INTRODUCTION TO CLOUD SYSTEMS

Lecture 4 – Alpine, UBI Images, CNI, eBPF

Previous lecture

Alpine Linux

Alpine Linux is a Linux distribution built around musl libc and BusyBox. The image is only 5 MB in size and has access to a package repository that is much more complete than other BusyBox based images. This makes Alpine Linux a great image base for utilities and even production applications.

Pros:

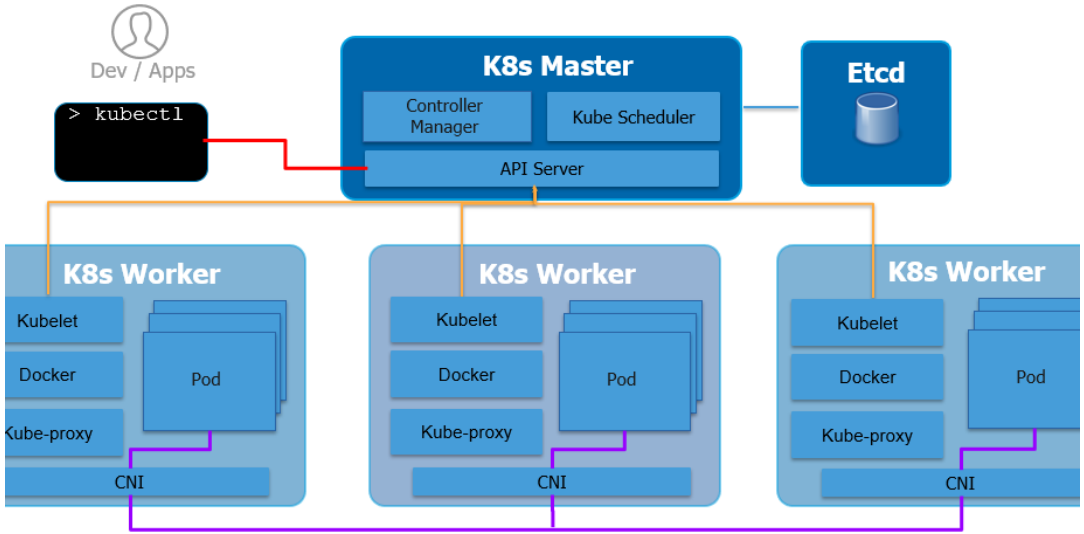
- It has a smaller footprint, and therefore a smaller attack surface
- It takes up less disk space.
- It offers a strong base for customization.
- It’s built with simplicity in mind.

UBI Images

Red Hat Universal Base Images (UBI) are OCI-compliant container base operating system images with complementary runtime languages and packages that are freely redistributable. They are built from portions of Red Hat Enterprise Linux (RHEL). UBI images can be obtained from the Red Hat container catalog and be built and deployed anywhere. Red Hat Universal Base Images (UBI) provide the same quality trusted foundation for building container images as their non-UBI predecessors (rhel6, rhel7, rhel-init, and rhel-minimal base images), but offer more freedom in how they are used and distributed.

CNI – Container Network Interface

CNI is a network framework that allows the dynamic configuration of networking resources through a group of Go-written specifications and libraries.



CNI – Container Network Interface

A CNI plugin is responsible for inserting a network interface into the container network namespace (e.g., one end of a virtual ethernet (veth) pair) and making any necessary changes on the host (e.g., attaching the other end of the veth into a bridge). It then assigns an IP address to the interface and sets up the routes consistent with the IP Address Management section by invoking the appropriate IP Address Management (IPAM) plugin.

The container/pod initially has no network interface. The container runtime calls the CNI plugin with verbs such as ADD, DEL, CHECK, etc. ADD creates a new network interface for the container, and details of what is to be added are passed to CNI via JSON payload.

CNI – Container Network Interface

Two major parts of the CNI:

1. Specification documents
   - Libcni – runtime implementation
   - Skel – referenece plugin implementation
2. Refereneces and example plugins
   - Interface plugins – ptp, bridge, macvlan…
   - Other plugins – portmap, bandwidth, tuning

CNI – Container Network Interface

Specification:

- Vendor neutral
- Also used by CRI-O, Mesos, CloudFoundry…
- Describe a basic execution flow and configuration format for network operations
- Mostly try to keep things simple and backwards compatible

CNI – Container Network Interface

Most popular CNI frameworks:

- Flannel
- Calico
- Cilium
- Weavenet
- Canal

CNI – Container Network Interface

Calico - an open source networking and network security solution for containers, virtual machines, and native host-based workloads. Powering 1.5M+ nodes daily across 166 countries.

Pros:

- Multi dataplanes (Linux eBPF, Standard Linux, Windows HNS)
- Support for Network Policies
- High network performance
- Active community

Cons

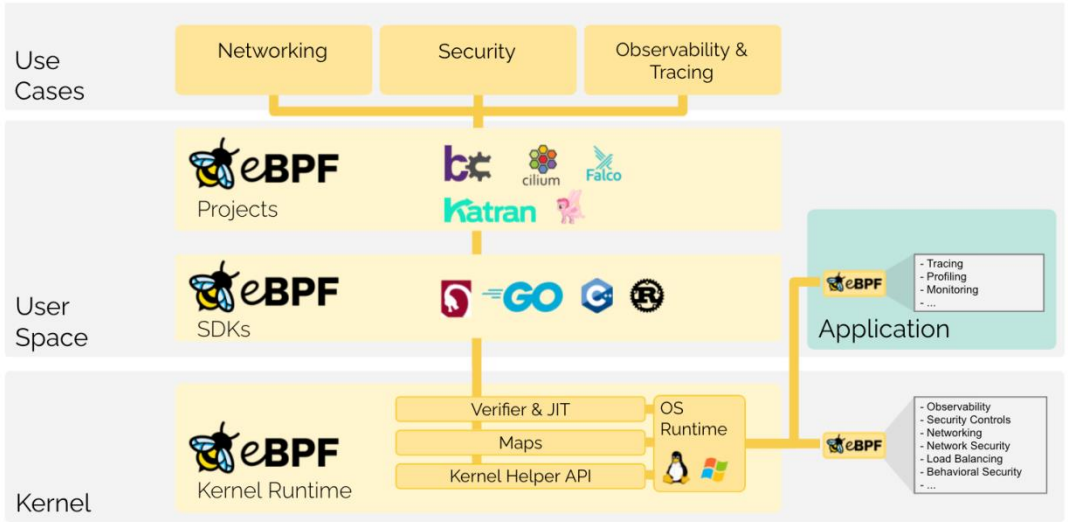
- No multicast support

eBPF

eBPF stands for “extended Berkeley Packet Filter”.

eBPF

A virtual machine embedded within the Linux kernel. It allows small programs to be loaded into the kernel, and attached to hooks, which are triggered when some event occurs.



eBPF

eBPF is typically used to trace user-space processes, and have a lot of advantages. It's a safe and useful method to ensure:

- **Speed and performance.** eBPF can move packet processing from the kernel-space and into the user-space. Likewise, eBPF is a just-in-time (JIT) compiler. After the bytecode is compiled, eBPF is invoked rather than a new interpretation of the bytecode for every method.
- **Low intrusiveness.** When leveraged as a debugger, eBPF doesn't need to stop a program to observe its state.
- **Security.** Programs are effectively sandboxed, meaning kernel source code remains protected and unchanged. The verification step ensures that resources don't get choked up with programs that run infinite loops.

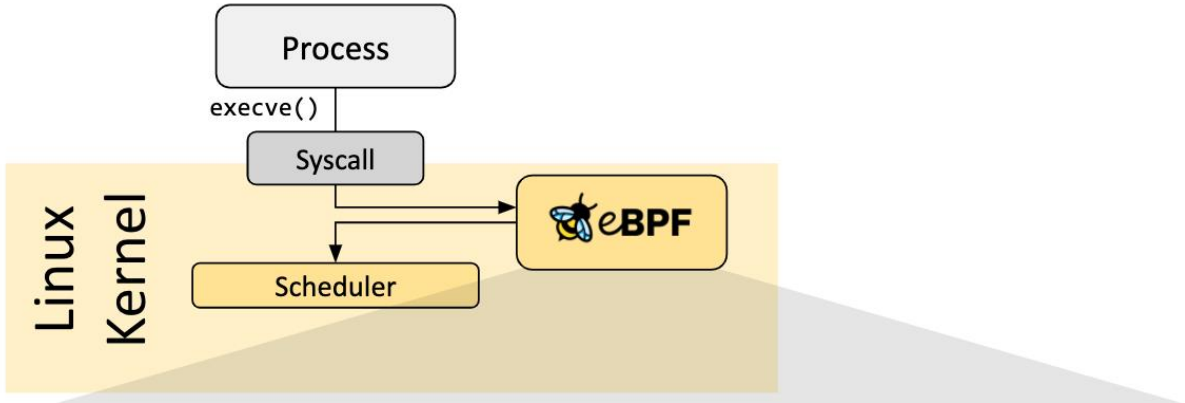
eBPF

- **Convenience.** It's less work to create code that hooks kernel functions than it is to build and maintain kernel modules.
- **Unified tracing.** eBPF gives you a single, powerful, and accessible framework for tracing processes. This increases visibility and security.
- **Programmability.** Using eBPF helps increase the feature-richness of an environment without adding additional layers. Likewise, since code is run directly in the kernel, it's possible to store data between eBPF events instead of dumping it like other tracers do.
- **Expressiveness.** eBPF is expressive, capable of performing functions usually only found in high-level languages.

eBPF

Hook overview

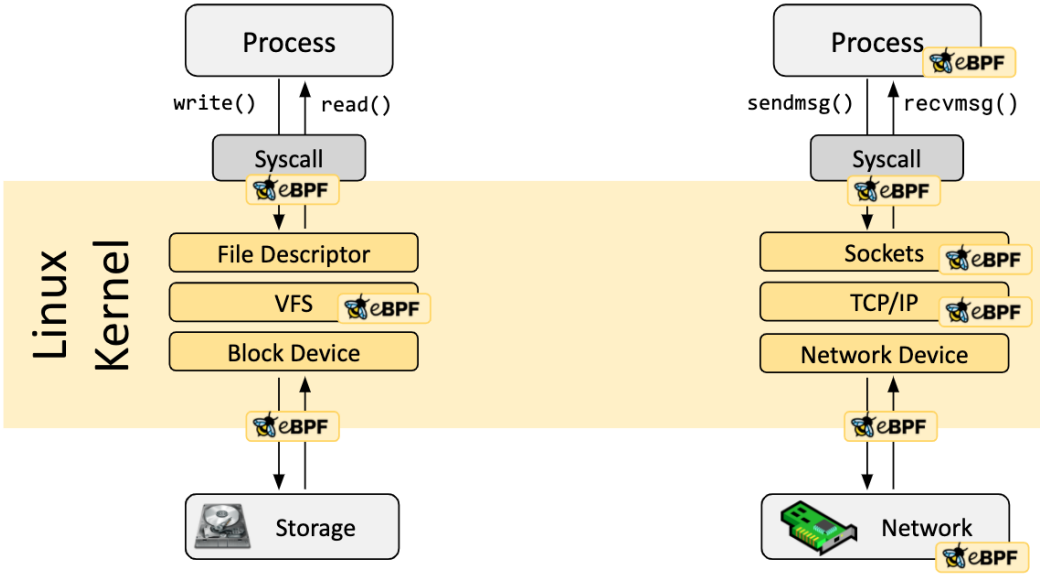
eBPF programs are event-driven and are run when the kernel or an application passes a certain hook point. Pre-defined hooks include system calls, function entry/exit, kernel tracepoints, network events, and several others.



eBPF

Hook overview

If a predefined hook does not exist for a particular need, it is possible to create a kernel probe (kprobe) or user probe (uprobe) to attach eBPF programs almost anywhere in kernel or user applications.



eBPF

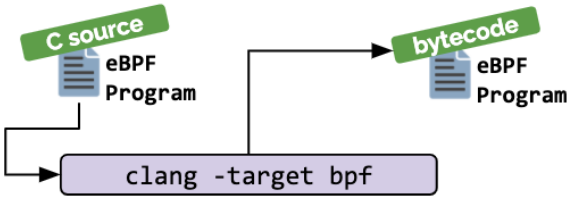
How are eBPF programs written?

In a lot of scenarios, eBPF is not used directly but indirectly via projects like Cilium, bcc, or bpftrace which provide an abstraction on top of eBPF and do not require to write programs directly but instead offer the ability to specify intent-based definitions which are then implemented with eBPF.

eBPF

How are eBPF programs written?

If no higher-level abstraction exists, programs need to be written directly. The Linux kernel expects eBPF programs to be loaded in the form of bytecode. While it is of course possible to write bytecode directly, the more common development practice is to leverage a compiler suite like LLVM to compile pseudo-C code into eBPF bytecode.



eBPF

Hook overview

There are several classes of hooks to which eBPF programs can be attached within the kernel. The capabilities of an eBPF program depend hugely on the hook to which it is attached:

- **Tracing**
- **Traffic Control (tc)**.
- **XDP**, or “eXpress Data Path”
- **More**

eBPF

Hook overview

- **Tracing** programs can be attached to a significant proportion of the functions in the kernel. Tracing programs are useful for collecting statistics and deep-dive debugging of the kernel. Most tracing hooks only allow read-only access to the data that the function is processing but there are some that allow data to be modified. The Calico team use tracing programs to help debug Calico during development; for example, to figure out why the kernel unexpectedly dropped a packet.

eBPF

Hook overview

- **Traffic Control (tc)** programs can be attached at ingress and egress to a given network device. The kernel executes the programs once for each packet. Since the hooks are for packet processing, the kernel allows the programs to modify or extend the packet, drop the packet, mark it for queueing, or redirect the packet to another interface. Calico’s eBPF dataplane is based on this type of hook; we use tc programs to load balance Kubernetes services, to implement network policy, and, to create a fast-path for traffic of established connections.

eBPF

Hook overview

- **XDP**, or “eXpress Data Path”, is actually the name of an eBPF hook. Each network device has an XDP ingress hook that is triggered once for each incoming packet before the kernel allocates a socket buffer for the packet. XDP can give outstanding performance for use cases such as DoS protection (as supported in Calico’s standard Linux dataplane) and ingress load balancing (as used in facebook’s Katran). The downside of XDP is that it requires network device driver support to get good performance. XDP isn’t sufficient on its own to implement all of the logic needed for Kubernetes pod networking, but a combination of XDP and Traffic Control hooks works well.

eBPF

Hook overview

- Several types of **socket** programs hook into various operations on sockets, allowing the eBPF program to, for example, change the destination IP of a newly-created socket, or force a socket to bind to the “correct” source IP address. Calico uses such programs to do connect-time load balancing of Kubernetes Services; this reduces overhead because there is no DNAT on the packet processing path.
- There are various security-related hooks that allow for program behaviour to be policed in various ways. For example, the seccomp hooks allow for syscalls to be policed in fine-grained ways.