
# So you want to write memory with eBPF?

*Nikola Grcevski, Grafana Labs*
*Mike Dame, Odigos*

# Who we are

Cilium + eBPF Day NORTH AMERICA

**November 12, 2024**
**Salt Lake City**

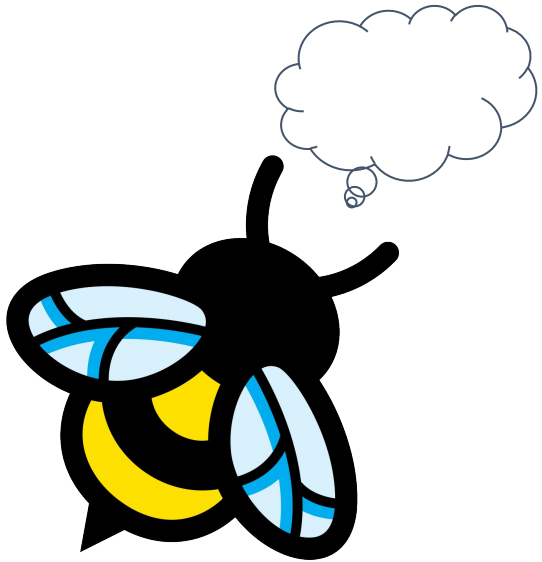
**Nikola Grcevski**
Software Engineer
*Grafana Labs*

**Mike Dame**
Software Engineer
*Odigos*

# In this talk…



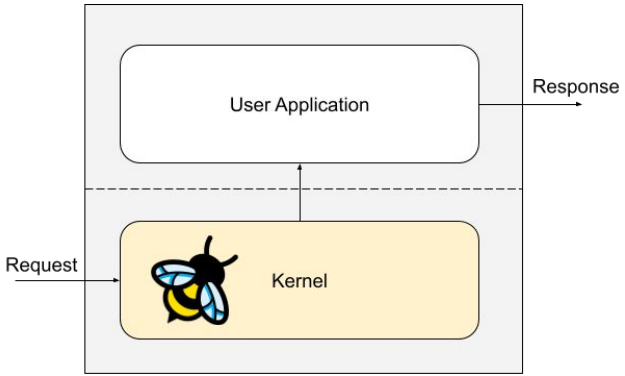
- **Why** write user space memory with eBPF?
- **Current state** of options
- **Alternatives** to writing memory with eBPF
- **Summary** and looking ahead



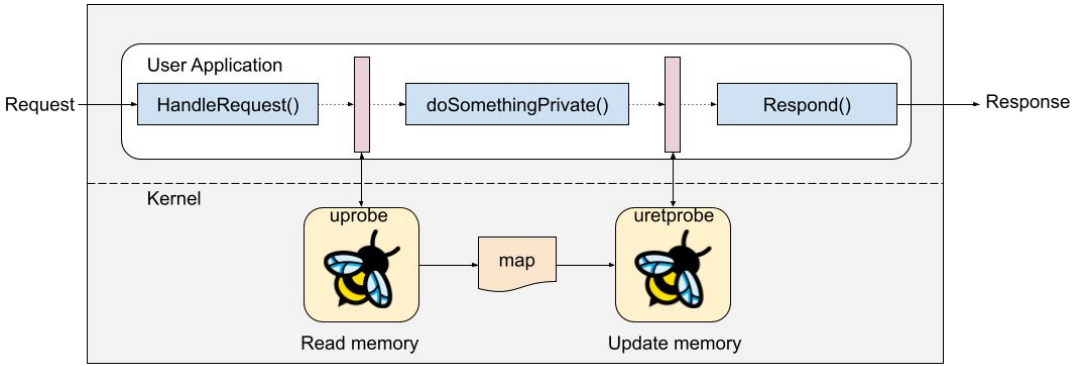
# So you want to write memory with eBPF?

- Traditional eBPF programs don't write user-space memory
  - Writing usually happens before/after user space, eg networking

- Security, networking, and observability can mostly be handled this way




Normal

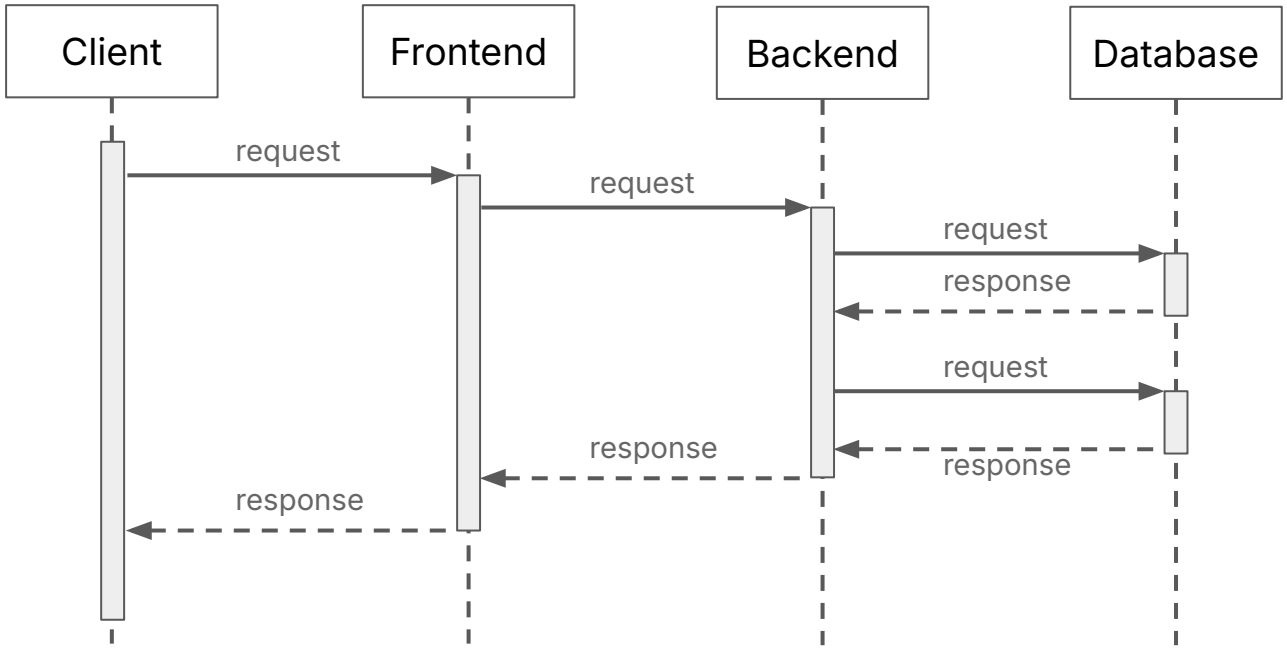



Memory write

# Why write memory with eBPF?



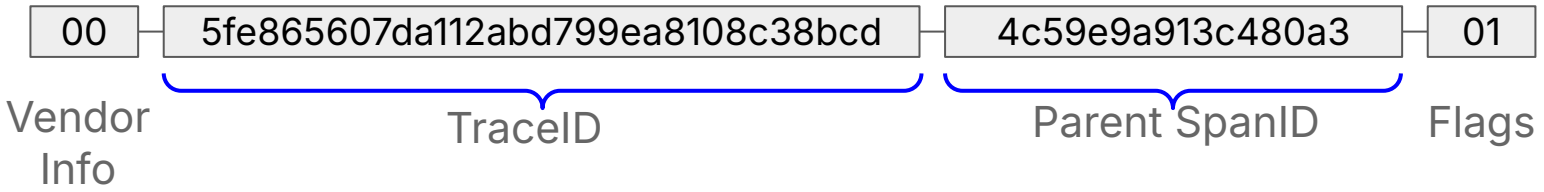
- OpenTelemetry Tracing: **outgoing request context propagation**
  - Other observability signals can be done read-only (logs, metrics)
  - Tracing requires context to be sent to downstream services



# How is context propagated between services?



- Each new request gets unique 16 hex character **SpanID**
- W3C defines a request header field called “traceparent”




- The **TraceID** is common for all spans of one trace
- This traceparent value is propagated through outgoing header calls

# Go pseudo code that does this

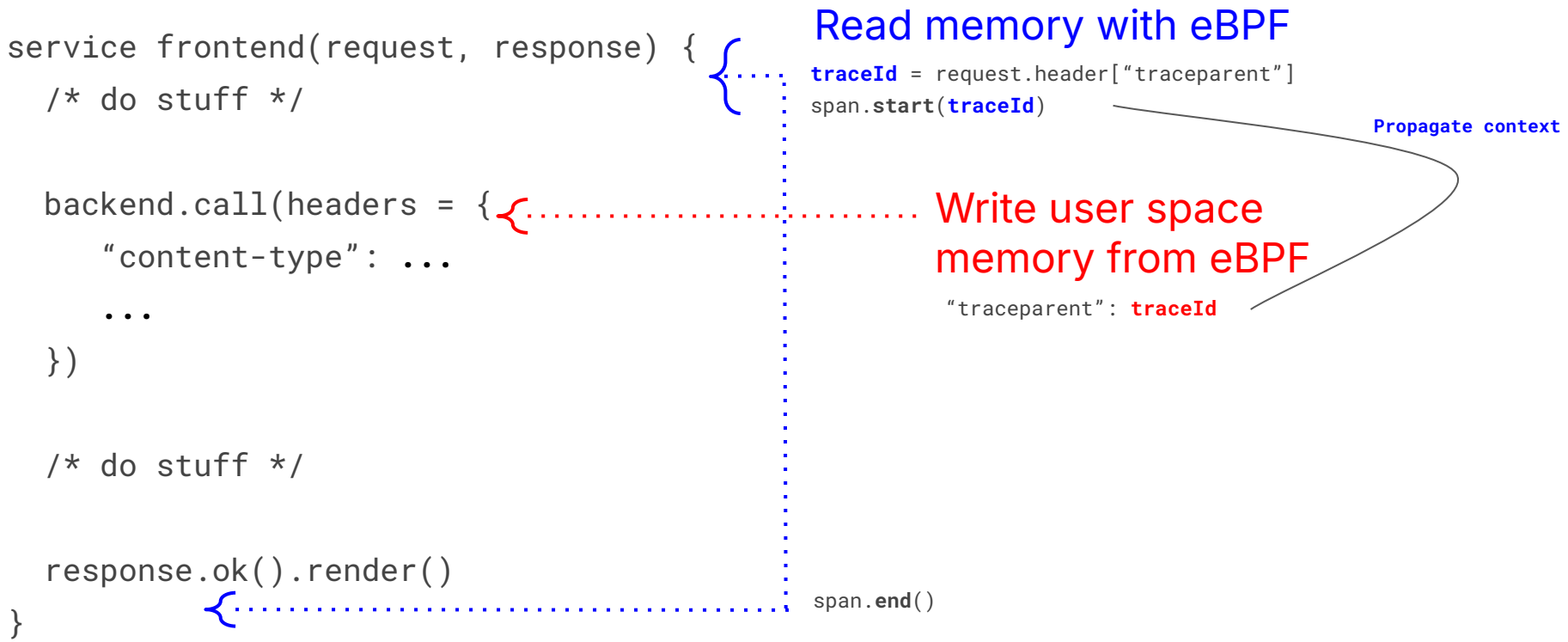


```
service frontend(request, response) {
  traceId = request.header[“traceparent”]
  span.start(traceId)

  /* do stuff */

  backend.call(headers = {
     “traceparent”: traceId
  })

  /* do stuff */

  response.ok().render()
  span.end()
}
```

**Propagate context:**
**- same Trace ID**
**- new Span ID**

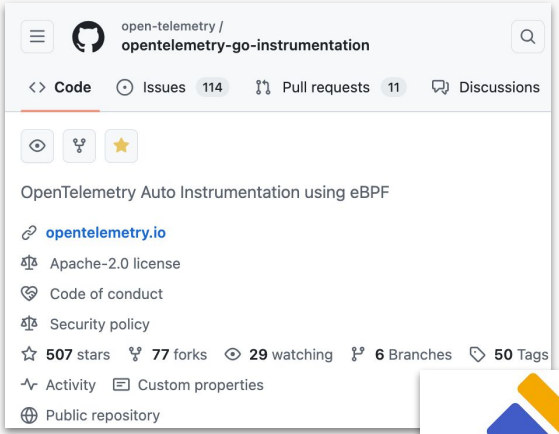
Can be injected by an instrumentation SDK or agent

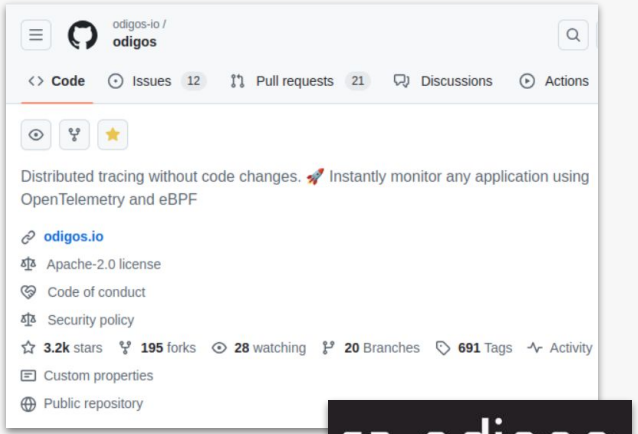
# We'd like do the same thing automatically

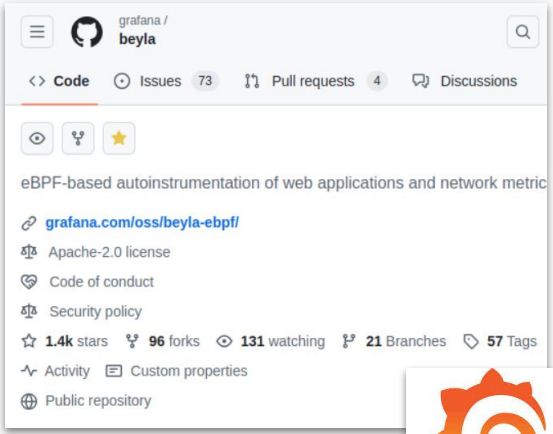


# Projects which write user memory with eBPF

Cilium + eBPF Day NORTH AMERICA




Projects based on this approach to eBPF Auto-Instrumentation:

**5,000+ stars**

**300+ forks**

**100's of contributors**

# How do you write user memory with eBPF?



- **bpf_probe_write_user()**
  - Added in `linux@96ae522` *(July 2016)*
  - Meant to "debug, divert, and manipulate execution of semi-cooperative processes."
  - "Meant for experiments" with a "high risk of crashing the system"
  - Requires CAP_SYS_ADMIN

- Most other projects on GitHub (besides **OTel-Go-Auto, Odigos** and **Beyla**) use this helper for ***malicious purposes*** (with root access)

- Locked down in `linux@51e1bb9` *(August 2021)* under LSM LOCKDOWN_BPF_WRITE_USER (Kernel 5.10+)
  - *"These days we have better mechanisms in BPF for achieving the same (e.g. for load-balancers), but without having to write to userspace memory."*

**So what other options are available?**

# **Proposal:** bpf_probe_write_user_registered()



- Proposed in April 2024 by Marco Elver (Google) to address memory safety issues with bpf_probe_write_user()
  - **Use case:** User space threads want to check for frequent scheduler events along a "very hot" code path to make optimal decisions. eBPF provides performance and deployment trade-offs for system-based heuristics. **Another use-case for bpf_probe_write_user**

- Enforce writes to specific, expected regions

- User space application explicitly registers writable memory regions for eBPF programs

# **Denied:** bpf_probe_write_user_registered()

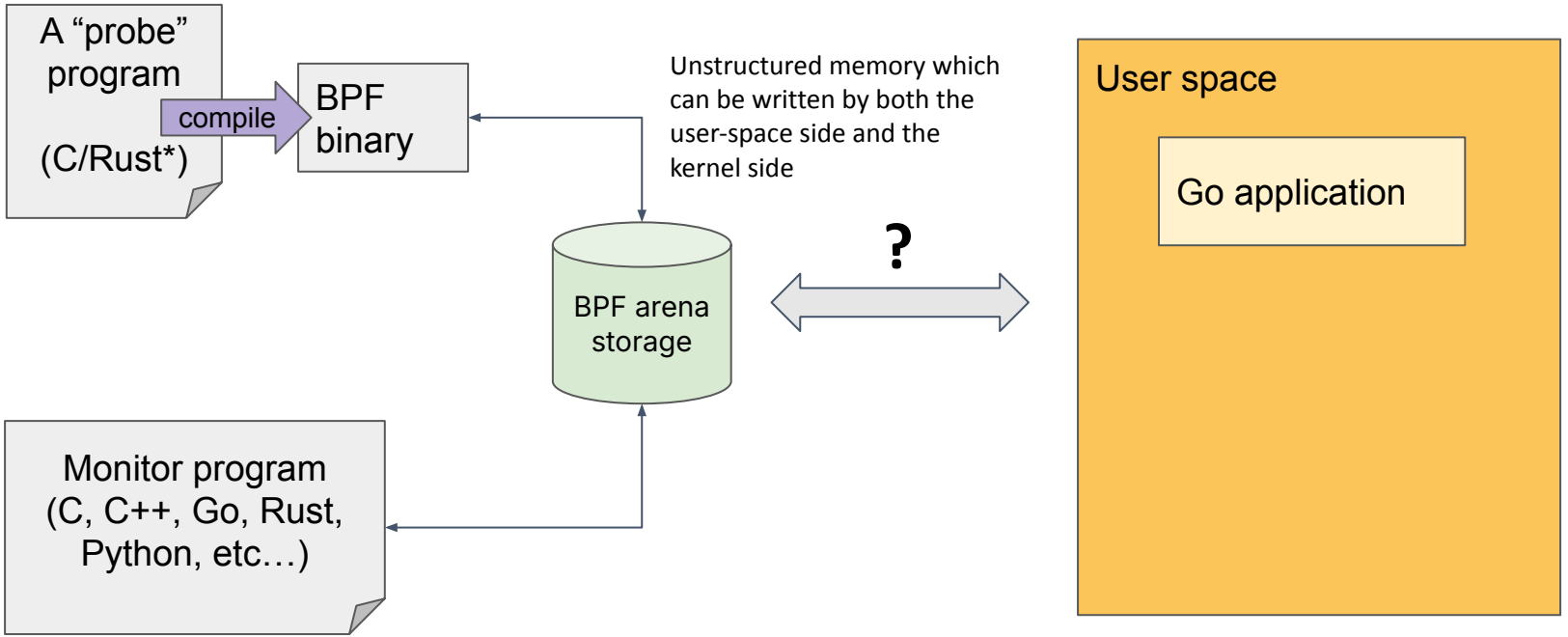


- Userspace and kernel have other mechanisms to communicate
- No longer accepting new bpf helpers
- BPF Arena likely better fit
- Better approach "on todo list" to let user code register memory to eBPF map, kernel pins pages, and BPF can read+write directly to user memory

# Alternative: BPF Arena



- Added in `linux@3174603` *(March 2024),* supports "millions of tasks"



# Approach: BPF Arena



- Allocate Go runtime memory as a BPF Arena?

```
func (h Header) writeSubset(w io.Writer,...
  /* iterate and write all headers */
  ...
}
```

uprobe

Write user space memory from eBPF

`"traceparent": traceparent`

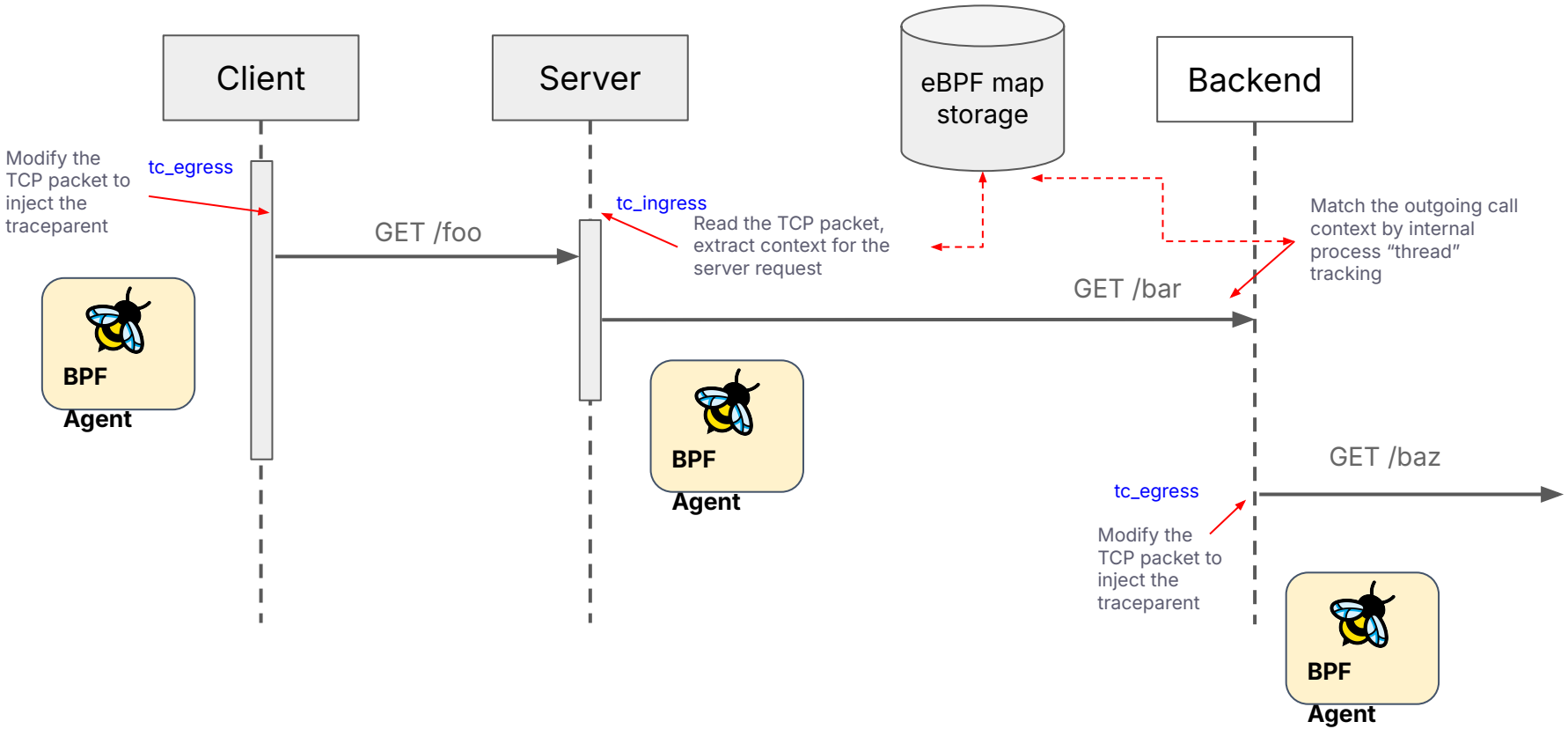
- Access to the arena in user-space is by the loader program
  - Needs elevated permissions, not just any userspace application
- Arenas (BPF maps) are zero initialized
  - We can't just map an arena to the whole of the Go heap
- Our use-cases of ***bpf_probe_write_user*** can't be replaced

# Do you need to write user memory?



- Is your problem a memory problem?
- Trace context is stored in request headers
  - Is there another approach, for example at network level
- We can write memory with ***bpf_skb_store_bytes*** in Linux Traffic Control (TC) and eXpress Data Path (XDP) BPF programs

# Alternative: L4 context propagation




*Beyla has a prototype code for this

# Problem: L4 context propagation



- Requires that the eBPF agent runs on both sides
  - The TCP packet can only be decoded by custom eBPF code
  - Unable to pass the trace information to OpenTelemetry SDK instrumented application

- Doesn’t work with L7 proxies (load balancers)
  - Unless those proxies are instrumented too

# Alternative: L7 context propagation



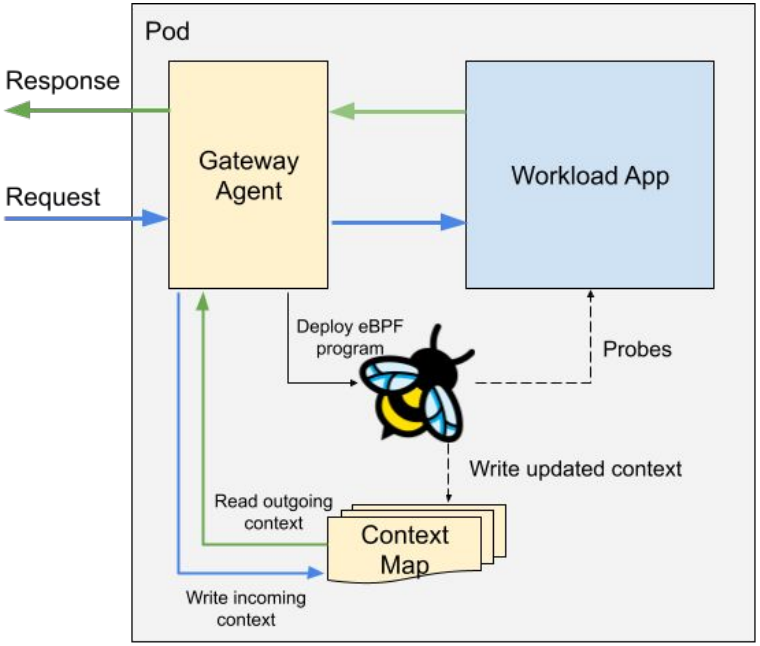
- Incoming request headers are parsed for parent spans
- In-process execution path is tracked
- Extend the packets in a BPF_PROG_TYPE_SK_MSG
- Use Linux Traffic Control (TC) to write the ‘traceparent’ header on egress

- Problem: Encryption prevents reads/writes at L7

# Alternative: Gateway Agent



- Agent loads eBPF probes and acts as a request/response gateway to update headers at application layer
- eBPF only needs to probe symbols and write to a map and agent only needs CAP_SYS_BPF
- Requires networking updates
- Adds 2 hops per request per app



# Do you really need to write memory with BPF?

# Alternative: library interpositioning/patching



- Use library interpositioning, e.g. LD_PRELOAD
  - Example: Hijack calls to **libc** or **libssl**
  - Substitute the single library call to multiple calls to write additional memory

```
SSL_write("GET /hello\r\nUser-agent: curl…")
```

```
SSL_write_hijacked("GET /hello…") {
  SSL_write("GET /hello\r\n")
  SSL_write("Traceparent: 00-123…\r\n")
  SSL_write("User-agent: curl…")
}
```

# Problem: library interpositioning/patching



- Attaching to existing running processes is not straightforward
  - The program must be stopped with ptrace
- Clean-up is not easy, requires code patching and stopping the program
- Statically linked code requires more involved patching
  - But there are libraries that do this, e.g. Frida Core https://github.com/frida/frida-gum

# Alternative: bpftime



- https://github.com/eunomia-bpf/bpftime
  - This project uses the previous approach with patching/interpositioning
  - Has implementation of the BPF api, BPF programs can be easily ported to userspace applications
- This actually works fine, if it’s an acceptable solution for the end users not to use the kernel BPF runtime
- Works really well for testing use-cases

# So… you want to write memory with eBPF?

You can do it, *but*:

→ Current best options in eBPF lack strong upstream support.

→ Alternatives have tradeoffs in security and UX.

**It's super cool that eBPF makes this possible today!** *But…*

*"The only way to really be cool is to follow the rules."*

# Looking ahead

- **Are there valid use cases for writing user memory with eBPF?**
  - We think so…
  - If not enough now, could there be someday as eBPF grows?
- **How could shared kernel/userspace memory access be implemented safely and reliably?**
  - Likely need some coordination between eBPF and user space to prevent out-of-range writes and bad actors
- **How can we build stronger native support?**

# Thank you!

https://github.com/open-telemetry/opentelemetry-go-instrumentation

**#otel-go-instrumentation** on **slack.cncf.io**

https://github.com/odigos-io/odigos/

**#odigos** on **odigos.slack.com**

https://github.com/grafana/beyla

**#beyla** on **slack.grafana.com**

# Alternative: L4 context propagation



- Incoming request headers are parsed for parent spans
- In-process execution path is tracked
- Use Linux Traffic Control (TC) to write memory
- Outgoing calls embed the ‘traceparent’ in the IP/TCP packet payload

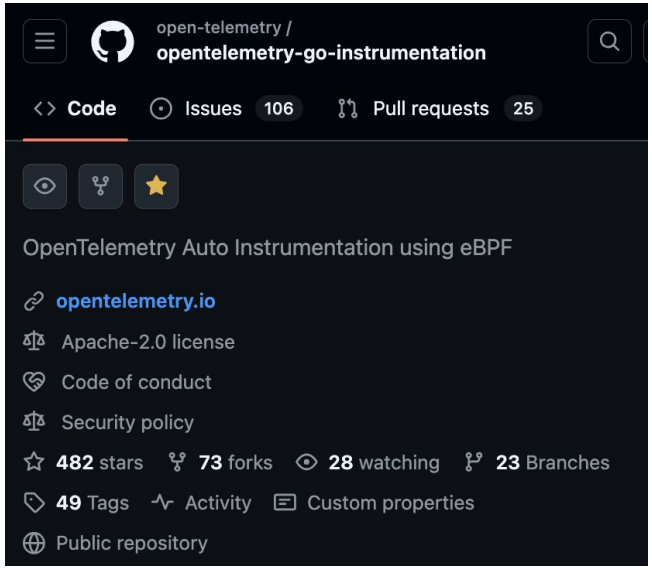
# OpenTelemetry Go auto-instrumentation

Cilium + eBPF Day NORTH AMERICA

https://github.com/open-telemetry/opentelemetry-go-instrumentation

- **Goal:** Automatic, no-code distributed tracing of common (and custom) userspace application libraries written in Go

- **Requirements:**
  - Comply with OpenTelemetry Semantic Conventions
  - Provide stable instrumentation and API
  - Minimal (->zero) effort from users



# Other potential use cases



- Security/Compliance (Offensive)
  - Prevent sensitive values from being stored in memory
  - Block encrypted exploits
- Providing kernel patches to kernels which cannot be patched
  - IoT devices
- Leverage stability of userspace libraries vs kernel ABI

# Alternative: ptrace



- Use ptrace to attach shared memory space to target program
  - Do most of the work with eBPF - track the requests and internal correlation as usual
  - Use ptrace interrupt at the point when we need to write memory

# Problem: ptrace



- Interrupts are very expensive to process
- Application is effectively serialized on the handling of the interrupt

# Problem: BPF Arena



- Access to the arena in user-space is by the loader program
  - Needs elevated permissions, not any userspace application can do this
- Arenas (BPF maps) are zero initialized
  - We can’t just map an arena to the whole of the Go heap
- Not all of our use-cases of ***bpf_probe_write_user*** can be replaced

# So do you really need to write memory with BPF?