Cilium + eBPF Day

EUROPE

Curveballs - learnings from instrumenting managed runtime applications with eBPF

Cilium + eBPF Day EUROPE

19 March 2024 | Paris, France

Nikola Grcevski
Software Engineer
Grafana Labs

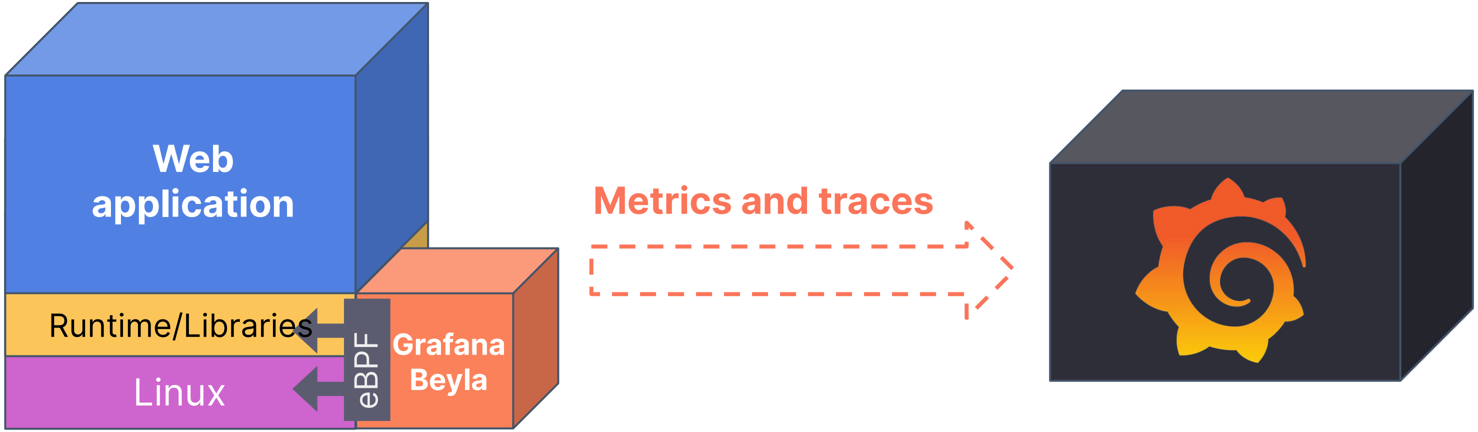
Mario Macias
Software Engineer
Grafana Labs

Contents

- Usual ways we instrument ELF binaries with eBPF
- Managed runtime language pose unique challenges in instrumentation
  - Managed memory and garbage collection
  - Threading models
  - Different linkage conventions
- Ways we can overcome some of the challenges
  - Intermediate level: Go
  - Death-march: Java

How do we instrument applications?

Auto-instrumentation with eBPF - Grafana Beyla



Instrumenting binaries

- Uprobes/Uretprobes and USDTs (user statically defined tracepoints)
  - Know when a function call starts/ends
  - Get parameters information
- USDTs are very nice, but they are typically uncommon
  - OpenJDK makes extensive use of of USDTs, but they are not built with by default 😞

Libssl3 read example

```
SEC("uprobe/libssl.so:SSL_read")
int BPF_UPROBE(uprobe_ssl_read, void *ssl, const void *buf, int num) {
   u64 id = bpf_get_current_pid_tgid();
   // stash the pointer to the buffer and num bytes
   ssl_args_t args = { .buf = buf, .num = num };
   bpf_map_update_elem(&active_ssl_read_args, &id, &args, BPF_ANY);
   return 0;
}

SEC("uretprobe/libssl.so:SSL_read")
int BPF_URETPROBE(uretprobe_ssl_read, int ret) {
   u64 id = bpf_get_current_pid_tgid();
   if (ret < 0) return 0;
   ssl_args_t *args = bpf_map_lookup_elem(&active_ssl_read_args, &id);
   if (args) handle_ssl_buf(id, args, ret);
   bpf_map_delete_elem(&active_ssl_read_args, &id);
   return 0;
}
```

Arguments are available on function enter. We must preserve them, because we only get the function return value on exit.

At this point we've read the SSL buffer, we can do something with it. We fetch the saved function arguments to get ***buf** and ***num***.

Assumptions

Cilium + eBPF Day EUROPE

```
SEC("uprobe/libssl.so:SSL_read")
int BPF_UPROBE(uprobe_ssl_read, void *ssl, const void *buf, int num) {
    u64 id = bpf_get_current_pid_tgid();
    // stash the pointer to the buffer and num bytes
    ssl_args_t args = { .buf = buf, .num = num };
    bpf_map_update_elem(&active_ssl_read_args, &id, &args, BPF_ANY);
    return 0;
}

SEC("uretprobe/libssl.so:SSL_read")
int BPF_URETPROBE(uretprobe_ssl_read, int ret) {
    u64 id = bpf_get_current_pid_tgid();
    if (ret < 0) return 0;
    ssl_args_t *args = bpf_map_lookup_elem(&active_ssl_read_args, &id);
    if (args) handle_ssl_buf(id, args, ret);
    bpf_map_delete_elem(&active_ssl_read_args, &id);
    return 0;
}
```

We use the PID:TID pair as a map key, we assume the application doesn't overlay virtual threads on top of system threads.

We assume the address of ***buf** doesn't change.

These are valid assumptions for ***libssl***, it's an unmanaged library written in C.

Instrumenting binaries

- Uprobes and Uretprobes work almost always
    - Special care needs to be taken to ensure function arguments and memory offsets haven’t changed
    - No “*Compile Once-Run Everywhere* (CO-RE)” for uprobes
    - Binaries without symbols are hard to deal with

Changing Offsets

foolib.h v1.3.1

```
struct flow_metrics {
   u32 packets;
   u64 bytes;
   u8  errno;
}
```

Offset
Offset: 0
Offset: 4
Offset: 12

foolib.h v1.4.0

```
struct flow_metrics {
   u32 packets;
   u64 bytes;
   u16 flags;
   u8  errno;
}
```

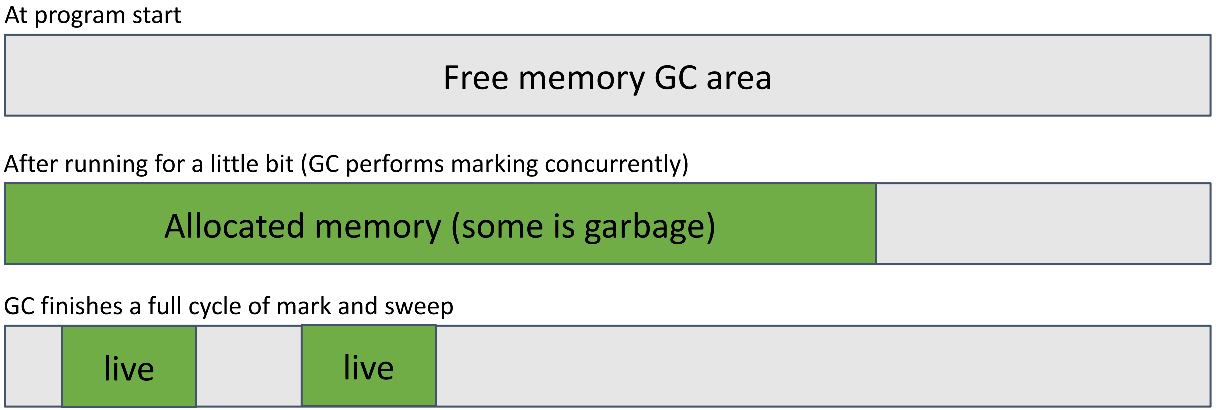
Offset
Offset: 0
Offset: 4
Offset: 12
Offset: 14

Managed runtimes

- Garbage Collection
  - There are many different kinds of garbage collectors
  - We mostly care about what they do with our pointer references
- Managed stacks
  - Can stacks grow, shrink or move?
- Managed threads
  - Does the managed runtime have virtual threads (or goroutines, green threads, etc.)?
- What linkage (or calling convention) does the program use?

Intermediate level: Go

- Garbage Collection
  - Concurrent mark and sweep, non-compacting, non-generational
  - You can’t get microsecond latency if you copy memory around
  - Heap memory references don’t move 👌



At program start

After running for a little bit (GC performs marking concurrently)

GC finishes a full cycle of mark and sweep

Go issue: managed stacks

- Stacks can grow and move (if there isn’t enough room)
- **Uretprobes often don’t work** 😒
- Solution: use **uprobes** always
  - Uretprobes can be implemented with uprobes on the return instructions
  - Requires disassembly and scanning the function code for the platform return opcode

Go issue: managed threads

- Many goroutines dynamically map to an underlying (smaller) pool of system threads
- Solution: get goroutine pointer
  - The current goroutine is always in a well defined register (Go 1.17+)
  - We can use this value as a key instead of the PID:TID pair

Go issue: linkage/calling conventions

- Go 1.16 → 1.17 changed the function calling conventions
  - Breaking changes!
- Go 1.17+ uses register calling convention, but it’s not the same as the System V ABI
- eBPF probes are sensible to linker options
- Workarounds
  - Adapt our argument register macros to match the Go linkage
    - Go version can be discovered from the binary
  - Maintain your own database of offsets
    - Homebrewed CO-RE

Go example

```
SEC("uprobe/server_handleStream")
int uprobe_server_handleStream(struct pt_regs *ctx) {
   void *goroutine_addr = GOROUTINE_PTR(ctx);
   void *stream_ptr = GO_PARAM4(ctx);
   grpc_srv_func_invocation_t invocation = {
       .start_monotime_ns = bpf_ktime_get_ns(),
       .stream = (u64)stream_ptr
   };
   bpf_map_update_elem(&ongoing_grpc_server_requests,
                       &goroutine_addr, &invocation, BPF_ANY));
   return 0;
}
SEC("uprobe/server_handleStream")
int uprobe_server_handleStream_return(struct pt_regs *ctx) {
   void *goroutine_addr = GOROUTINE_PTR(ctx);
   grpc_srv_func_invocation_t *invocation =
                        bpf_map_lookup_elem(&ongoing_grpc_server_requests, &goroutine_addr);
   if (invocation) {
      // Do something
   }
   bpf_map_delete_elem(&ongoing_grpc_server_requests, &goroutine_addr);
   return 0;
}
```

We use the goroutine address as a map key, instead of the PID:TID pair.

We save the 4th function parameter, stream, by using a Go specific macro to map the 4th argument register in the Go calling convention.

Go corner cases…

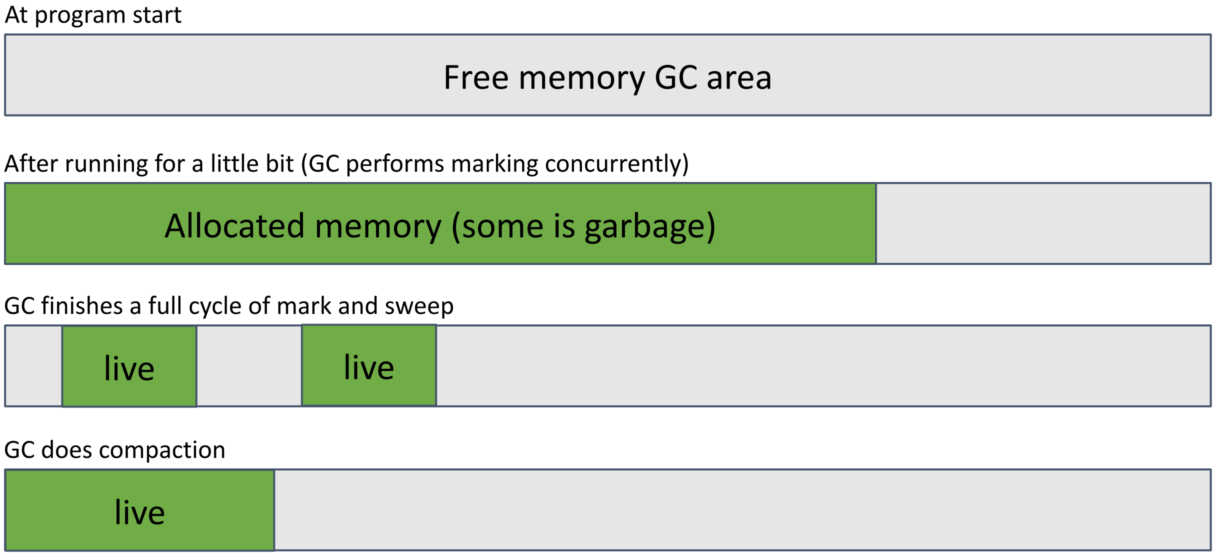
- Go heap memory references don’t move, but the stacks move 😦
- Go’s compiler performs escape analysis on pointers:
  - It looks to prove that a pointer doesn’t “escape” beyond the scope of the function call
  - If it doesn’t escape, the struct will be allocated on the stack and the pointer will be a stack pointer
- Instrumentation targets need to be inspected to ensure that the pointer values are safe to be tracked

Death-march: Java

- Different virtual machines
  - OpenJDK, GraalVM, J9, Azul Zing
  - Discussion will be limited to OpenJDK
- Garbage Collection
  - Java has number of different garbage collectors
  - All of them move object references, even the mark and sweep collector does “occasional” compaction
  - **We can’t remember object references in BPF maps** 😒

Death-march: Java

- Garbage Collection
    - All are compacting and moving memory
    - Even the simplest collector moves memory



At program start

Free memory GC area

After running for a little bit (GC performs marking concurrently)

Allocated memory (some is garbage)

GC finishes a full cycle of mark and sweep

live

live

GC does compaction

live

Death-march: Java

- Managed stacks
  - Stacks are stored on the heap when virtual threads are used and heap references move
  - **Uretprobes can't work** 😒
- Managed threads
  - Yes, if virtual threads are used
  - There's a dedicated current thread register, so it's easy to find a key
- What linkage (or calling convention) does the program use?
  - Uses register calling convention, but it's not the same as the System V ABI

Solutions for Java

Solutions are somewhat similar to the solutions for Go:

- Use uprobes only, use the dedicated VM Thread register to find a key for BPF maps
- Don’t remember references, assume everything will move
- If you need to read data from the Java heap, read on method enter unless the method returns a reference
- Instrument more than one method to read something like a received buffer
- We can adapt our argument register macros to match the Java linkage

So we said use uprobes?

- We can only instrument JIT compiled methods
  - Java programs start interpreted, most useful methods get compiled
- JIT compiled methods are difficult to deal with:
  - They are generated on the fly, there are no binary files to inspect
  - The JVM will regularly recompile methods, probes must be dynamically inserted as compile events happen
  - Inlining is unstable and driven by runtime profiling
    - Multiple symbols need to be instrumented sometimes to overcome this challenge
  - The runtime patches the code, disassembly might not always work for attaching probes on ‘return’ on some platforms (e.g. x86)

How do we find Java symbols?

- We need to keep monitoring the compiled methods
    - Attach a uprobe to libjvm.so on ***register_nmethod***
    - Alternatively, we can attach a Java agent to get us the compilation events
- If the executable is GraalVM native compiled binary, this is just like any other binary
    - E.g: Java_java_util_zip_Inflater_inflateBytesBytes
- If we started after java, we need to get a list of all existing compiled methods
    - Without JVM options this requires running ***jcmd*** or similar programs which attach to the JVM.
    - If we can control the launch of the java program, we can get a compilation method log
        - -XX:+UnlockDiagnosticVMOptions -XX:+LogCompilation (e.g. with JDK_JAVA_OPTIONS/JAVA_TOOL_OPTIONS)

Impossible: Interpreters and Trace Compilers

- Interpreters interpret code
  - There’s are no function/method symbols to attach a probe to
- Partially compiled methods
  - Once called, neverending methods/functions
- Trace JIT compilers (e.g. LuaJIT)
  - No method/function compilation boundaries
  - Only traces of hot-paths
- We can still attach to any native statically compiled libraries the runtime uses
  - E.g: Python makes extensive use of native libraries

What we've implemented so far

- We have released full support for Go program instrumentation
- We work on OpenTelemetry observability
  - Java, .NET are well supported with OpenTelemetry auto-instrumentation
  - Our primary interest in Java, .NET and others is related to instrumenting native compiled binaries, e.g. GraalVM Native Image, .NET Native AOT…
- We'll be adding support for more user-level instrumentation for more managed runtimes
- eBPF is also great for getting runtime metrics from the managed runtime
  - We can use uprobes to find: GC times, number of goroutines in flight, event loop lag…

Summary

- We can instrument programs build on top of managed runtimes by using eBPF
- Some managed runtime programs are easier to instrument than others
- Some managed runtime programs are impossible to instrument
- Typical approaches for instrumenting statically compiled programs must be adjusted to match the runtime environment reality

Connect with us

Cilium + eBPF Day EUROPE

- You can find us on the CNCF Slack
- We are also on the Cilium & eBPF Slack
- #ebpf on the Grafana Labs Community Slack

Thank you!

Connect with us at

https://github.com/grafana/beyla