KubeCon | CloudNativeCon

Europe 2022

WELCOME TO VALENCIA

KubeCon CloudNativeCon Europe 2022

# Securing Kubernetes Applications by Crafting Custom Seccomp Profiles

Sascha Grunert

# Contents of this session



1. A brief history about seccomp in Kubernetes
2. Crafting a custom seccomp profile by hand
3. Automating away those manual efforts
4. The bright future of a per-default more secure Kubernetes

# A brief history about seccomp in Kubernetes



seccomp is a syscall (Linux API) interceptor feature of the Linux Kernel

- Can boost application security by limiting the list of allowed syscalls
- Supports different actions if a syscall rule has been hit

Has been added to Kubernetes a long time ago

- Supports a default profile chosen by the container runtime like containerd, CRI-O, docker (shim)

# A brief history about seccomp in Kubernetes



seccomp is GA (generally available) since Kubernetes v1.19.0

- Can be considered as stable
- Supported by most Linux kernel versions

Usable via the API field or the annotation (deprecated) per container or whole pod

- Goal to remove the annotation support in v1.25.0

# A brief history about seccomp in Kubernetes



All workloads run unconfined (disabled seccomp) by default

- Special feature `SeccompDefault` allows to change that behavior (alpha)
- Default profiles may differ between container runtimes

Custom profiles have to be defined as JSON files

- The profiles have to be distributed to all nodes
- Container runtime applies the profile from disk

# Crafting a custom seccomp profile by hand



The overall goal is to understand

- How seccomp profiles work and which possibilities they allow
- How my application behaves and which syscalls it executes during runtime
- Which syscalls are required to be allowed additionally

We have to take the whole cluster setup into account to not create too restrictive profiles

- Different Linux architectures require different syscalls
- The workload configuration may have influence on executed syscalls

# Crafting a custom seccomp profile by hand



Example project: **kube-rbac-proxy**

- HTTP proxy to perform RBAC authorization against the Kubernetes API
- Allows to restrict requests to the API
- Initially developed to protect Prometheus metrics endpoints
- Single container deployment (simplifies the syscall tracing)

https://github.com/brancz/kube-rbac-proxy

# Crafting a custom seccomp profile by hand



Collecting the syscalls method #1 - tracing the Kernel logs

- Requires auditd or syslog to be installed and configured

Kernel log rate limiting may trick us:

```
> sysctl -w kernel.printk_ratelimit=0

> sysctl -w kernel.printk_ratelimit_burst=0
```

# Crafting a custom seccomp profile by hand



Tracing the audit_seccomp Kernel function:

```
3005  void audit_seccomp(unsigned long syscall, long signr, int code)
3006  {
3007          struct audit_buffer *ab;
3008
3009          ab = audit_log_start(audit_context(), GFP_KERNEL, AUDIT_SECCOMP);
3010          if (unlikely(!ab))
3011                  return;
3012          audit_log_task(ab);
3013          audit_log_format(ab, " sig=%ld arch=%x syscall=%ld compat=%d ip=0x%lx code=0x%x",
3014                           signr, syscall_get_arch(current), syscall,
3015                           in_compat_syscall(), KSTK_EIP(current), code);
3016          audit_log_end(ab);
3017  }
```

# Crafting a custom seccomp profile by hand



syscalls will only get logged if requested

- Logging has a high performance impact (blocks the application)
- A special seccomp action is available for logging

Creating a logger profile for the application:

```
{ "defaultAction": "SCMP_ACT_LOG" }
```

Double check that `log` is part of `/proc/sys/kernel/seccomp/actions_logged`

# Crafting a custom seccomp profile by hand



Put the profile into `/var/lib/kubelet/seccomp/log.json`

Change the application to use the profile as part of the SecurityContext:

```
spec:
  containers:
  - name: …
    securityContext:
      seccompProfile:
        type: Localhost
        localhostProfile: log.json
```

# Crafting a custom seccomp profile by hand



Now we can finally run our demo application and trace /var/log/audit/audit.log

- Beware that those files rotate based on their size
- auditd can be configured with a rate limit (check `sudo auditctl -s`)

We need to find the process identifier (PID) of the container:

```
> export CTR=$(sudo crictl ps --name kube-rbac-proxy -q)
> export PID=$(sudo crictl inspect $CTR | jd .info.pid)
```

# Crafting a custom seccomp profile by hand



Finally obtain a list of syscalls:

```
> sudo cat /var/log/audit/audit.log | \
    rg type=SECCOMP | rg pid=$PID | \
    rg -o "SYSCALL=(.*)" -r '$1' | sort -u | tr '\n' ' '
```

```
accept4 arch_prctl bind clone close epoll_create1 epoll_ctl
epoll_pwait execve fcntl fstat futex getpid getrandom getsockname
gettid listen madvise mmap nanosleep newfstatat openat pipe2 pread
read readlinkat rt_sigaction rt_sigprocmask rt_sigreturn
sched_getaffinity sched_yield setsockopt sigaltstack socket tgkill
uname write
```

# Crafting a custom seccomp profile by hand



The list of syscalls reflects only the startup of the application, not its usage.

Running the client usage example will reveal additionally required syscalls:

```
> sudo cat …

… connect … getpeername … getsockopt …
```

The execution of all available code paths is crucial for this approach.

# Crafting a custom seccomp profile by hand



Creating an allowlist is now possible for the deployment:

```
{
  "defaultAction": "SCMP_ACT_ERRNO",
  "defaultErrnoRet": 1,
  "defaultErrno": "EPERM",
  "syscalls": [{
    "action": "SCMP_ACT_ALLOW",
    "names": [
      "accept4",
      …
      "write"
    ]
  }]
}
```

# Crafting a custom seccomp profile by hand



Thoughts on the overall approach:

- Creating profiles via the logs can be slow when considering CI/CD based automation
- Nodes have to be pre-configured to not rate-limit logs
- Gathering all application code paths is hard

There is another way doing this by utilizing eBPF (https://ebpf.io)

# Crafting a custom seccomp profile by hand



eBPF provides enter and exit tracepoints for syscalls

- `tracepoint:raw_syscalls:sys_enter` provides the syscall ID for every execution on the system

The system global scope of eBPF applications requires us to correlate the collected syscalls to the container process.

# Crafting a custom seccomp profile by hand



For example, using tools like bpftrace already allows us to collect the required data:

```
> sudo bpftrace -e 'tracepoint:raw_syscalls:sys_enter
    /comm == "kube-rbac-proxy"/
    { printf("%d: %d\n", pid, args->id); }' > output

# Start the workload and abort the script when tests are done

> rg "$PID:\s(\d+)" -r '$1' output | sort -u | tr '\n' ' '

0 1 13 131 14 15 158 186 202 204 233 234 24 257 262 267 28 281 288
291 293 3 318 35 39 41 49 5 50 51 54 56 63 72 9
```

# Crafting a custom seccomp profile by hand



The syscall numbers can be converted back into the name by using ausyscall:

```
> ausyscall --dump
Using x86_64 syscall table:
0       read
1       write
2       open
…
```

This way it is possible to create seccomp profiles manually without having to write an

eBPF application from scratch.

# Crafting a custom seccomp profile by hand



Thoughts on this approach:

- Creating profiles would not affect system performance in the same way as the logging
- Nodes have to be pre-configured to contain either the custom eBPF application or the dependent tools
- Gathering all application code paths is still hard

There must be a better way in doing this…

# Automating away those manual efforts

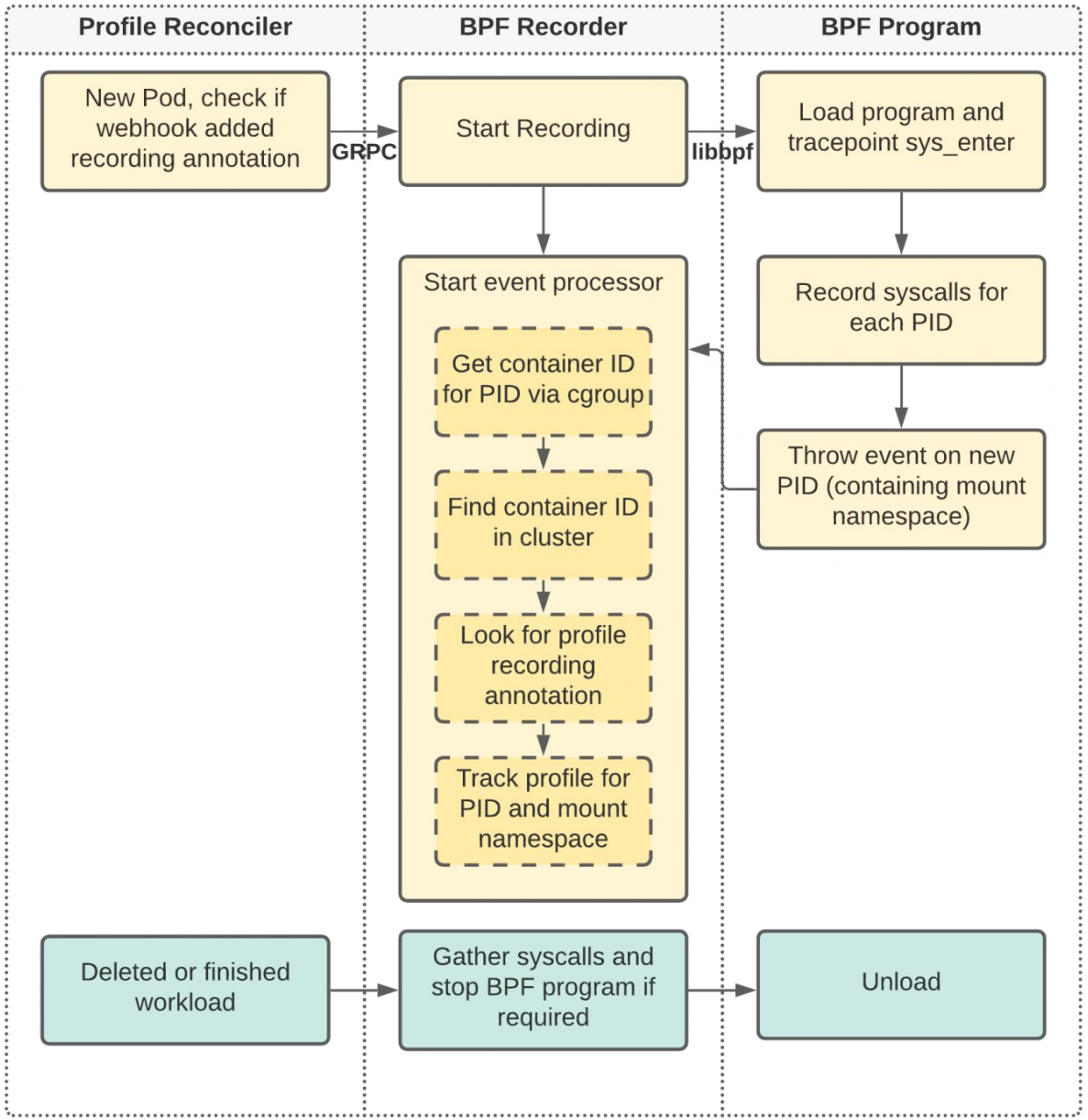


The Security Profiles Operator provides automation around log and eBPF based profile recording: https://sigs.k8s.io/security-profiles-operator

- It automatically traces the logs at the right time and extracts the data
- It can use eBPF to record profiles with automatic correlation of the workload to the underlying process
- It creates seccomp profiles after a recording has been finished
- It automatically reconciles the profiles to all nodes

# Automating away those manual efforts





# Automating away those manual efforts



Thoughts on this approach:

- We mostly get rid of all manual collection obstacles
- Gathering all application code paths is the hardest part

The integration into a CI/CD workflow would allow updating the seccomp profiles with the application lifecycle.

The operator could be used in production to distribute the profiles by using the CRD.

# The bright future of a per-default more secure Kubernetes



The SeccompDefault feature should graduate to beta in Kubernetes v1.25.0

- Since Kubernetes v1.24.0 beta features are not enabled by default any more
- Graduating the feature to stable would gain us a security boost in Kubernetes
- Handling upgrade paths is the most complex part of the graduation

Please, help us achieving this by:

- Using custom seccomp profiles or RuntimeDefault for your applications
- Enable SeccompDefault in Kubernetes

# Securing Kubernetes Applications by Crafting Custom Seccomp Profiles

Thank you for listening to this talk!