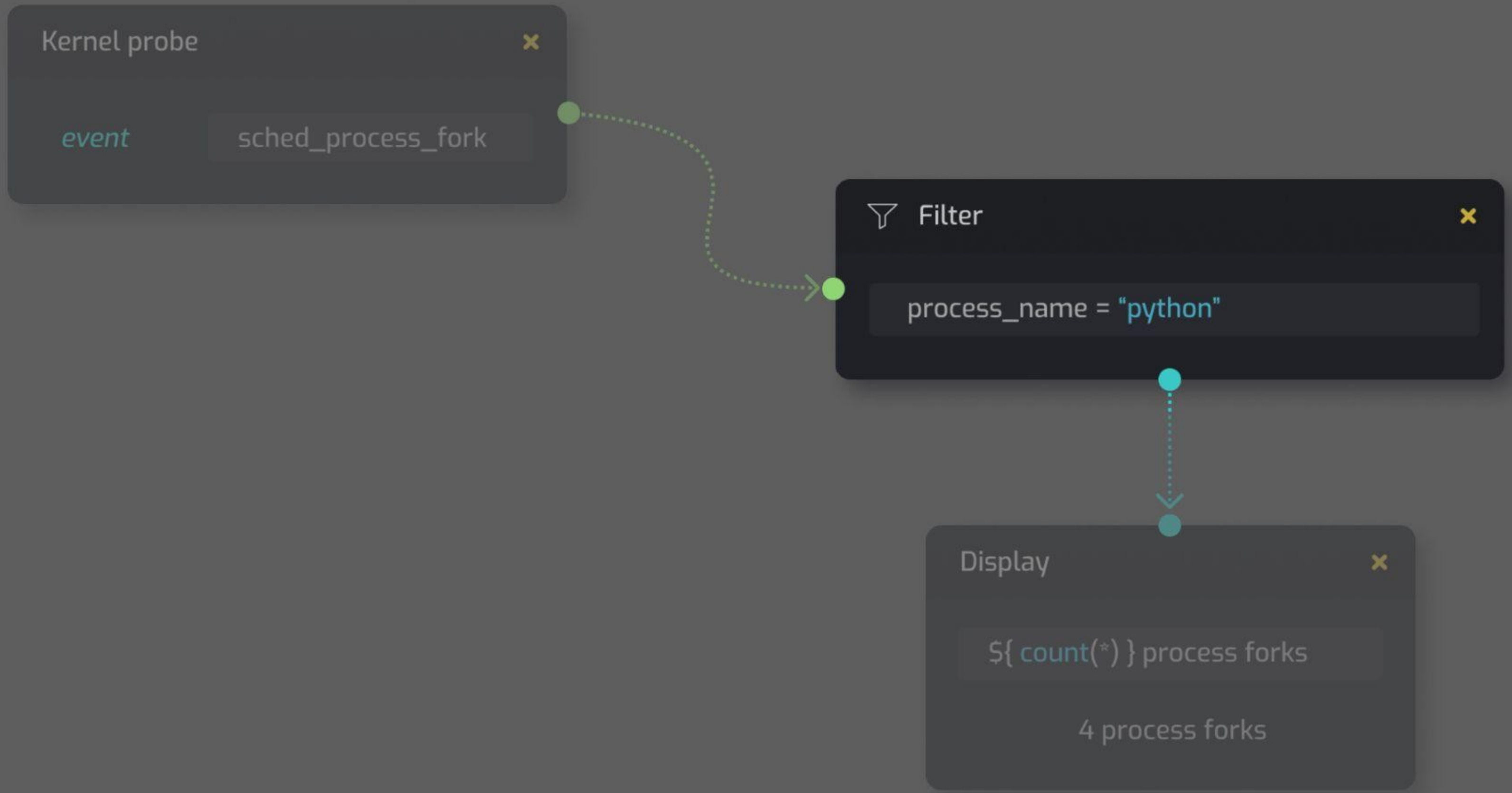
process_name = “python”

Display

${ count(*) } process forks

4 process forks

```
SELECT COUNT(*) FROM "sched_process_fork" WHERE process_name = "python"
```



Kernel probe

event sched_process_fork

Filter

process_name = “python”

Display

${ count(*) } process forks

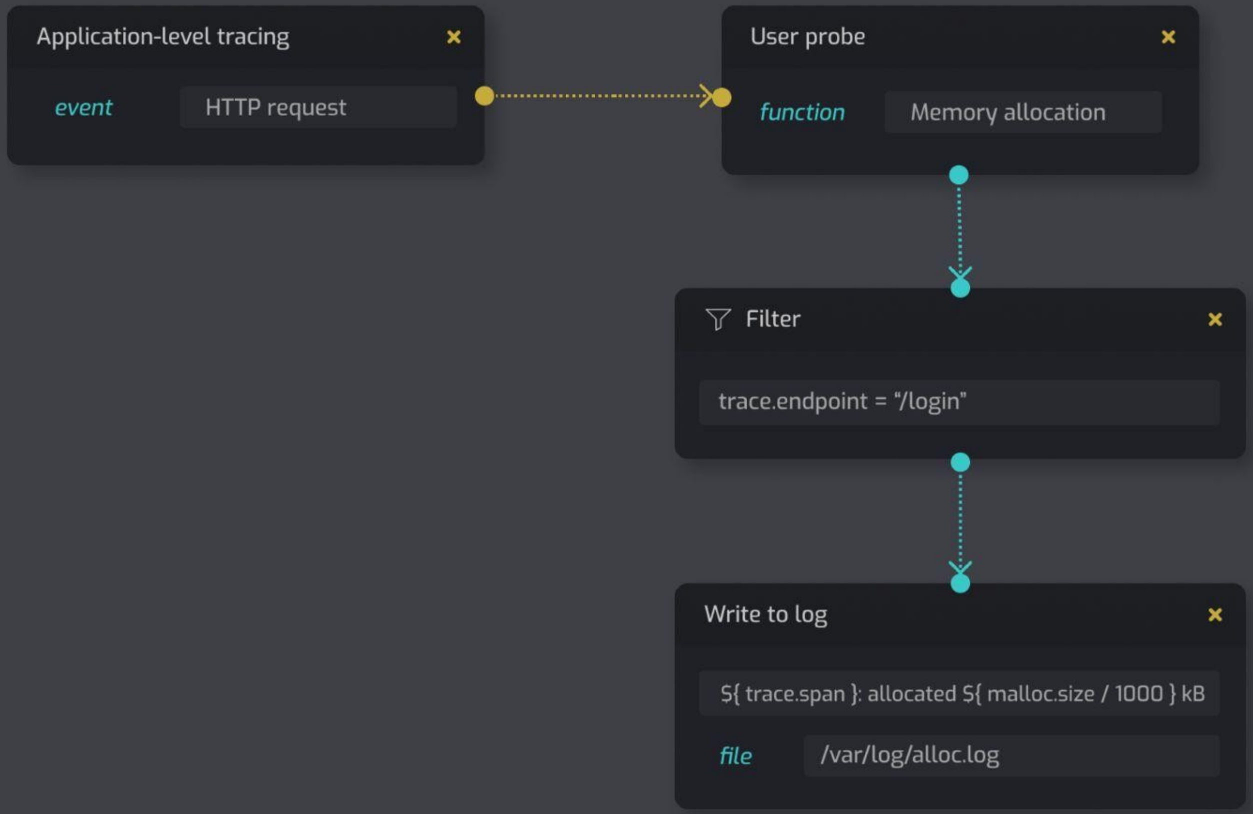
4 process forks

Pros and Cons of visual programming

- Pros
  - Gives immediate live feedback
  - More intuitive user experience
- Cons
  - Text is *too* ubiquitous and universal
  - Complex programs can get messy

More advantages

- More ways of visualising data
- Easier to *compose* programs
    - Combining multiple inputs
    - Streaming to multiple outputs



Application-level tracing

event HTTP request

User probe

function Memory allocation

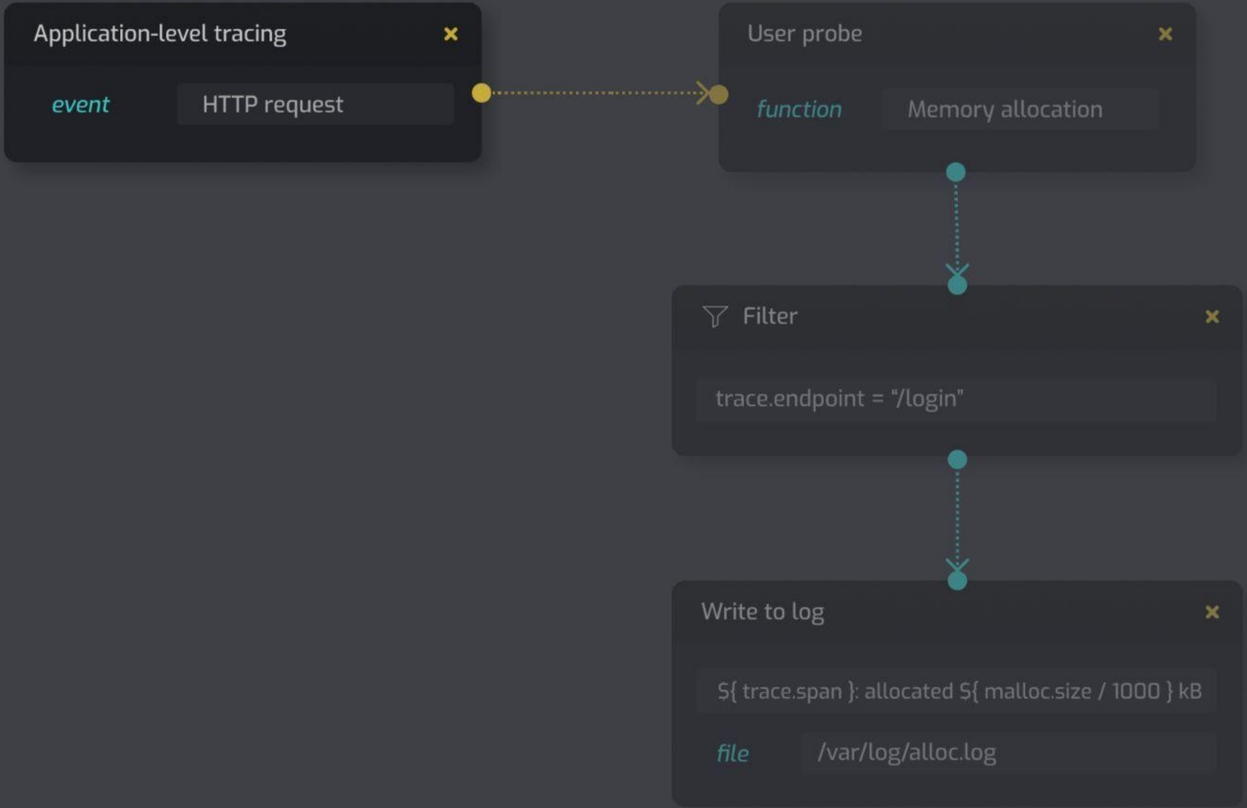
Filter

trace.endpoint = “/login”

Write to log

${ trace.span }: allocated ${ malloc.size / 1000 } kB

file /var/log/alloc.log



Application-level tracing

event: HTTP request

User probe

function: Memory allocation

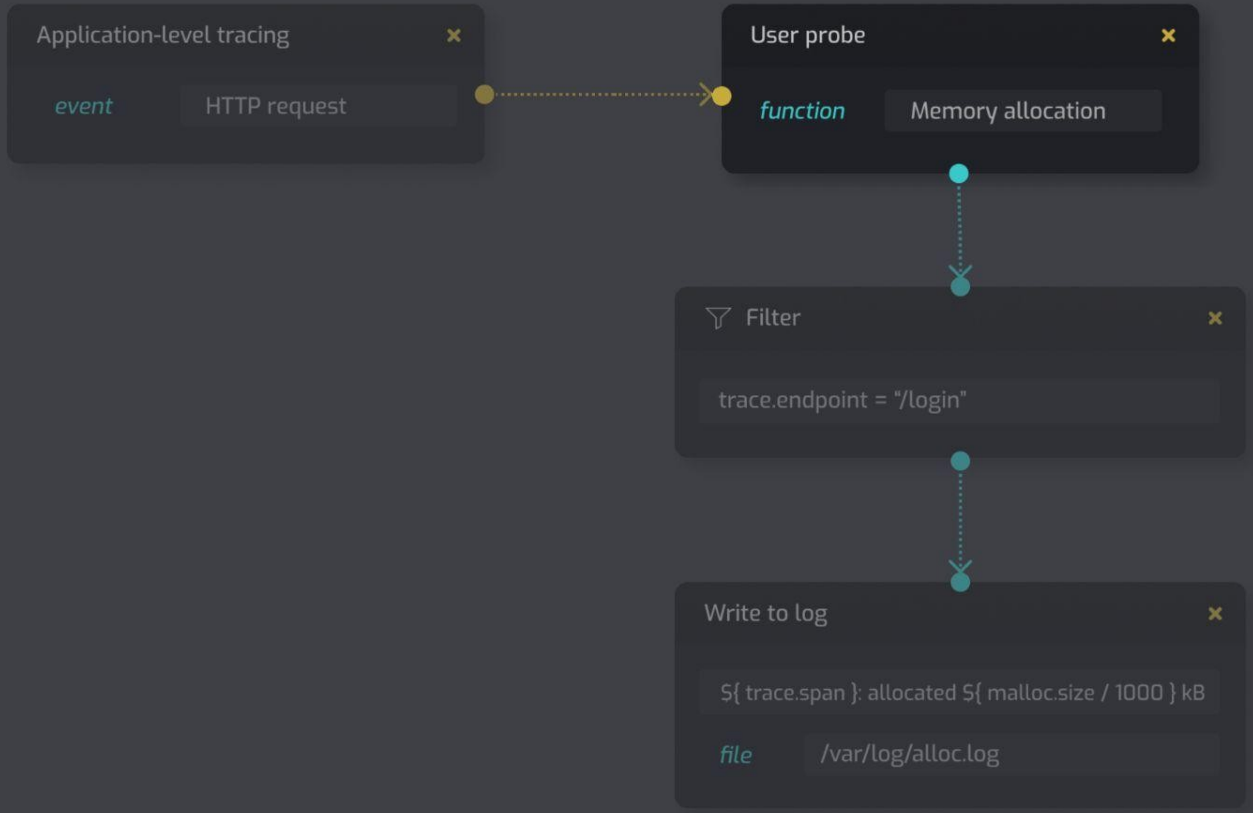
Filter

trace.endpoint = "/login"

Write to log

${ trace.span }: allocated ${ malloc.size / 1000 } kB

file: /var/log/alloc.log



Application-level tracing

event HTTP request

User probe

function Memory allocation

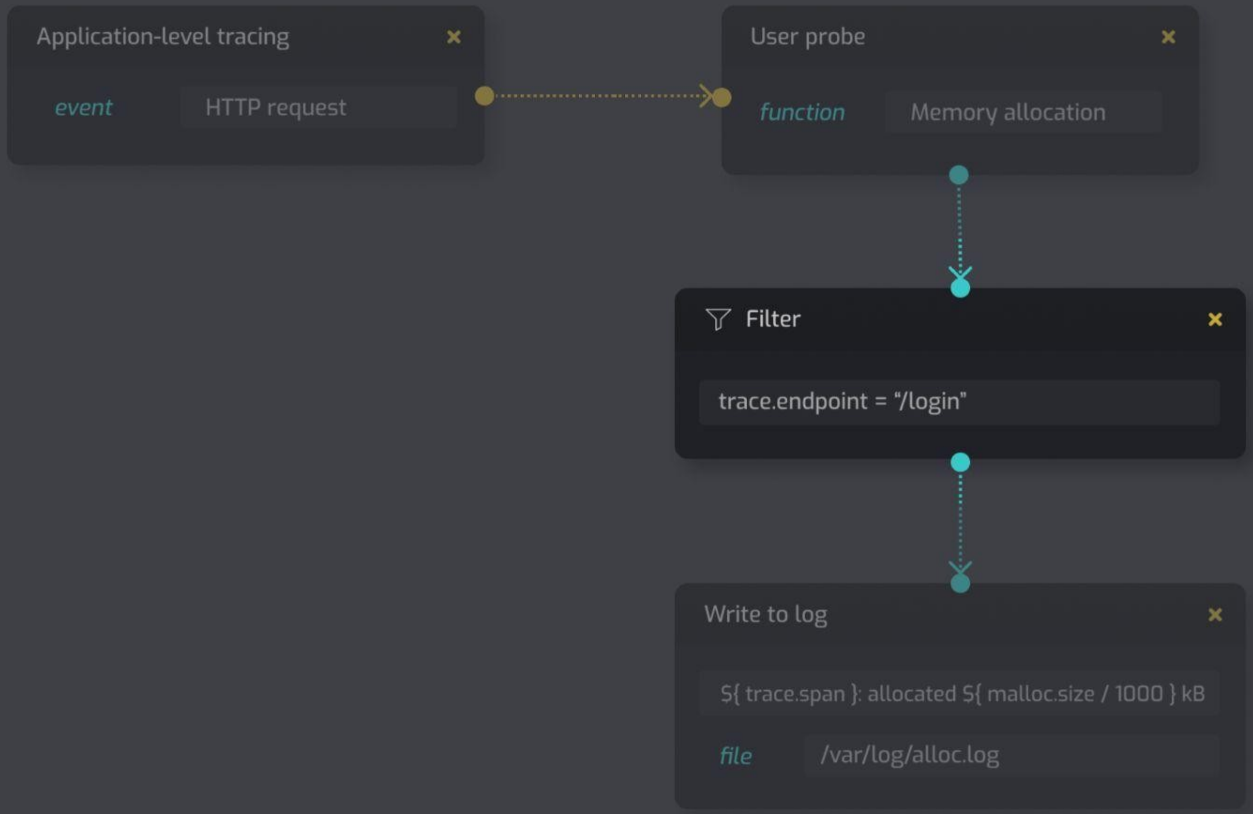
Filter

trace.endpoint = "/login"

Write to log

${ trace.span }: allocated ${ malloc.size / 1000 } kB

file /var/log/alloc.log



Application-level tracing

event: HTTP request

User probe

function: Memory allocation

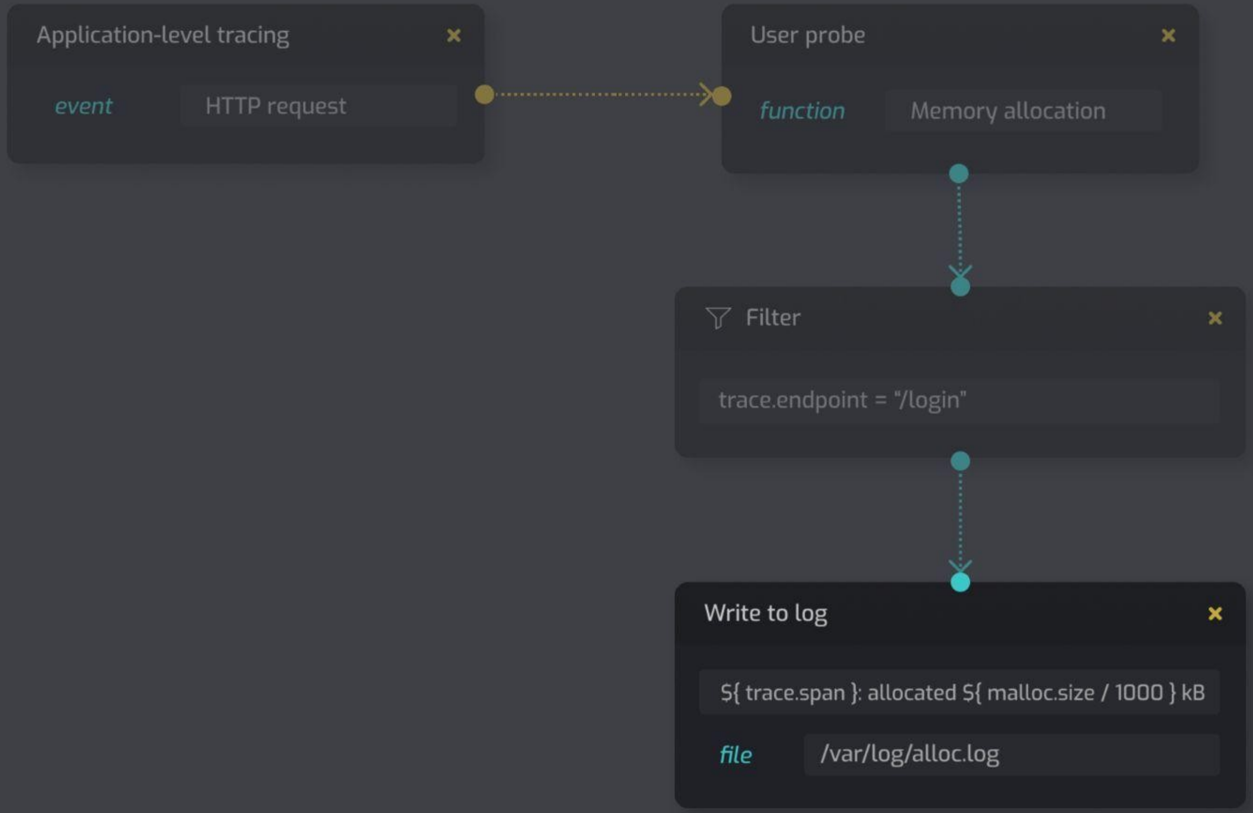
Filter

trace.endpoint = “/login”

Write to log

```
${ trace.span }: allocated ${ malloc.size / 1000 } kB
```

file: /var/log/alloc.log



Application-level tracing

event HTTP request

User probe

function Memory allocation

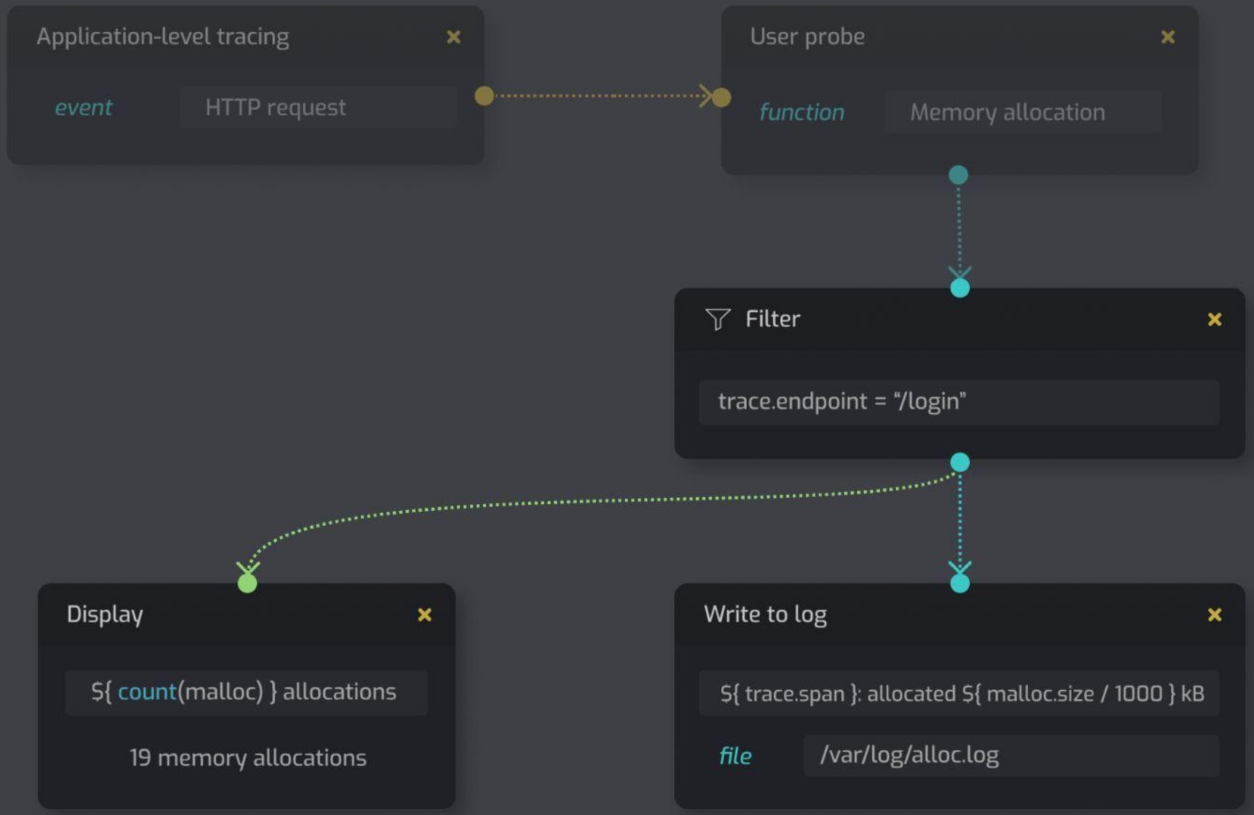
Filter

trace.endpoint = “/login”

Write to log

${ trace.span }: allocated ${ malloc.size / 1000 } kB

file /var/log/alloc.log



Application-level tracing

event HTTP request

User probe

function Memory allocation

Filter

trace.endpoint = "/login"

Display

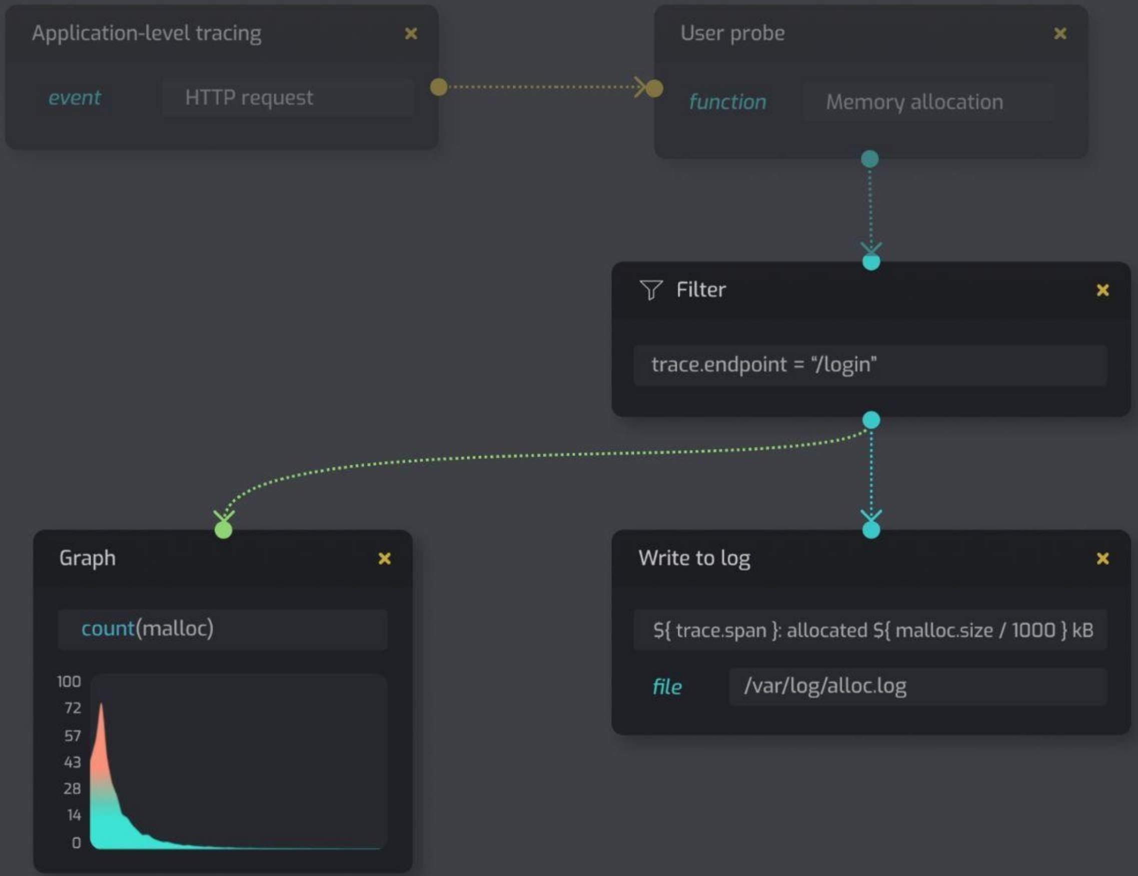
${ count(malloc) } allocations

19 memory allocations

Write to log

${ trace.span }: allocated ${ malloc.size / 1000 } kB

file /var/log/alloc.log



Application-level tracing

event HTTP request

User probe

function Memory allocation

Filter

trace.endpoint = “/login”

Graph

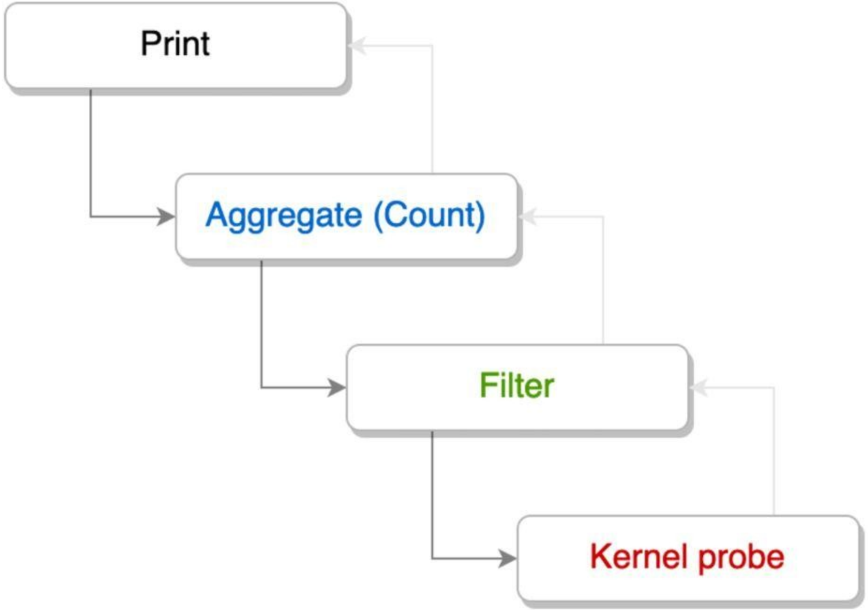
count(malloc)

Write to log

${ trace.span }: allocated ${ malloc.size / 1000 } kB

file /var/log/alloc.log

How does it work?



```
@bpf_map
events_count = 0

@kernel_probe(“syscall”)
def event_handler(arg):
  if (arg == 1):
    events_count += 1
```

How does it work in userspace

- Data is exchanged through ring buffers (Thanks Andrii Nakryiko!)
- Sent directly to a web browser using WebSockets
- Lots of visualisation options

More can be done!

- LLVM IR can be compiled into WebAssembly
- IDE-like capabilities
  - Code completion (yay BTF and CO-RE!)
  - Snippets/patterns
- Optimisations techniques borrowed from DBs

More can be done!

- LLVM IR can be compiled into WebAssembly
- IDE-like capabilities
  - Code completion (yay BTF and CO-RE!)
  - Snippets/patterns
- Optimisations techniques borrowed from DBs

Conclusion

- Linux is a kind of a database!
- Bringing visual programming into tracing world
- Visualisation can improve developers experience
- Open source on Github:

  https://github.com/nbaksalyar/metalens