Cillium Tracepoint Example Analysis

Mark Yoshikawa

Apr 27, 2023

Introduction

This document is for self-learning of eBPF programs is work in progress grokking as I go thru the code. Plan to do more analysis docs after this one. Please contact me at mark.yoshikawa@gmail.com for any questions. Date above is the last time edited.

Imports

Links to documentation for packages used…

"errors" - Helps create text only error messages with New

"log" - Creates Logger toe messages to standard error and prints the date and time of each logged message on a separate line: if the message being printed does not end in a newline, the logger will add one. The Fatal functions call os.Exit(1) after writing the log message. The Panic functions call panic after writing the log message.

"os"- By default, a synchronous signal is converted into a run-time panic. A SIGHUP, SIGINT, or SIGTERM signal causes the program to exit. A SIGQUIT, SIGILL, SIGTRAP, SIGABRT, SIGSTKFLT, SIGEMT, or SIGSYS signal causes the program to exit with a stack dump. A SIGTSTP, SIGTTIN, or SIGTTOU signal gets the system default behavior (these signals are used by the shell for job control). The SIGPROF signal is handled directly by the Go runtime to implement runtime.CPUProfile. Other signals will be caught but no action will be taken.

"syscall" - Package syscall contains an interface to the low-level operating system primitives. The details vary depending on the underlying system, and by default, godoc will display the syscall documentation for the current system. If you want godoc to display syscall documentation for another system, set $GOOS and $GOARCH to the desired system. For example, if you want to view documentation for freebsd/arm on linux/amd64, set $GOOS to freebsd and $GOARCH to arm. The primary use of syscall is inside other packages that provide a more portable interface to the system, such as "os", "time" and "net". Use those packages rather than this one if you can. For details of the functions and data types in this package consult the manuals for the appropriate operating system. These calls return err == nil to indicate success; otherwise err is an operating system error describing the failure. On most systems, that error has type syscall.Errno. Deprecated: this package is locked down. Callers should use the corresponding package in the golang.org/x/sys repository instead. That is also where updates required by new systems or versions should be applied. See https://golang.org/s/go1.4-syscall for more information

"github.com/cilium/ebpf" - Package ebpf is a toolkit for working with eBPF programs. eBPF programs are small snippets of code which are executed directly in a VM in the Linux kernel, which makes them very fast and flexible. Many Linux subsystems now accept eBPF

programs. This makes it possible to implement highly application specific logic inside the kernel, without having to modify the actual kernel itself. This package is designed for long-running processes which want to use eBPF to implement part of their application logic. It has no run-time dependencies outside of the library and the Linux kernel itself. eBPF code should be compiled ahead of time using clang, and shipped with your application as any other resource. This package doesn't include code required to attach eBPF to Linux subsystems, since this varies per subsystem.

"github.com/cilium/ebpf/asm" - Package asm is an assembler for eBPF bytecode.

"github.com/cilium/ebpf/link" - Package link allows attaching eBPF programs to various kernel hooks.

"github.com/cilium/ebpf/perf" - Package perf allows interacting with Linux perf_events. BPF allows submitting custom perf_events to a ring-buffer set up by userspace. This is very useful to push things like packet samples from BPF to a daemon running in user space.

"github.com/cilium/ebpf/rlimit" - Package rlimit allows raising RLIMIT_MEMLOCK if necessary for the use of BPF.

Globals

ProgramSpec is a golang structure w.r.t. to the go package for eBPF. It covers what properties the eBPF program will use to interact with the kernel. Here we define selected properties for this example.

main()

The main program makes use of Golang *make* to create a slice from the *chan* *os.Signal* for goroutine communication to other goroutines. In this case to allow using *signal.Notify* to set the OS to terminate the the program through CTRL-C for instance

We check for errors using rlimit.RemoveMemlock() for pre 1.15 golang to allow max use of memory for the eBPF program we are writing

```
// Create a perf event array for the kernel to write perf records to.
// These records will be read by userspace below.
events, err := ebpf.NewMap(&ebpf.MapSpec{
		Type: ebpf.PerfEventArray,
		Name: "my_perf_array",
	})
```

https://en.m.wikipedia.org/wiki/Circular_buffer
Same as ring buffer.

```
// Open a perf reader from userspace into the perf event array
```

```
// created earlier


// Close the reader when the process receives a signal, which will exit
// the read loop.


// Minimal program that writes the static value '123' to the perf ring on
// each event. Note that this program refers to the file descriptor of
// the perf event array created above, which needs to be created prior to the
// program being verified by and inserted into the kernel.

// Asm instructions
        // store the integer 123 at FP[-8]

        // load registers with arguments for call of FnPerfEventOutput

        // call FnPerfEventOutput, an eBPF kernel helper

        // set exit code to 0

// Instantiate and insert the program into the kernel.

// Open a trace event based on a pre-existing kernel hook (tracepoint).
// Each time a userspace program uses the 'openat()' syscall, the eBPF
// program specified above will be executed and a '123' value will appear
// in the perf ring.
```