在 Linux 下使用 unshare 和 pivot_root 搭建一个有限的沙盒应用

陈嘉

此文章最初是发布在开源中国的博客, 但不知由于何种原因, 导致发表的内容, 仅自己登录后可见, 所以为了让更多的人看见, 现在也发在知乎.

当然自己的另一个 ID 也暴露了.

开源中国博客地址:

> BID540602的个人空间 - OSCHINA - 中文开源技术交流社区
> my.oschina.net/u/2335770

unshare 和 nsenter 都是来自包 util-linux 的命令, 依靠它们可用于操控 Linux 的进程 namemspace, 以实现对特定系统对象或资源的隔离, 例如 /proc, uts, cgroup. 这样就可以实现 systemd-nspawn, systemd-chroot, bubblewrap, Docker/Podman 提供的大部分功能.

不过由于自己最初只测试了 urxvt, 所以没有发现其中潜在的问题. 因为要跑的应用来自一个 OS 映像文件, 而不是来自宿主系统, 所以必须使用 nsenter 命令的 --root 参数. 这个参数的核心功能用到 chroot syscal, 而这样做, 会导致在 chroot 环境的程序一旦用到 clone syscall, 会报 EPRM, 这会导致像 Chromium 和 GNOME Web(epiphany) 无法运行.

发现问题后, 也尝试用 chroot 替换 nsenter, 以及先 chroot 然后在其内调用 unshare, 但结果都一样. 由于 GNOME Web 运行需要用到 bubblewrap, 所以自己也用其进行测试, 发现这个软件不受影响, 在通过 strace -c 以及阅读其源码后, 发现了 pivot_root, 幸运的是, 在 util-linux 包有一个同名命令文件. 经过一系列尝试后, 只使用 unshare 和 pivot_root 就可以达到 chroot 的效果, 同时成功运行 Chromium 和 GNOME Web.

来自于 man 2 clone 的描述

EPERM (since Linux 3.9)

CLONE_NEWUSER was specified in flags and the caller is in a chroot environment (i.e., the caller's root directory does not match the root directory of the mount namespace in which it resides).

验证方法

Chromium 在地址栏输入 chrome://sandbox. 可以看到其显示

```
Sandbox Status
 Layer 1 Sandbox Namespace
 PID namespaces Yes
 Network namespaces Yes
 Seccomp-BPF sandbox Yes
 Seccomp-BPF sandbox supports TSYNC Yes
 Ptrace Protection with Yama LSM (Broker) Yes
 Ptrace Protection with Yama LSM (Non-broker) No
 You are adequately sandboxed.
```

GNOME Web

从终端执行命令 epiphany, 输出的调试信息不会有 Crashed 字样, 同时浏览器可正常打开页面. 如果有问题, 选项卡页面只会显示一片空白.

标题之所以提到的有限的沙盒应用, 是因为这个方案其用途是跑桌面应用. 而桌面应用由于各种各样的现实制约, 没法实现很严格的沙盒机制. 比如 Xorg 的自身缺陷, 没有用户访问资源的分离机制. 隔离 IPC 会导致明显的图形性能下降. 故本文介绍的方法, 是按照如下机制进行隔离:

0. 隔离应用所依存的 OS 环境, 不使用宿主系统, 而是通过自制脚本搭建 OS 容器映像. 这样的好处是, 容器映像可通过 OS 对应的包管理器获得持续更新. 可控制宿主系统的包数量, 可以实现宿主系统是纯 64 位系统, 而容器内运行 32 位应用, 例如 WINE.

要获取自制脚本, 以及搭建 OS 容器方法, 请参见在开源中国的博客文章.

1. UID/GID 是特定用户. 不与登录到桌面环境的用户一样, 也更不可能是 root 用户. 自己之所做的 systemd-chroot, 就是依靠此来限制容器应用的宿主系统其他进程的探知.

2. 隔离 /proc, 只能看到同一个 /proc 下的进程.

3. 应用和数据分离. 程序文件所属的目录, 大部分被设置为 ro, 仅个别以及用户目录被设置为 rw.

4. 隔离不同应用的用户目录. 出去潜在的安全考虑, 没有将应用所属的用户目录放在 /home 下.

由于本人时间有限, 所以目前的文章, 仅作为知识分享, 予以发布. 提供的内容, 并没有细细描述, 需要有一定基础. 文章也没有什么排版.

风险自担

正文

宿主系统

Ubuntu 18.04 + hwe 5.4 内核

需要说明的是, 18.04 还有一个默认内核 4.15, 由 Canonical 反向移植和维护. 这个内核是否运行正常, 尚未测试, 不过预估没有问题, 因为之前用 systemd-nspawn 和 systemd-chroot 是正常的.

这里提供的脚本还是属于概念验证.

容器应用

容器应用所属 OS 为 Arch Linux. 本身是用于运行 systemd-chroot 项目, 在此基础上,

进行了微小的调整, 新建目录和放置脚本. 就使得同一个 OS 映像文件, 为两个项目服务.

运行容器应用的用户

这里假定, 用户名为 egret, 其 UID 和 GID 皆为 2015.

大致说一下, 需要注意的地方. 同时在宿主系统和容器系统都创建该用户. 如果用户需要

使用诸如 Intel 核显的设备文件 /dev/dri/{card0,renderD128}, 那么有三种方法.

第一种是只在宿主系统, 将 egret 添加到设备文件所属的组(gpasswd). 但这种方法,

由于未知原因, 不适用于 unshare 和 nsenter 方案.

注: 在变更为 pivot_root 方案, 尚未测试是否有所改变.

第二种, 使用 setfacl 为对应的设备文件添加 egret 的 ACL rw 权限. 这种方法的

适用性最好, 但如果存在一种机制, 强制定时重置 /dev 下的 ACL 权限, 则此方法会

带来问题.

第三种, 如果设备文件所属的组不存在于容器内系统, 则使用 groupadd 添加该组,

但不需要将 egret 也添加到这些组里. 不过这种方法存在潜在问题, 如果对应的

GID 已存在于容器内系统, 但作用域不同, 比如是访问某种高权限资源, 则这个方法

可能会导致容器内的用户, 发生权限提升的可能, 即便该提升是意外的.

运行在宿主系统下的脚本 - control-sandbox-app

```
#!/bin/bash

set -o nounset
export IFS=$'\n'

[ ${#@} -ne 2 ] && exit 1

declare -r  USER_NAME=$1
declare -r  COMM=$2
declare -r  D_IMG_D='/entry/arch/apps/rw'
declare -r  J_IMG_D='/entry/arch/'${COMM}'/ro'
declare -r  PUT_OLD='host'
declare -r  D_USER_D='/data/arch/'${COMM}'/user'
declare -r  J_USER_D='/entry/arch/'${COMM}'/user'
declare -r  D_TMP_D='/data/arch/'${COMM}'/tmp'
declare -r  J_TMP_D=${J_IMG_D}'/tmp'
declare -r  D_X11_D='.X11-unix'
declare -r  D_RUN_D='/run/arch.'${COMM}
declare -r  D_PROC_D='/data/arch/'${COMM}'/sandbox/proc.d'
declare -r  D_DEV_D='/data/arch/'${COMM}'/sandbox/dev.d'
declare -r  J_RUN_D=${J_IMG_D}'/run'
declare -r  HOST_NAME='arch-'${COMM}
declare -r  J_HOME_D=${J_IMG_D}'/home/'${USER_NAME}
declare -r  FIFO='/sandbox/arch-'${COMM}
declare -r  DISPLAY=':0'

function prepare()
{
  findmnt ${J_IMG_D} &> /nul
  [ $? -ne 0 ] && \
    mount ${D_IMG_D} ${J_IMG_D} --options=bind,private,ro

  findmnt ${J_IMG_D}'/proc' &> /nul
  [ $? -ne 0 ] && \
    mount ${D_PROC_D} ${J_IMG_D}'/proc' --options=bind,private

  findmnt ${J_IMG_D}'/dev' &> /nul
  [ $? -ne 0 ] && \
    mount ${D_DEV_D} ${J_IMG_D}'/dev' --options=bind,private

  findmnt ${J_USER_D} &> /nul
  [ $? -ne 0 ] && \
    mount ${D_USER_D} ${J_USER_D} --options=bind,private

  findmnt ${J_HOME_D} &> /nul
  [ $? -ne 0 ] && \
    mount ${J_USER_D} ${J_HOME_D} --options=bind,private

  findmnt ${J_TMP_D} &> /nul
  if [ $? -ne 0 ]; then
    mount ${D_TMP_D} ${J_TMP_D} --options=bind,private
    mkdir ${J_TMP_D}'/'${D_X11_D}
    mount '/tmp/'${D_X11_D} ${J_TMP_D}'/'${D_X11_D} --options=bind,private
  fi

  if [ ! -d ${D_RUN_D} ]; then
    mkdir --parent ${D_RUN_D}
    chown ${USER_NAME}:${USER_NAME} ${D_RUN_D}
  fi

  findmnt ${J_RUN_D} &> /nul
  [ $? -ne 0 ] && \
    mount ${D_RUN_D} ${J_RUN_D} --options=bind,private

  findmnt ${FIFO} &> /nul
  [ $? -ne 0 ] && \
    mount ${FIFO} ${J_IMG_D}${FIFO} --options=bind
}

function fin()
{
  umount ${J_IMG_D}${FIFO}
  umount ${J_RUN_D}
  umount ${J_TMP_D}'/'${D_X11_D}
  umount ${J_TMP_D}
  umount ${J_USER_D}
  umount ${J_HOME_D}
  umount ${J_IMG_D}'/proc'
  umount ${J_IMG_D}'/dev'
  umount ${J_IMG_D}
}

trap fin EXIT

function main()
{
  prepare

  [ ! -e ${J_IMG_D}'/etc/os-release' ] && return 1

  unshare --mount --uts --fork --pid --mount-proc \
    /usr/local/bin/ubuntu-pc.bin/enter-app ${J_IMG_D} ${USER_NAME} ${COMM}

  return 0
}

main $1 $2
```

/nul 是 /dev/null 的符号链接.

PUT_OLD 变量对应的是位于容器 OS 根目录的空白子目录. 其用于临时存放宿主系统原根

目录.

D_X11_D 是用来重绑定宿主系统的 /tmp/.X11-unix . 当然可以绑定 socket 文件.

本来这是不需要的, 但是在测试中发现, bubblewrap 会移除来自于父进程的环境变量

DISPLAY, 导致 GNOME Web 无法运行.

该脚本运行机制是, 启动 main 函数后, 先通过函数 prepare, 依靠 mount 命令的 bind

参数, 以 ro 方式构建作为容器应用的根目录, 同时 private 参数, 使得该挂载不影响到

宿主系统. 还有把用于在宿主系统向容器内应用传递指令的管道文件连接到指定位置.

最后通过 unshare 命令来启动一个从宿主系统过度到容器应用的脚本.

当 main 结束运行后, 会先自动结束所有自此脚本运行的进程. 然后通过 trap 的设置,

自动调用 fin 函数, umount 之前 bind 的目录和文件.

由于目前这个脚本主要是用来观察运行效果, 所以 prepare 和 fin 函数内, 是直接对需

要进行的操作逐一罗列, 而没有采取从配置文件读取的方式.

运行在宿主系统下的脚本 - enter-app

```
#!/bin/bash

set -o nounset
export IFS=$'\n'

function main()
{
  local -r J_IMG_D=$1
  local -r PUT_OLD=host
  local -r USER_NAME=$2
  local -r COMM=$3

  cd ${J_IMG_D}
  pivot_root . ${PUT_OLD}
  hostname arch-${COMM}
  /usr/local/bin/arch-init
  /usr/local/bin/arch-daemon ${USER_NAME} ${COMM}
}

main $1 $2 $3
```

该脚本实现 chroot 功能, 设置适当的容器名称, 构建属于容器的 /proc, /sys, /run 等

VFS 目录, 以及启动位于容器内的脚本.

容器内的脚本 - arch-init

```
#!/bin/bash

set -o nounset
export IFS=$'\n'

findmnt /proc &> /nul
[ $? -ne 0 ] && \
  mount proc /proc --types proc --options=rw,nosuid,nodev,noexec,relatime

findmnt /sys &> /nul
[ $? -ne 0 ] && \
  mount sys /sys --types sysfs --options=ro,nosuid,nodev,noexec,relatime

findmnt /dev/pts &> /nul
if [ $? -ne 0 ]; then
  mount dev /dev --types devtmpfs --options=rw,nosuid,relatime,size=1003812k,nr_inode
  mkdir --parent /dev/pts
  mount devpts /dev/pts --types devpts --options=rw,nosuid,noexec,relatime,gid=5,mode
fi

findmnt /dev/shm &> /nul
if [ $? -ne 0 ]; then
  mkdir --parent /dev/shm
  mount tmpfs /dev/shm --types tmpfs --options=rw,nosuid,nodev
fi

findmnt /proc/sys &> /nul
[ $? -ne 0 ] && \
  mount /proc/sys /proc/sys --options=bind,private,ro

findmnt /proc/sysrq-trigger &> /nul
[ $? -ne 0 ] && \
  mount /proc/sysrq-trigger /proc/sysrq-trigger --options=bind,private,ro

findmnt /sys/block &> /nul
[ $? -ne 0 ] && \
 mount /empty /sys/block --options=bind,private,ro

mount /empty /host --options=bind,private,ro
```

这个脚本如前所说, 就是为容器内的应用准备所需的 /proc, /dev, /sys 这些目录. 从

自己的观察看, 虽然像 /dev 目录, 可以通过 mknod, chown 这些命令来进行构建, 但部

分设备文件, 是不能被程序识别的. 比如 /dev/ptmx, urxvt 会报找不到文件. 同时对于

/sys 的某些目录, 自己参考 systemd-nspawn 的说明, 对某些目录限制其写操作. 把目录

/sys/block 目录被链接到一个空目录. 是不希望容器内的程序看到宿主系统的磁盘.

这也保护临时存放宿主系统根目录的 /host, 必须对这个进行限制.

容器内的脚本 - arch-daemon

```
#!/bin/bash

set -o nounset
export IFS=$'\n'
set -x

[ ${#@} -ne 2 ] && exit 1

declare -r  USER_NAME=$1
declare -r  COMM=$2
declare -r  FIFO='/sandbox/arch-'${COMM}
declare -ri MAX=16
declare -r  EXIT='exit'
declare -r  DISPLAY=':0'

function main()
{
        [[ -e ${FIFO} && ! -p ${FIFO} ]] && return 1

        local value=''

        while [ 0 ];
        do
                value=$(head --bytes=16 ${FIFO})
                if [ ${value} == ${COMM} ]; then
                  s   u       -    ${USER_NAME} --command "/usr/local/bin/start-${C
                elif [ ${value} == ${EXIT} ]; then
                        break
                fi
        done

        return 0
}

if [ ${#@} -ne 2 ]; then
        exit 1
else
  m a i n  $1 $2
        exit 0
fi
```

这个脚本, 通过轮询读取管道文件, 来接收指令, 以运行程序, 或结束脚本运行.

先说说管道文件. 这个管道文件, 需要同时位于宿主系统和容器内, 两个位置可以不一致.

其访问权限, 除了 root 可以访问外, 通过 setfacl 命令设置为, 仅仅宿主系统的指定用

户可以访问. 容器内运行程序的用户是不能访问的.

由于自己考虑是一个实例, 只启动一个应用, 所以管道文件是每个应用有自己对应的文件.

读取指令使用 head 命令的 byte 参数, 来限制读取长度, 不接受过长的字符串. 同时又

由于当管道文件没有内容时, head 命令会自动挂起. 这样巧妙避免了轮询过程对 CPU 的

占用. 当接收到的指令且符合预期, 则通过 su 命令, 以指用户身份运行特定程序, 同时

由于程序运行被设置为后台运行, 则该程序自己没有限制的话, 可以运行多个实例.

调用 control-sandbox-app 的 systemd service

/lib/systemd/system/control-sandbox-app.service

```
[Unit]
Description=Arch: Example
Requires=launch-arch-apps.service
After=launch-arch-apps.service

[Service]
Type=simple
KillMode=control-group

# SystemCallFilter=~umount2
SystemCallFilter=~ptrace

ProtectSystem=strict

NoNewPrivileges=yes
ProtectKernelTunables=yes
ProtectKernelModules=yes

ReadWritePaths=/entry/arch/urxvt/user
ReadWritePaths=/entry/arch/urxvt/tmp
ReadWritePaths=/run/arch.urxvt

RuntimeMaxSec=infinity
ExecStart=/usr/local/bin/ubuntu-pc.bin/control-sandbox-app egret urxvt
```

这个服务简单展示了通过 systemd 管理脚本的方式, 同时该服务也对特定 syscall 进行了限制, 例如 ptrace. 如果在其内执行诸如 strace id, 会显示

Bad system call (core dumped)

这里多说一句, 之前在 systemd-chroot 内希望启动 GNOME Web(epiphany), 但即便确认了 kernel.unprivileged_userns_clone = 1 的前提下, epiphany 依然无法正常运行, 可以看到浏览器界面, 但输入网址, 页面一片空白. 经过排查, 发现是因为将 umount2 过滤掉, 从而导致 epiphany 没法运行.

运行容器应用

自己目前只测试了 urxvt. 运行方式如下

以宿主系统非 root 账户, 执行命令

echo urxvt > /sandbox/arch-urxvt

可以看到, 每执行完成一次命令, urxvt 就启动一个. 这些 urxvt 都共享同一个 /proc, 可以在其中一个 urxvt 里启动 top, 然后在其他 urxvt 都可以看到这个进程, 也可以用 kill 结束其运行. 同时也可以从 top 看到, 里面运行的进程与宿主系统的截然不同. 此外, 执行 lsblk 是看不到, 机器所拥有的磁盘列表.

可通过如下方式, 一次运行 100 个 urxvt 实例.

for i in {0..99}; do echo urxvt > /sandbox/arch-urxvt; sleep .1; done

在使用 pivot_root 实现 chroot 后, Chromium 和 GNOME Web 运行都没有问题了. 不过需要说明的是 Chromium 的 GPU 硬件加速(图形和视频解码)尚未测试, 但执行

cat /dev/dri/card0

cat /dev/dri/renderD128

是没有问题的.

执行

echo exit > /sandbox/arch-urxvt

可使得该服务直接结束, 而不需要通过 systemctl 来进行. 即便是在终端运行脚本 control-sandbox-app, 用上述方式, 也可以快速结束 100 个 urxvt 实例.

存在的缺陷和不足

缺少对 syscall 的过滤, 虽然程序是以普通用户方式运行.

没有 cgroup 支持, 就是说没有限制应用对系统资源的调用, 比如 CPU 使用情况. 不过 cgroup, 可以通过 mkdir 在 /sys/fs/cgroup 创建相关目录, 以生成对应的接口文件.

缺少 SELinux/AppArmor 支持.

由于需要保障 GUI 程序的运行性能, 或者说能够使用 X11 的一些扩展, 所以没有对 IPC 进行隔离.

出于访问网络的便捷性, 网络部分也缺少隔离.

缺少针对多个桌面用户切换/运行的机制. 例如登录到桌面的用户 A 通过此方法运行一个应用后, 如果重新以用户 B 身份登录, 并也想运行应用, 则缺少对应的机制.

编辑于 2021-08-30 22:09

Linux 运维