安全容器分享

现状

与传统安全相比，云原生时代的安全挑战发生了很大变化，主要表现在三方面：

1. 一台服务器可能混合运行着成百上千的应用容器，其密度是传统模式的数十倍
2. 容器和镜像加速了敏捷快速迭代的效率，应用变更更加频繁
3. 云原生时代，越来越多的开源技术和第三方不可信应用被引入到企业内部

相较于以往一台机器部署单一应用，容器化后，一台机器混合部署多种应用，提高了应用风险。

参考资料

容器安全分类

运行时安全

- 安全容器运行时（安全容器，隔离不可信应用、阻断恶意非法访问）
- 网络隔离（Network Policy等）
- 漏洞检测（主机OS，Guest OS等）
- 安全监控（进程异常行为、网络活动检测等）

数据安全

- 镜像安全（镜像签名、镜像扫描）
- 数据保护（存储状态数据、传输状态数据、使用中或计算中状态数据）

用户权限

- 完善认证和鉴权机制
- 细粒度权限管理

传统容器当前存在的问题

以传统容器运行时 Runc 为例，最大的问题就是**隔离问题**。

- 系统隔离（安全隔离）
- 数据隔离（安全隔离）
- 网络隔离（安全隔离）
- 性能隔离
- 故障隔离

比如隔离不足、资源争抢导致延迟敏感型应用抖动、主机上的容器故障（CPU 泄露、频繁 CoreDump 等）都会影响主机上的其他容器。

此外，由于 runC 容器基于 Namespace 技术隔离，导致大部分非 Namespace 类别的内核参数的设置会影响主机上的所有其他容器。

容器漏洞 POC

- Dirty CoW

  Linux漏洞导致的容器逃逸。Dirty CoW是由于内核内存管理系统实现CoW时产生的漏洞。通过条件竞争，把握好在恰当的时机，利用CoW的特性可以将文件的read-only映射改为write。子进程不停地检查是否成功写入。父进程创建二个线程，ptrace_thread线程向vDSO写入shellcode。madvise_thread线程释放vDSO映射空间，影响ptrace_thread线程CoW的过程，产生条件竞争，当条件触发就能写入成功。执行shellcode，等待从宿主机返回root shell，成功后恢复vDSO原始数据。

  POC

- CVE-2019-5736：runc - container breakout vulnerability

  runc在使用文件系统描述符时存在漏洞，该漏洞可导致特权容器被利用，造成容器逃逸以及访问宿主机文件系统；攻击者也可以使用恶意镜像，或修改运行中的容器内的配置来利用此漏洞。

  POC

参考资料

安全容器运行时技术对比

	安全容器运行时分类			
	OS容器+安全机制	用户态内核	Library OS	MicroVM
内核模式	共享内核	独立内核		
典型代表	Docker OS容器	gVisor	UniKernel Nabla Containers	Kata-Containers Firecracker
特点	• RunC 共享内核。 • 访问控制增强：SELinux、AppArmor、Seccomp等。 • Rootless Mode (docker 19.03+)	• 独立用户态内核，代理应用所有系统调用。 • 进程虚拟化增强。	• 基于 LibOS技术，内核深度裁剪，启动速度较快。 • 内核需与应用编译打包在一起。	• 基于轻量虚拟化技术，扩展能力优。 • 内核OS可定制。 • 应用兼容性优。 • 安全性最优。
缺点	• 无法有效防范内核漏洞问题。 • 安全访问控制工具对管理员认知和技能要求高。 • 安全性相对最差。	• 应用兼容性以及系统调用性能较差。 • 不支持 Virtio，扩展性较差。	• 应用兼容性差。 • 应用和LibOS的捆绑编译和部署为传统的 DevOPS 带来挑战。	• Overhead 较大，启动速度相对较慢。

OS容器+安全机制

主要原理在于在传统的OS容器基础上增加一些辅助手段来提高安全性，例如SELinux、AppArmor、Seccomp等。以及Docker 19.03+ 让docker处于rootless模式下。

缺点：在于无法解决容器于Host共享内核，利用内核漏洞逃逸带来的安全隐患问题。

同时这些安全访问控制工具对管理员的认知和技能要求较高，安全性相对来说也不够理想。

用户态内核

典型代表Google的gVisor，通过实现独立的用户态内核去捕捉和代理应用的所有系统调用，隔离非安全的系统调用，间接性达到安全目的，属于进程虚拟化增强。

缺点：系统调用的代理和过滤的这种机制，导致其应用兼容性以及系统调用方面性能相对传统OS容器较差。由于不支持virt-io等虚拟框架，扩展性较差，不支持设备热插拔。

Library OS

基于LibOS 技术的安全容器运行时。例如 Unikernel、Nabla-Containers。

LibOS 技术本质上是针对应用内核的深度定制和裁减，需要把LibOS 与应用编译打包在一起。

缺点：将OS与应用打包，本身兼容性比较差，应用和LibOS的捆绑编译和部署，为传统的DevOps带来挑战。

MicroVM

MicroVM 轻量虚拟化技术是对传统虚拟化技术的裁减，例如：kata-containers(借助qemu、firecracker的VMM能力)、Firecracker(VMM，Secure and fast microVMs for serverless computing)。VM GuestOS包括内核均可以自由定制。由于具备完整的OS和内核，应用兼容性优秀。

独立内核的好处在于即使出现安全漏洞，也可以将安全影响范围限制在一个VM内部。

缺点：相对于传统容器，引入VM开销，容器启动速度稍有影响。

小结

1. 对大部分内部可信业务来说，这类应用安全风险较低，使用普通的 runC 容器即可；
2. 对来自三方、开源或存在潜在漏洞风险的应用，需要通过安全容器进行隔离；若追求高执行效率和更小的开销，可以考虑 LibOS 形态的安全容器，缺点是需要 LibOS 和应用捆绑编译；若追求通用性，可以考虑基于轻量虚拟机技术的安全容器，比如开源的 Kata Containers、firecracker。另外，开源的用户态内核实现，比如 gVisor，虽然开销较小、启动速度较快，但是兼容性稍差于 runC 以及轻量虚拟机安全容器。

kata 安全容器特点

1. 安全：在专用内核中运行，提供网络，I / O和内存的**隔离**，并可以使用虚拟化技术使硬件强制隔离。
2. 兼容性：支持行业标准，包括OCI容器格式，Kubernetes CRI接口以及传统虚拟化技术。
3. 性能：提供与标准Linux容器一致的性能；提高隔离度，同时没有标准虚拟机的性能开销。
4. 简单：消除了在完整的虚拟机内部嵌套容器的要求；标准接口使启动和硬件插拔变得容易。

kata 安全容器缺陷

存储性能不够理想

kata采用轻量级虚拟机和硬件虚拟化技术来提供强隔离，以构建安全的容器运行时。由于虚拟机的使用导致容器根文件系统(rootfs无法和runc一样直接挂载Host目录，而是需要将host目录共享给guest。

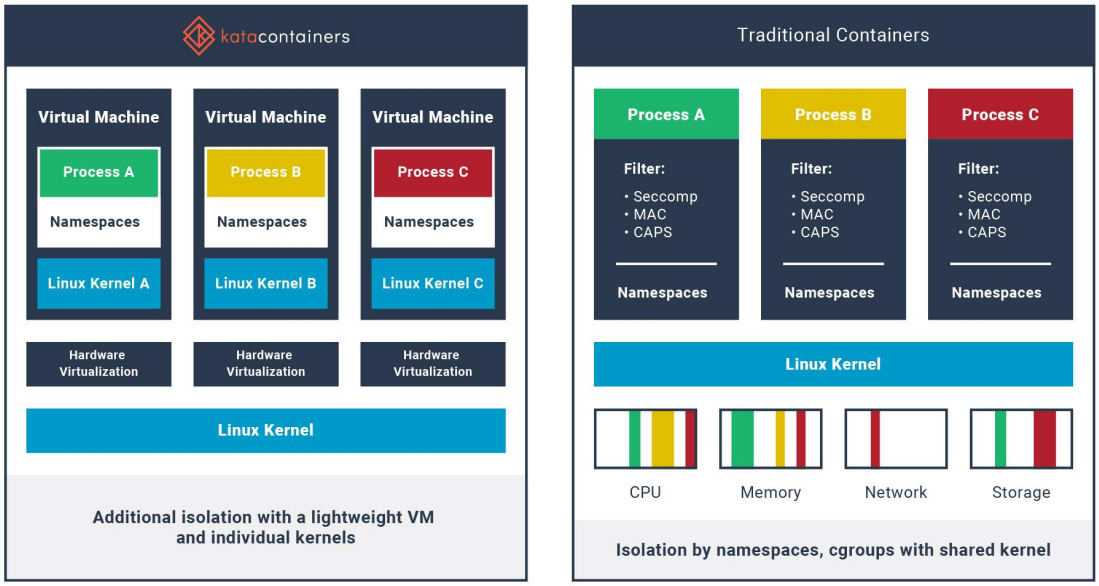
存在两种方式实现：

- 基于文件的9pfs
    - 优点：因为使用host上overlayfs，能充分利用host pagecache；部署简单，不需要额外的组件
    - 缺点：基于网络的协议，性能差；POSIX语义兼容性方面不好（主要体现在mmap(2)的支持上，在特殊情况下有数据丢失的风险）
- 基于块设备的devicemapper
    - 优点：良好的性能；很好的POSIX语义兼容性
    - 缺点：无法充分利用host pagecache，部署运维复杂（需要维护lvm volume）

目前kata 1.10 默认是用9pfs。**痛点**

参考资料

kata 与传统容器的对比



kata 编译、安装、部署

Kata-Containers 编译、安装、部署

kata 如何与 Kubernetes 集成

1. 编译 runtime 仓库，获取 containerd-shim-kata-v2
2. 修改 containerd 配置

   当前环境 containerd 支持两种运行时：

   - runc

   - kata runtime(containerd-shim-kata-v2)

   containerd 配置文件可以看到运行时支持

```
# /etc/containerd/config.toml
[plugins."io.containerd.grpc.v1.cri".containerd]
...
      [plugins."io.containerd.grpc.v1.cri".containerd.runtimes]
        [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc]
          runtime_type = "io.containerd.runc.v2"
...
        [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.ecr]
         runtime_type = "io.containerd.ecr.v2"
```

3. 创建 RuntimeClass

   Kubernetes 1.16 版本通过 容器运行时类(RuntimeClass) 可指定Pod运行使用的容器运行时。kata容器运行时类配置如下：

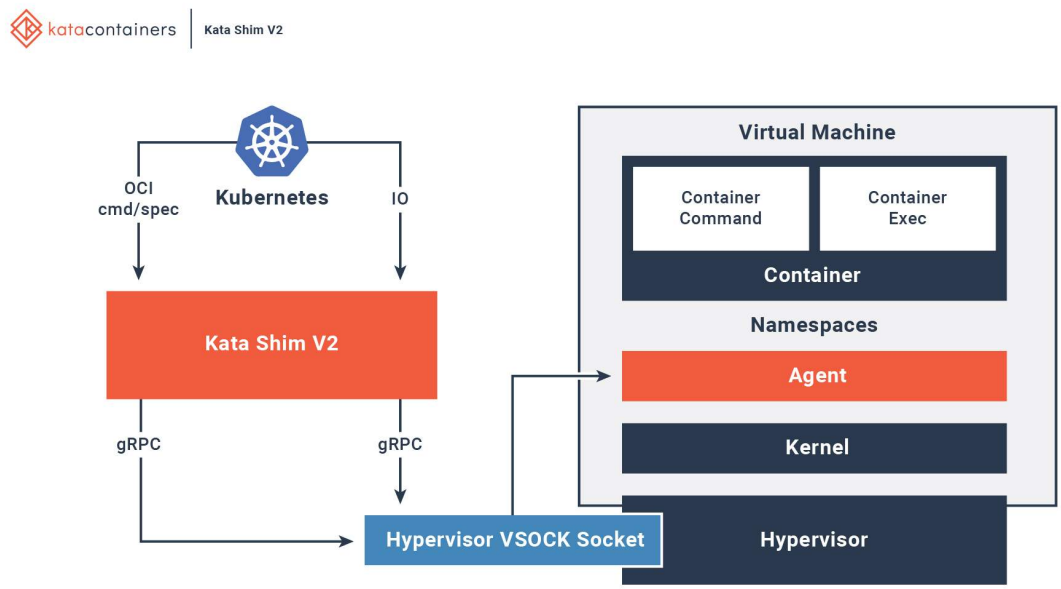
```
// kubectl get runtimeclass rune -oyaml
apiVersion: node.k8s.io/v1beta1
handler: ecr
kind: RuntimeClass
metadata:
  name: rune
```

4. 创建安全容器 Pod

   创建资源时，指定RuntimeClass即可指定容器运行时，以deployment为例：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  ...
spec:
  template:
    spec:
      runtimeClassName: rune # 指定Pod使用的容器运行时
```

同一Pod内的容器创建在同一虚拟机中，虚拟机内通过 agent 组件创建并管理容器（代码实现：复用 runc/libcontainer代码）



kata 定制开发

需求概述：

Kata-containers 支持使用 qcow2 镜像创建VM

相关实现：

Runtime 实现

Govmm 实现

kata 为什么安全

1. kata 相比于传统容器引入隔离层，将容器从share kernel 变成 isolated kernel，同时具备虚拟机资源隔离特性。

kata 优点

容器层面

1. 实现OCI标准，具备普通容器一样的特点，复用容器镜像

虚拟机层面

1. Micro VM，轻量级虚拟化

隔离层的优势

1. 出现内核0day漏洞时，可以通过更新runtime使用guest内核，而不是修改host内核来修复漏洞。
2. 硬件资源强制隔离

kata v1 和 v2 版本对比

kata 2.0 ChangeLOG

- 使用 rust 重构 kata-agent，显著减少整体内存开销和攻击面。agent 体积从11MB降低到300KB。**关注**
- kata-agent 协议已简化为使用ttRPC，代替原来的gRPC。**关注**
- 引入新的组件 Kata-monitor，以提高可观察性和可管理性。**关注**
- 引入新的组件 agent-ctl，便于验证 agent API。
- 所有项目代码和文档合并到一个单一仓库中 github/kata-containers/kata-containers 。
- Virtio-fs是默认的共享文件系统类型。与virtio-9p相比，具有更好的POSIX兼容性。**关注**
- 与最新的 QEMU 具有相同的Cloud Hypervisor支持。**关注**
- 仅支持shimv2 API，简化代码并进一步减少攻击面。这也意味着`kata-shim`与`kata-proxy`只在1.x中使用，2.0 不再需要。
- Guset 虚拟机内核更新至 v5.4.71。
- QEMU 版本更新到v5.0.0。

如何构建安全的 Kubernetes 环境

kubernetes 集群安全

- 控制对kubernetes API的访问
    - API开启TLS
    - API认证
    - API鉴权
- 控制对kubelet的访问
- 控制运行时负载或用户的能力
    - 限制集群资源的使用
    - 控制容器运行的特权
    - 限制网络访问
- 保护集群组件免受破坏
    - 限制访问ETCD
    - 开启审计日志
    - 限制使用alpha和beta特性
    - 频繁回收基础设施证书

- 对secret进行静态加密
- 接收安全更新和报告漏洞的情报