به نام خداوند بخشنده مهربان

سیستم های عامل

استاد جلیلی

مبینا حیدری

تاریخ گزارش: ۱۰ تیر ۱۴۰۴ **گزارش نهایی پروژه container** شماره دانشجویی: ۴۰۱۱۰۵۸۶۱

۱ شرح مفاهیم اولیه

۱.۱ محفظه (Container)

یک محفظه (Container) واحد نرم افزاری سبک، قابل حمل و خودکفاست که یک برنامه را همراه با تمام وابستگی هایش (کتابخانه ها، فریمورک ها، محیط اجرا، تنظیمات و غیره) در یک بسته اجرایی واحد قرار میدهد. کانتینرها با ایزوله کردن برنامه ها از زیرساخت میزبان، اجرای یکنواخت آنها را در محیط های مختلف تضمین می کنند.

ویژگی های کلیدی محفظه ها:

۱. ایزوله سازی:
کانتینرها در فضای کاربری مجزا اجرا میشوند و تضمین میکنند که برنامه ها و وابستگی هایشان با فرآیندها یا کانتینرهای دیگر روی یک سیستم تداخل نداشته باشند.

۲. قابلیت حمل:
کانتینرها روی هر سیستمی که از زمان اجرای کانتینر مانند Docker پشتیبانی کند، اجرا میشوند (صرف نظر از سیستم عامل یا زیرساخت میزبان).

۳. سبک وزنی:
برخلاف ماشینهای مجازی ،(VM) کانتینرها از هسته (Kernel) سیستم عامل میزبان استفاده میکنند و نیاز به نصب یک OS کامل ندارند. این ویژگی باعث راه اندازی سریعتر و مصرف منابع کمتر میشود.

۴. یکنواختی:
کانتینرها تضمین میکنند که رفتار برنامه در هر محیطی یکسان است .

تفاوت محفظه ها و ماشین های مجازی:
ماشین های مجازی نیاز به یک os کامل برای هر ماشین مجازی دارند در حالی که محفظه ها از هسته ی سیستم عامل میزبان استفاده می کنند.
ماشین های مجازی سنگین ، کند و پرمصرف منابع محفظه ها سبک وزن و سریع در راه اندازی هستند.
برای اجرای چندین os روی یک میزبان مناسب اند در حالی که محفظه ها برای بسته بندی برنامه ها مناسب هستند.

۲.۱ namespace در لینوکس :

namespace ها مکانیزمی در هسته ی لینوکس برای ایزوله سازی منابع بین فرایند ها هستند و به فرایند ها اجازه می دهند تا دید مجزایی از سیستم داشته باشند، طوری که انگار روی یک ماشین اختصاصی در حال اجرا هستند.
انواع اصلی آن ها عبارت اند از :
PID Namespace : فرایند ها در این namespace شناسه های منحصر به فرد خود را دارند و مستقل از میزبان یا namespace های دیگر عمل می کنند.
Network Namespace : جداسازی شبکه (آدرس های IP ، جداول مسیریابی). هر محفظه شبکه مختص به خود را دارد.
Mount Namespace :جداسازی نقطه های اتصال (mount points). فایل سیستم محفظه از میزبان مستقل است.
UTS Namespace : جداسازی hostname و domainname .
User Namespace : جداسازی شناسه های کاربری (UID/GID) . کاربر داخل محفظه میتواند نقش root داشته باشد بدون نیاز به دسترسی root در میزبان.

شکل ۱: PID namespace

۳.۱ cgroup در لینوکس :

برای محدودسازی و نظارت بر منابع استفاده شده توسط فرایندها استفاده میشوند. قابلیت ها شامل:

محدودیت CPU : تعیین سهم پردازنده (مثلاً با cpu.shares).

محدودیت حافظه: تعیین حداکثر حافظه قابل استفاده (با memory.limit_in_bytes).

محدودیت I/O: کنترل دسترسی به دیسک (با blkio.throttle.read_bps_device).
فرض کنید میخواهید حداکثر حافظه یک برنامه را به ۵۰۰ مگابایت محدود کنید:

۱. یک گروه حافظه ایجاد کنید:

شکل ۲: cgroups

۲. محدودیت حافظه را تنظیم کنید:

شکل ۳: cgroups

۳. فرآیند را به این گروه اضافه کنید:

شکل ۴: cgroups

اگر فرآیند از این حد بیشتر مصرف کند، توسط هسته لینوکس متوقف یا خاتمه داده میشود .

۴.۱ chroot در لینوکس :

chroot مخفف (change root) یک فراخوان سیستمی و دستور در لینوکس است که دایرکتوری ریشه ظاهری را برای یک فرآیند و فرآیندهای فرزند آن تغییر میدهد. این قابلیت یک محیط ایزوله ایجاد میکند که فرآیند فقط به فایل ها و دایرکتوری های داخل دایرکتوری ریشه مشخص شده دسترسی دارد و عملاً در یک «زندان» (jail) محدود میشود. chroot یکی از قدیمی ترین روشهای ایزوله سازی سیستم فایل در سیستم های شبه یونیکس است.
تفاوت محفظه و chroot :
محفظه سیستم فایل، فرآیندها، شبکه، کاربران و غیره را ایزوله میکند. chroot فقط سیستم فایل را ایزوله میکند.
محفظه از cgroup برای محدودیت حافظه یا cpu استفاده می کند . chroot محدودیت منابع ندارد.

۵.۱ سیستم های فایل union (unionFS) :

سیستم فایلی که چندین لایه دایرکتوری را ترکیب میکند. برای مثال:
لایه پایه (lowerdir) : محتوای فقط خواندنی.
لایه بالایی (upperdir) : تغییرات نوشتنی.
لایه ادغام شده (merged) : نمای یکپارچه برای کاربر.

شکل ۵: مثالی برای unionFS

فایل های merged/ ترکیبی از lower/ و upper/ خواهند بود

۶.۱ eBPF :

یک فناوری پیشرفته در هسته لینوکس برای اجرای کدهای امن و کارآمد در فضای هسته. کاربردها شامل:
ردیابی فراخوانی های سیستمی (مانند ()clone با فلگ های namespace).
مانیتورینگ عملکرد و ایجاد لاگ برای رویدادهای خاص.

۲ معرفی ابزارها و محیط اجرایی

۱.۲ ابزارهای اصلی

unshare : ایجاد هایnamespace جدید برای فرایند (مثال: unshare –pid –mount-proc).
nsenter : ورود به namespace های موجود.
cgcreate/cgset : مدیریت cgroups (مثال: cgcreate -g cpu،memory:/my_container).
OverlayFS : برای ساخت سیستم فایل ترکیبی (مثال: نمونه کد در قسمت unionFS آمده).
Tools eBPF :
BCC (Collection Compiler BPF) : مجموعه ابزارهای پایتون برای توسعه برنامه های eBPF .
bpftrace : زبان اسکریپت نویسی برای ردیابی رویدادهای هسته.

۲.۲ محیط اجرایی

سیستم عامل: لینوکس .
زبان برنامه نویسی: C/Python برای تعامل با syscall ها و eBPF .
کتابخانه ها:
libbpf : کتابخانه C برای مدیریت eBPF .
rsync : برای کپی کردن فایل سیستم پایه به محفظه.

۳ چالش های پیشرو

هماهنگی بین Namespace ها و Cgroups : اطمینان از اعمال صحیح محدودیت ها روی هر محفظه.
مدیریت Mount های پویا: پشتیبانی از overlayfs و انتشار mount ها بین محفظه ها.
بهینه سازی کارایی: کاهش سربار ناشی از ایزوله سازی و eBPF .

۴ گزارش فاز دوم

۱.۴ نحوه‌ی پیاده‌سازی

در این بخش، توضیح دقیقی از پیاده‌سازی بخش‌های مختلف شامل namespaceها، cgroupها و chroot ارائه می‌دهیم.

۱.۱.۴ Namespace ها

Namespace در لینوکس مکانیزمی است که به هر پردازه اجازه می‌دهد تا نمای محدودی از منابع سیستم را مشاهده کند. هر namespace یک نوع خاص از منابع (مانند PID، شبکه، hostname و ...) را ایزوله می‌کند، به طوری‌که پردازه‌ها فقط منابع محلی‌شده را مشاهده و استفاده می‌کنند.
در این پروژه از شش نوع namespace برای ایزوله‌سازی کامل استفاده شده است:

CLONE_NEWUTS (ایزوله‌سازی نام میزبان) این namespace امکان تغییر نام میزبان کانتینر را فراهم می‌کند، بدون آن‌که بر میزبان تأثیر بگذارد. این قابلیت از طریق فلگ CLONE_NEWUTS فعال شده و با دستور hostname داخل کانتینر تست می‌شود.

CLONE_NEWNS (ایزوله‌سازی فضای mount) این namespace باعث جداسازی point mount ها می‌شود. mount و un-mount کردن در کانتینر روی میزبان تأثیری ندارد.

CLONE_NEWNET (ایزوله‌سازی شبکه) این قابلیت اجازه می‌دهد کانتینر stack شبکه‌ای اختصاصی، آدرس IP و تنظیمات routing جداگانه داشته باشد.

CLONE_NEWPID (ایزوله‌سازی PID) در این حالت پردازه‌ها در کانتینر دارای هایPID جداگانه‌ای هستند که از ۱ شروع می‌شوند. پس از اجرای clone با این فلگ، لازم است fork یا exec انجام شود تا پردازه‌ی init جدید اجرا گردد.

CLONE_NEWIPC (ایزوله‌سازی منابع اشتراکی بین پردازه‌ای) این قابلیت باعث جداسازی منابعی مانند message، shared queue و semaphore می‌شود. در صورت غیرفعال بودن این قابلیت با فلگ share-ipc–، کانتینر منابع مشترک میزبان را مشاهده می‌کند.

CLONE_NEWUSER (ایزوله‌سازی شناسه کاربر) این namespace اجازه می‌دهد یک کاربر عادی بدون دسترسی ریشه، کانتینری با دسترسی root در فضای جداگانه ایجاد کند (container rootless).

نحوه‌ی پیاده‌سازی Namespace User

برای فعال‌سازی این قابلیت از یکی از توابع زیر استفاده می‌شود:

- unshare(CLONE_NEWUSER)
- clone(..., CLONE_NEWUSER | SIGCHLD, ...)

سپس، جهت دسترسی واقعی به root، باید نگاشت شناسه‌های کاربری انجام شود.

فایل‌های نگاشت شناسه (Mapping)

- /proc/[pid]/uid_map
- /proc/[pid]/gid_map

مقداردهی به این فایل‌ها مشخص می‌کند که شناسه‌ی کاربر اصلی سیستم در user namespace جدید چه نقشی دارد.

تحلیل تابع do_start

تابع do_start وظیفه راه‌اندازی کانتینر با استفاده از namespace ها، تنظیمات پردازشی و هاcgroup را دارد.

نقش clone در do_start

تابع clone برای ایجاد یک پردازه فرزند ایزوله شده استفاده می‌شود:

```
pid_t pid = clone(child_func, child_stack + STACK_SIZE, flags, &args);
```

در این خط:

- child_func تابعی است که در پردازه‌ی فرزند اجرا می‌شود.
- flags تعیین‌کننده هایnamespace فعال است، مانند:

- CLONE_NEWPID
- CLONE_NEWNET
- CLONE_NEWNS
- ...

- ساختار args شامل مسیر rootfs و تنظیمات دیگر کانتینر است.

اهمیت clone

اگر از clone استفاده نشود، کانتینر به‌صورت ایزوله اجرا نخواهد شد. ایزوله‌سازی PID ، شبکه، فضای فایل و... تنها از طریق clone با names-pace های مربوطه قابل دستیابی است.

۲.۴ cgroup ها

در این متن به بررسی کامل بخش پیاده‌سازی cgroup v2 در کدی که محدودیت منابع را برای کانتینرها اعمال می‌کند، می‌پردازیم.

تابع setup_cgroup_v2 (pid pid_t)

این تابع وظیفه ایجاد یک cgroup اختصاصی برای کانتینر جدید با شناسه پردازه pid و اعمال محدودیت‌های منابع روی آن را دارد.

ساخت مسیر cgroup

ابتدا مسیر پایه /sys/fs/cgroup/my_runtime ساخته می‌شود (در صورت عدم وجود):

```
mkdir(CGROUP_V2_BASE, 0755);
```

سپس دایرکتوری مخصوص کانتینر با نام simple_container_pid ایجاد می‌شود، مانند:

/sys/fs/cgroup/my_runtime/container_1234

محدود کردن حافظه

با نوشتن مقدار 100MB در فایل memory.max محدودیت حافظه تنظیم می‌شود:

محدود کردن CPU

با نوشتن " 50000 100000 " در فایل cpu.max تعیین می‌شود که کانتینر تنها ۵۰% از CPU را استفاده کند:

محدود کردن سرعت I/O دیسک

محدودیت نرخ خواندن و نوشتن به ۵۰ مگابایت بر ثانیه اعمال می‌شود:

محدود کردن تعداد پردازه‌ها

تعداد پردازه‌ها به حداکثر ۳۲ محدود می‌شود:

اضافه کردن پردازه کانتینر به cgroup

شناسه پردازه pid در فایل cgroup.procs نوشته می‌شود تا کانتینر زیرمجموعه این cgroup قرار بگیرد:

تابع cleanup_cgroup_v2 (pid pid_t)

پس از اتمام کار کانتینر، دایرکتوری cgroup مربوطه حذف می‌شود:

```
rmdir /sys/fs/cgroup/my_runtime/simple_container_ < pid
```

نکات مهم

- این پیاده‌سازی از cgroup v2 استفاده می‌کند که مدیریت منابع یکپارچه‌تری نسبت به نسخه ۱ دارد.
- فایل‌های محدودیت منابع در cgroup v2 متفاوت و کامل‌تر هستند.
- با اضافه کردن پردازه به فایل cgroup.procs، تمامی محدودیت‌ها بر آن و فرزندانش اعمال می‌شود.

chroot ۳.۴

در این سند، یک اسکریپت Bash بررسی می‌شود که با هدف ساخت یک محیط ایزوله‌شده لینوکسی طراحی شده است. این محیط با استفاده از ابزار chroot ایجاد می‌شود و امکان اجرای برنامه‌ها در یک فضای محدود و جدا از سیستم میزبان را فراهم می‌کند.

بخش اول: ساختن محیط rootfs

هدف از این مرحله، ایجاد ساختار پایه‌ای فایل‌سیستم لینوکس است که درون آن، برنامه‌هایی مانند bash، ls، و کتابخانه‌های موردنیاز آن‌ها قرار دارند.

ساخت دایرکتوری‌های پایه

ابتدا دایرکتوری‌هایی مانند bin/، lib/، usr/bin/، proc/، sys/، و dev/ در مسیر دلخواه (مثلاً /tmp/my_container_rootfs) ساخته می‌شوند. این ساختار شبیه به یک سیستم لینوکس واقعی است.

کپی باینری‌های ضروری

در این مرحله، فایل‌های اجرایی حیاتی مانند bash، ls، sh، cat، و hostname به دایرکتوری‌های مربوطه کپی می‌شوند. این فایل‌ها برای اجرای دستورات پایه در محیط ایزوله‌شده ضروری هستند.

کپی کتابخانه‌های موردنیاز

هر یک از برنامه‌های بالا برای اجرا به کتابخانه‌هایی (libraries shared) نیاز دارند. با استفاده از دستور ldd می‌توان این کتابخانه‌ها را شناسایی کرده و سپس در مسیر درست مانند lib/x86_64-linux-gnu/ یا lib64/ کپی کرد.

افزودن برنامه دلخواه

می‌توان یک برنامه دلخواه (مانند myprogram) را به محیط rootfs کپی کرد و آن را با chmod +x قابل اجرا نمود تا در محیط chroot تست شود.

بخش دوم: mount کردن فایل‌سیستم‌ها

برای عملکرد صحیح برخی برنامه‌ها، لازم است فایل‌سیستم‌های مجازی مانند dev/، proc/، و sys/ به محیط rootfs bind-mount شوند.

- dev/ برای دسترسی به دستگاه‌ها (مانند dev/null/ یا dev/tty/) مورد نیاز است.
- proc/ برای خواندن اطلاعات سیستم (مانند proc/cpuinfo/) ضروری است.
- sys/ برای اطلاعات مربوط به سخت‌افزار استفاده می‌شود.

بخش سوم: ورود به محیط با chroot

با اجرای دستور:

```
sudo chroot /tmp/my_container_rootfs /bin/bash
```

شما وارد یک محیط ایزوله می‌شوید که در آن ریشه فایل‌سیستم همان دایرکتوری rootfs است. تمامی دستورات مانند ls، hostname، و برنامه‌های سفارشی فقط در این محیط اجرا می‌شوند.
نکته: chroot فقط فایل‌سیستم را ایزوله می‌کند و همچنان پردازه‌ها، شبکه و کاربران با سیستم میزبان مشترک هستند.

بخش چهارم: خروج و پاک‌سازی

پس از خروج از محیط chroot با دستور exit، اسکریپت هایmount انجام‌شده را به صورت معکوس پاک می‌کند:

```
umount /tmp/my_container_rootfs/sys
umount /tmp/my_container_rootfs/proc
umount /tmp/my_container_rootfs/dev
```

این مرحله از باقی‌ماندن منابع استفاده‌شده و هایmount بلااستفاده جلوگیری می‌کند.
تابع chroot یکی از ابزارهای مهم در ایجاد محیط ایزوله برای پردازه‌ها است و در این کد، نقش آن محدود کردن فضای فایل‌ها برای پردازهٔ فرزند (کانتینر) است.
در تابع child_func، ابتدا سیستم فایل جدیدی بر پایهٔ overlayfs در مسیر mnt/ سوار می‌شود. سپس، با استفاده از دستور زیر:

```
if (chroot("/mnt") == -1 || chdir("/") == -1) {
    perror("[child] chroot or chdir");
    return -1;
}
```

مراحل زیر انجام می‌شود:

- **chroot("/mnt")**: این فراخوانی باعث می‌شود مسیر /mnt به عنوان ریشهٔ جدید فضای فایل‌ها برای پردازه در نظر گرفته شود. از این لحظه، تمام مسیرهای مطلق (/) برای پردازه نسبت به مسیر /mnt تفسیر می‌شوند.
- **chdir("/")**: پس از تغییر ریشه، پردازه همچنان ممکن است در پوشه‌ای خارج از ریشهٔ جدید باشد. این دستور باعث می‌شود دایرکتوری جاری به ریشهٔ جدید تنظیم شود و از خطاهای احتمالی جلوگیری شود.

این ترکیب باعث می‌شود پردازهٔ فرزند (کانتینر) نتواند به فایل‌های میزبان اصلی دسترسی داشته باشد و فقط محیط محدودشده‌ای که در مسیر /mnt فراهم شده است، در اختیارش باشد.
در نتیجه، chroot یکی از ابزارهای کلیدی برای پیاده‌سازی امنیت و ایزولاسیون در سطح سیستم فایل است که همراه با namespaceها و cgroupها، کانتینر را از سیستم میزبان جدا می‌کند.

توضیح درباره‌ی ترکیب `rootfs` و `chroot`

در بسیاری از اسکریپت‌ها و برنامه‌های مرتبط با محیط‌های کانتینری و جعبه‌های ایزوله‌شده، از مفهوم `rootfs` و دستور `chroot` استفاده می‌شود.
در این توضیح، به این سؤال پاسخ داده می‌شود که چگونه ممکن است ریشه سیستم فایل (rootfs) در مسیر `/tmp` قرار داشته باشد، اما دستور `chroot` روی مسیر `/mnt` اجرا شود و هیچ تناقضی ایجاد نکند.

خلاصه

مسیر `/mnt` یک نقطه‌ی مونت (mount point) است که با استفاده از `overlayfs` روی آن یک سیستم فایل مجازی شامل `rootfs` قرار داده شده است. بنابراین، اجرای `chroot /mnt` به معنای ورود به همان سیستم فایل ریشه‌ای است که در واقع از `rootfs` موجود در `/tmp` مشتق شده است.

شرح کامل

۱. در بخش راه‌اندازی، یک سیستم فایل `overlayfs` روی مسیر `/mnt` سوار (mount) می‌شود. این سیستم فایل از چند دایرکتوری تشکیل شده که یکی از آن‌ها `rootfs` است که در مسیر `/tmp` واقع شده است.

۲. در واقع گزینه‌ی `lowerdir` در `overlayfs` به مسیر `rootfs` اشاره دارد، یعنی `/tmp/my_container_rootfs` یا مسیری مشابه.

۳. با اجرای دستور `chroot /mnt`، ریشه‌ی سیستم فایل برای فرایند جاری به مسیر `/mnt` که حاوی سیستم فایل ترکیبی `overlayfs` است، تغییر داده می‌شود.

۴. این یعنی فرایند به جای دیدن ساختار فایل‌سیستم اصلی سیستم، ساختار فایل‌سیستم موجود در `/mnt` را به عنوان ریشه می‌بیند. چون `/mnt` شامل محتوای `rootfs` است، عملاً فرایند وارد محیطی می‌شود که در اصل همان `rootfs` در `/tmp` است.

تصویر ذهنی

فرض کنید:

- `/tmp/my_container_rootfs` شامل ساختار کامل فایل‌سیستم مورد نیاز است (دایرکتوری‌هایی مانند `bin`، `lib`، `etc` و ...).
- با استفاده از `overlayfs` این ساختار روی `/mnt` سوار شده است.
- پس `/mnt` نمایانگر یک سیستم فایل ترکیبی است که شامل محتویات `rootfs` است.

در نتیجه:

`chroot /mnt` یعنی ورود به ریشه‌ای که همان `rootfs` ما در `/tmp` است، ولی از طریق نقطه مونت `/mnt`.

۴.۴ OverlayFS

در این کد، برای ایزوله‌سازی سیستم فایل ریشه کانتینر، از **OverlayFS** استفاده شده است.

OverlayFS چیست؟

OverlayFS یک نوع فایل سیستم لایه‌ای است که اجازه می‌دهد دو دایرکتوری (یا لایه) با هم ترکیب شوند:

- **Lowerdir**: دایرکتوری فقط خواندنی که معمولا شامل فایل‌های پایه است (در این کد، مسیر `rootfs_path` است).
- **Upperdir**: دایرکتوری قابل نوشتن که تغییرات روی آن اعمال می‌شود
  (در این کد، دایرکتوری موقتی ایجاد شده در `/tmp/container_{pid}_upper`).
- **Workdir**: دایرکتوری کمکی که OverlayFS برای مدیریت داده‌ها به آن نیاز دارد (در این کد، `/tmp/container_{pid}_work`).

این ساختار باعث می‌شود که کانتینر بتواند تغییرات خود را فقط در لایه `upperdir` ثبت کند بدون اینکه لایه اصلی (lowerdir) تغییر کند.

مراحل تنظیم OverlayFS در کد

۱. ابتدا دایرکتوری‌های `upperdir` و `workdir` با استفاده از `mkdir` ایجاد می‌شوند.

۲. سپس با استفاده از تابع `mount`، فایل سیستم `overlay` با گزینه‌های زیر در مسیر `/mnt` مونت می‌شود:

```
workdir=workdir ,upperdir=upperdir ,lowerdir=rootfs_path
```

۳. پس از آن، تنظیمات `mount propagation` برای `/mnt` به صورت `shared` انجام می‌شود تا تغییرات در این مسیر به سایر namespace ها منتقل شود.

همچنین با `chdir("/")` پس از `chroot` مسیر کاری فعلی به ریشه جدید تنظیم می‌شود.
با استفاده از OverlayFS در مسیر `/mnt`، محیط ایزوله شده برای کانتینر فراهم می‌شود.
این روش به صورت کارآمد و سبک به کانتینر اجازه می‌دهد که محیطی مجزا از سیستم اصلی داشته باشد و تغییرات فقط در لایه قابل نوشتن ذخیره شوند.

فضای نام IPC چیست؟

IPC مخفف Communication Inter-Process به معنای «ارتباط بین فرآیندی» است. در سیستم‌عامل لینوکس، فرآیندها می‌توانند از طریق مکانیزم‌هایی مثل `memory shared`، `semaphores` و `queues message` با یکدیگر ارتباط برقرار کنند.
IPC Namespace باعث ایزوله شدن این منابع بین کانتینر و میزبان می‌شود.

نقش پرچم `share-ipc--` در کد

در تابع `do_start` بخشی از کد بررسی می‌کند که آیا پرچم `share-ipc--` به برنامه داده شده است یا خیر
اگر `share-ipc--` فعال باشد، فضای نام IPC جدید ایجاد نمی‌شود و کانتینر IPC سیستم میزبان را به اشتراک می‌گذارد. در غیر این صورت، این فضای نام به طور کامل ایزوله خواهد شد:

کاربردهای عملی

- **بدون `share-ipc--`**: کانتینر نمی‌تواند به حافظه‌های مشترک یا semaphores سیستم میزبان دسترسی داشته باشد. مناسب برای امنیت بیشتر و ایزوله‌سازی کامل.
- **با `share-ipc--`**: کانتینر می‌تواند به IPC میزبان دسترسی داشته باشد. این برای بعضی از سناریوها مانند اشتراک‌گذاری حافظه با میزبان یا بین چند کانتینر مناسب است.

نتیجه‌گیری

پرچم `share-ipc--` یک امکان انعطاف‌پذیر به برنامه اضافه می‌کند تا کاربر در صورت نیاز، ایزوله‌سازی فضای نام IPC را غیرفعال کند. این ویژگی باید با دقت استفاده شود چرا که اشتراک‌گذاری منابع IPC ممکن است به امنیت یا حریم خصوصی فرآیندها آسیب بزند.

۵ شرح جامع توابع سیستم کانتینر

در این بخش، توابع اصلی مورد استفاده در پیاده‌سازی یک سیستم کانتینر سبک با استفاده از فضای نام (namespaces) و فایل‌سیستم overlay به صورت مفصل تشریح شده‌اند. هر تابع به همراه پارامترها، فرآیند اجرا، اهمیت و کاربردهای آن بیان شده است.

۱.۵ تابع drop_capabilities

```
; () drop_capabilities void
```

شرح: در سیستم‌های لینوکسی، قابلیت‌ها (Capabilities) مجموعه‌ای از مجوزهای امنیتی جزئی هستند که به جای دادن دسترسی root کامل، به پروسس‌ها امکان انجام عملیات خاصی داده می‌شود. این تابع برای کاهش سطح دسترسی‌های پروسس کانتینر به کار می‌رود.

نحوه عملکرد: این تابع با فراخوانی‌های مناسب (مثلا با استفاده از `prctl` یا `capset`)، تمامی قابلیت‌های اضافی و غیرضروری را حذف می‌کند. هدف اصلی کاهش سطح دسترسی است تا در صورت نفوذ به کانتینر، آسیب به سیستم میزبان حداقل شود.

اهمیت و کاربرد: حذف قابلیت‌های غیرضروری یک اصل مهم در امنیت کانتینرهاست. این کار خطرات سوءاستفاده از کانتینر برای انجام حملات escalation privilege را به حداقل می‌رساند و به افزایش ایمنی محیط اجرای کانتینر کمک می‌کند.

۲.۵ تابع mount_proc

شرح: مونت کردن فایل‌سیستم مجازی proc در مسیر proc/ به منظور فراهم‌سازی اطلاعات کرنل و پردازش‌ها در فضای ایزوله‌شده‌ی کانتینر.

```
static int mount_proc(void) {
    return mount("proc", "/proc", "proc", 0, "") == -1 ?
           (perror("mount /proc"), -1) : 0;
}
```

کد تابع:

توضیح گام به گام:

- فایل‌سیستم proc یک سیستم فایل مجازی است که اطلاعات زنده‌ی مربوط به پردازش‌ها و وضعیت کرنل را فراهم می‌کند.
- با استفاده از mount call system، این فایل‌سیستم روی مسیر proc/ مونت می‌شود.
- اگر مونت کردن با خطا مواجه شود، از تابع perror برای چاپ پیام خطا استفاده می‌شود و مقدار 1- بازگردانده می‌شود.
- در صورت موفقیت، مقدار 0 بازگردانده می‌شود.

اهمیت و کاربرد :

بدون وجود proc/، ابزارهایی مانند ps و top و بسیاری از توابع POSIX قادر به دسترسی به اطلاعات فرآیندها نیستند. برای هر فضای PID namespace جدا، این فایل‌سیستم باید مجدد مونت شود.

۳.۵ تابع setup_overlayfs

شرح: ایجاد یک فایل‌سیستم قابل نوشتن بر اساس یک filesystem root فقط‌خواندنی با استفاده از OverlayFS و مونت کردن آن روی mnt/ برای استفاده در کانتینر.

کد تابع:

```
static int setup_overlayfs(const char *rootfs_path) {
    char upperdir[256], workdir[256], mount_opts[1024];
    snprintf(upperdir, sizeof(upperdir), "/tmp/container_%d_upper", getpid());
    snprintf(workdir, sizeof(workdir), "/tmp/container_%d_work", getpid());

    mkdir(upperdir, 0755);
    mkdir(workdir, 0755);

    snprintf(mount_opts, sizeof(mount_opts),
             "lowerdir=%s,upperdir=%s,workdir=%s",
             rootfs_path, upperdir, workdir);
```

```
    if (mount("overlay", "/mnt", "overlay", 0, mount_opts) == -1) {
        perror("mount overlay");
        return -1;
    }

    if (mount(NULL, "/mnt", NULL, MS_SHARED | MS_REC, NULL) == -1) {
        perror("mount propagation for /mnt");
        return -1;
    }

    return 0;
}
```

توضیح گام به گام:

۱. ابتدا دو مسیر برای دایرکتوری‌های upperdir و workdir با استفاده از شناسه پردازش (PID) ساخته می‌شود تا از تداخل میان کانتینرها جلوگیری شود.

۲. این مسیرها با سطح دسترسی 0755 ایجاد می‌شوند.

۳. گزینه‌های مونت (مانند lowerdir، upperdir و workdir) در قالب یک رشته ساخته می‌شوند.

۴. فایل‌سیستم overlay روی مسیر /mnt سوار می‌شود. این مسیر بعداً به عنوان root کانتینر استفاده خواهد شد.

۵. سپس، propagation اشتراکی با استفاده از MS_SHARED | MS_REC برای مسیر /mnt فعال می‌شود. این کار اطمینان حاصل می‌کند که هایی‌mount که در این مسیر انجام می‌شوند، قابل مشاهده در namespace descendant ها نیز هستند.

مفاهیم کلیدی:

- **OverlayFS:** یک فایل‌سیستم union است که اجازه می‌دهد چند لایه‌ی فایل‌سیستم با هم ترکیب شوند. لایه‌ی پایین (lowerdir) فقط‌خواندنی و لایه‌ی بالا (upperdir) قابل نوشتن است.
- **workdir:** دایرکتوری کمکی مورد نیاز برای عملکرد .OverlayFS
- **mount propagation:** مشخص می‌کند تغییرات در هاmount چگونه بین هاnamespace گسترش یابد. با MS_SHARED، هایmount جدید به namespace descendant منتقل می‌شوند.

۴.۵ تابع save_metadata

شرح: تابع save_metadata یکی از توابع مهم در فرایند اجرای کانتینر است که وظیفه آن ذخیره‌سازی اطلاعات محیطی یک کانتینر به‌منظور استفاده‌های بعدی از جمله بازیابی ،(restore) مانیتورینگ، یا دیباگ است.

کد تابع:

```
static void save_metadata(pid_t pid, const struct child_args *args) {
mkdir(METADATA_DIR, 0755);

char path[256];
snprintf(path, sizeof(path), "%s/%d.meta", METADATA_DIR, pid);
FILE *f = fopen(path, "w");
if (!f) return;

fprintf(f, "pid=%d\n", pid);
fprintf(f, "rootfs=%s\n", args->rootfs_path);
fprintf(f, "memory_limit=100MB\n");
fprintf(f, "cpu_quota=50%%\n");
fprintf(f, "overlay_upper=/tmp/container_%d_upper\n", pid);
fprintf(f, "overlay_work=/tmp/container_%d_work\n", pid);
fprintf(f, "mount_propagation=shared\n");

fclose(f);

}
```

توضیح گام به گام

- ; (METADATA_DIR, 0755)mkdir این دستور اطمینان حاصل می‌کند که دایرکتوری METADATA_DIR (مثلاً /var/lib/containers/meta) برای ذخیره فایل متادیتا وجود دارد. سطح دسترسی 0755 به مالک اجازه کامل و به دیگران اجازه فقط خواندن و اجرا را می‌دهد.
-
- ;("w", path)FILE *f = fopen فایل متادیتا را در حالت نوشتن باز می‌کند. در صورت خطا (عدم دسترسی یا وجود نداشتن دایرکتوری والد)، تابع خاتمه می‌یابد.
- (...)fprintf این دستورات داده‌های زیر را در فایل می‌نویسند:
  - شناسه پردازش (pid)
  - مسیر ریشه فایل‌سیستم (rootfs)
  - محدودیت حافظه (به‌صورت ثابت 100MB)
  - سهمیه CPU (به‌صورت ثابت
  - ; (f)fclose فایل را پس از نوشتن می‌بندد تا منابع سیستم آزاد و داده‌ها به‌درستی ذخیره شوند.

۵.۵ تابع cleanup_overlayfs

شرح: تابع cleanup_overlayfs به منظور پاک‌سازی فایل‌سیستم Overlay کانتینر طراحی شده‌است. این تابع وظیفه unmount کردن مسیر overlay و حذف دایرکتوری‌های موقتی ایجادشده را بر عهده دارد.

توضیح گام به گام:

۱. umount2("/mnt", MNT_DETACH): این فراخوانی call system مربوط به unmount کردن دایرکتوری /mnt است که فایل‌سیستم overlay روی آن mount شده بود. فلگ MNT_DETACH به سیستم می‌گوید که point mount را جدا کند بدون آنکه بلافاصله دسترسی فایل‌ها قطع شود؛ این برای ایمنی بیشتر و جلوگیری از کرش فرایندهای دیگر مفید است.

۲. سپس، دو رشته‌ی کاراکتری upperdir و workdir تعریف می‌شوند تا مسیرهای مربوط به دایرکتوری‌های کاری و بالایی overlay ساخته شوند. این مسیرها معمولاً به شکل:
   - /tmp/container_<pid>_upper
   - /tmp/container_<pid>_work

   هستند که با استفاده از ()getpid شناسه فرایند جاری به صورت پویا تولید می‌شود.

۳. rmdir(upperdir); و rmdir(workdir);: این دو فراخوانی، دایرکتوری‌های بالا و کاری که در هنگام راه‌اندازی overlay با تابع setup_overlayfs ساخته شده بودند را حذف می‌کنند.

نکات تکمیلی

- استفاده از MNT_DETACH باعث می‌شود که point mount به صورت "lazy" جدا شود. این یعنی تا زمانی که دیگر هیچ فرایندی به آن فایل‌سیستم دسترسی نداشته باشد، منابع آزاد نخواهند شد.

۶.۵ تابع child_func

```
int child_func ( void *args ) ;
```

شرح: این تابع نقطه شروع اجرای پروسس جدیدی است که توسط clone() ساخته شده و قرار است کانتینر را راه‌اندازی کند. تمام تنظیمات محیط کانتینر در این تابع انجام می‌شود.

توضیح گام به گام:

۱. **راه‌اندازی فایل‌سیستم overlay :** با فراخوانی `mount_overlay`، فضای فایل‌سیستم ایزوله شده برای کانتینر ساخته می‌شود.
۲. **تنظیم فضای نام روت:** با استفاده از `chroot` و تغییر دایرکتوری کاری به mountpoint، محیط ایزوله کانتینر ساخته می‌شود.
۳. **مونت کردن :proc/** با فراخوانی `setup_proc`، سیستم فایل proc به درستی در کانتینر سوار می‌شود.
۴. **حذف قابلیت‌های اضافی:** تابع `drop_capabilities` اجرا شده تا سطح دسترسی محدود شود.
۵. **اجرای شل یا برنامه اصلی کانتینر:** در نهایت، یک شل تعاملی مانند `/bin/bash` یا برنامه تعریف شده توسط کاربر اجرا می‌شود تا تعامل با کانتینر ممکن شود.

بازگشت تابع: این تابع در پایان کار کانتینر با مقدار مناسب (معمولاً ۰) خاتمه می‌یابد.

اهمیت و کاربرد: این تابع، به عنوان مغز اصلی راه‌اندازی کانتینر عمل می‌کند و محیط کاملاً ایزوله و امن برای اجرای برنامه‌ها را فراهم می‌آورد.

۷.۵ تابع save_pid

```
; ) pid pid_t ( save_pid void
```

شرح: این تابع برای ذخیره شناسه فرایند کانتینر در یک فایل مشخص استفاده می‌شود. ذخیره PID برای مدیریت کانتینرهای در حال اجرا (مثلاً ارسال سیگنال برای توقف یا مشاهده وضعیت) ضروری است.

نحوه عملکرد: فایل مشخصی (مانند `/var/run/mycontainer.pid`) باز شده و شناسه فرایند به صورت رشته‌ای در آن نوشته می‌شود. در صورت خطا، می‌توان پیام‌های مناسب لاگ کرد.

اهمیت و کاربرد: بدون ذخیره PID، مدیریت چرخه عمر کانتینر سخت و پیچیده می‌شود. این عمل به ابزاری برای مانیتورینگ و کنترل کانتینر امکان می‌دهد کارآمد و مطمئن عمل کند.

۸.۵ تابع remove_pid

```
; ) pid pid_t ( remove_pid void
```

شرح: این تابع پس از پایان کار کانتینر، فایل PID ذخیره شده را حذف می‌کند تا نشان دهد کانتینر دیگر فعال نیست.

نحوه عملکرد: با حذف فایل PID، دیگر ابزاری نمی‌تواند به اشتباه کانتینر را فعال فرض کند. این کار موجب پاکیزگی و نظم بیشتر مدیریت کانتینرها می‌شود.

۹.۵ تابع list_pids

شرح: تابع `list_pids` برای نمایش شناسه پردازه (PID) کانتینرهایی که در حال حاضر در حال اجرا هستند، به‌کار می‌رود. این تابع با استفاده از فایل `PID_STORE_FILE` (که معمولاً در هنگام اجرای هر کانتینر، PID آن در این فایل ذخیره شده است) عمل می‌کند.

توضیح گام به گام:

۱. ابتدا فایل ذخیره‌ی هاPID (که مسیر آن با PID_STORE_FILE مشخص می‌شود) باز می‌شود اگر این فایل وجود نداشته باشد یا قابل باز شدن نباشد، پیامی مبنی بر عدم وجود کانتینر در حال اجرا چاپ شده و تابع پایان می‌یابد.

۲. اگر فایل به‌درستی باز شود، پیامی برای اعلان کانتینرهای فعال چاپ می‌شود:

۳. سپس، محتوای فایل به‌صورت خط‌به‌خط خوانده می‌شود و در هر خط که یک PID ذخیره شده، پردازش انجام می‌گیرد:

- تابع fgets یک خط از فایل می‌خواند.
- با استفاده از atoi، مقدار عددی PID استخراج می‌شود.
- سپس با استفاده از فراخوانی سیستمی kill(pid, 0) بررسی می‌شود که آیا پردازه هنوز فعال است یا نه:
  * اگر پردازه وجود داشته باشد، مقدار برگشتی برابر صفر است و در این صورت، آن PID چاپ می‌شود.
  * اگر پردازه فعال نباشد (یعنی مرده باشد یا قبلاً بسته شده باشد)، چیزی چاپ نمی‌شود.

۴. در پایان، فایل بسته می‌شود

نکات مهم

- استفاده از kill(pid, 0) یک تکنیک رایج و امن برای بررسی زنده بودن پردازه‌هاست بدون اینکه سیگنالی واقعی ارسال شود.
- برای جلوگیری از چاپ هایPID مرده، شرط kill(pid, 0) == 0 وجود دارد که اطمینان حاصل می‌کند فقط پردازه‌های زنده نمایش داده شوند.

۱۰.۵ توابع setup_cgroup_v2 و cleanup_cgroup_v2

شرح: این توابع برای مدیریت کنترل منابع کانتینرها در لینوکس با استفاده از Control Groups Version 2 طراحی شده‌اند. هر کانتینر با شناسه پردازه خود در یک دایرکتوری جداگانه در مسیر /sys/fs/cgroup/ نگهداری می‌شود و محدودیت‌هایی نظیر حافظه، پردازنده، تعداد پردازه‌ها و I/O بر آن اعمال می‌گردد.

تابع setup_cgroup_v2

این تابع با دریافت PID کانتینر، اقدامات زیر را انجام می‌دهد:

- ساخت مسیر /sys/fs/cgroup/simple_container_<pid> برای هر کانتینر
- اعمال محدودیت‌های:
  * حافظه: memory.max = 100MB
  * Swap : غیر فعال (memory.swap.max = 0)
  * CPU : محدود به ۵۰٪ از یک هسته (cpu.max = 50000 100000)
  * دیسک: سرعت خواندن/نوشتن محدود به ۵۰MBps
  * تعداد پردازه‌ها: حداکثر ۳۲ عدد (pids.max = 32)
- افزودن PID پردازه به فایل cgroup.procs جهت فعال‌سازی محدودیت‌ها

تابع cleanup_cgroup_v2

این تابع مسیر ساخته‌شده برای CGroup مربوط به کانتینر را پاک‌سازی می‌کند. اگر هنوز پردازه‌ای در این گروه باقی مانده باشد، عملیات rmdir شکست خواهد خورد.

نکات مهم

- این توابع تنها در سیستم‌هایی با پشتیبانی از CGroup v2 به‌درستی عمل می‌کنند.
- اعمال محدودیت‌ها با استفاده از فایل‌سیستم مجازی cgroupfs انجام می‌شود.
- مدیریت منابع به‌صورت مرکزی و با امنیت بالا صورت می‌گیرد.

۱۱.۵ تابع setup_user_namespace

شرح: این تابع وظیفه‌ی تنظیم نگاشت‌های کاربر و گروه در فضای user namespace را بر عهده دارد. با این کار، یک کاربر عادی می‌تواند درون کانتینر به عنوان کاربر root فعالیت کند، بدون آنکه دسترسی root در سیستم میزبان داشته باشد.

توضیح گام به گام:

۱. **غیرفعال‌سازی setgroups :** قبل از تنظیم gid_map، باید setgroups را با مقدار deny غیرفعال کرد تا اجازه نوشتن در آن فایل داده شود.
۲. **نگاشت UID :**
   - فایل /proc/<pid>/uid_map باز می‌شود.
   - این بدان معناست که UID صفر (root) در کانتینر، معادل UID فعلی در میزبان است.
۳. **نگاشت GID :**
   - مشابه مورد قبل، فایل /proc/<pid>/gid_map تنظیم می‌شود.

اهمیت: این کار موجب می‌شود پردازه‌ی داخل کانتینر دارای دسترسی root به نظر برسد، در حالی که از دید کرنل، همچنان یک کاربر عادی است و از مزایای امنیتی فضای ایزوله بهره‌مند می‌شود.

۱۲.۵ تابع do_start

شرح: این تابع مسئول ایجاد و اجرای یک کانتینر ایزوله‌شده در لینوکس است. ایزولاسیون شامل فضای نام‌ها (namespaces) ، تخصیص CPU ، زمان‌بندی ، cgroup ها و فضای فایل‌سیستمی است.

توضیح گام به گام:

۱. بررسی آرگومان share-ipc– برای استفاده یا عدم استفاده از فضای IPC ایزوله.
۲. بررسی صحت آرگومان‌ها و استخراج مسیر rootfs.
۳. تنظیم فضای ایزوله با هایnamespace : UTS PID Mount Network IPC.
۴. ایجاد پردازه جدید با استفاده از clone.
۵. تخصیص یک CPU خاص برای اجرای کانتینر با استفاده از sched_setaffinity.
۶. تنظیم سیاست زمان‌بندی SCHED_RR با اولویت خاص.
۷. پیکربندی فضای نام کاربر User) (Namespace برای فراهم‌کردن دسترسی root داخل کانتینر بدون خطر امنیتی.
۸. ذخیره PID و اطلاعات متادیتای کانتینر.
۹. محدودسازی منابع با استفاده از cgroups v2.
۱۰. انتظار برای پایان پردازه کانتینر، و سپس آزادسازی منابع.

نتیجه: این تابع به صورت کامل یک کانتینر سبک لینوکسی را ایجاد و مدیریت می‌کند، و پس از پایان اجرای کانتینر، منابع تخصیص‌داده‌شده را آزاد می‌سازد.

۱۳.۵ توابع مدیریتی کانتینر CLI

۱. تابع do_status

هدف این تابع بررسی وضعیت اجرای کانتینر با استفاده از PID است. اگر پردازه‌ی مربوط به PID داده‌شده فعال باشد، پیام "در حال اجرا" چاپ می‌شود، در غیر این صورت پیام "متوقف شده" نمایش می‌یابد.

۲. تابع do_stop

این تابع کانتینر را با سیگنال SIGKILL متوقف کرده و سپس منابع تخصیص داده‌شده به آن شامل cgroup و اطلاعات متادیتا را پاک‌سازی می‌کند.

۳. تابع do_inspect

با استفاده از این تابع می‌توان فایل متادیتای کانتینر را خواند و تنظیمات پیکربندی آن شامل rootfs، محدودیت‌های حافظه و پردازنده، و تنظیمات overlay را مشاهده کرد.

۴. تابع do_freeze

این تابع کانتینر را به صورت موقت فریز می‌کند، به این معنا که اجرای پردازه‌های آن متوقف شده ولی حذف نمی‌شوند. این کار از طریق نوشتن عدد ۱ در فایل cgroup.freeze انجام می‌گیرد.

۵. تابع do_thaw

کانتینری که قبلاً فریز شده است با استفاده از این تابع دوباره به اجرای طبیعی بازمی‌گردد. با نوشتن عدد ۰ در cgroup.freeze، اجرای پردازه‌ها از سر گرفته می‌شود.

۶. تابع do_rm

این تابع حذف کامل یک کانتینر را به عهده دارد، چه در حال اجرا باشد یا نه. ابتدا پردازه‌ی مربوطه را می‌کشد (در صورت زنده بودن) و سپس منابع مربوطه شامل cgroup، متادیتا و PID ذخیره‌شده را پاک می‌کند.

۱۴.۵ تابع mount_shared_device

شرح: تابع mount_shared_device وظیفه دارد تا یک مسیر از سیستم میزبان را در فضای mount namespace یک کانتینر خاص (که با PID مشخص می‌شود) به‌صورت bind mount، در مسیر دلخواه داخل کانتینر، مونت کند. این کار اجازه می‌دهد تا داده‌ها یا دیوایس‌ها بین میزبان و کانتینر به اشتراک گذاشته شوند.

ورودی‌ها:

- pid: شناسه پردازه‌ای که کانتینر را اجرا می‌کند.
- src: مسیر منبع روی سیستم میزبان که قرار است مونت شود (مثلاً /data/shared).
- target_in_container: مسیر مقصد داخل کانتینر (مثلاً /shared).

توضیح گام به گام:

۱. ابتدا مسیر فایل فضای mount namespace کانتینر ساخته و باز می‌شود:

```
/proc/<pid>/ns/mnt
```

۲. سپس فضای namespace فعلی پردازه ذخیره می‌شود تا بعداً بازگردیم.

۳. با استفاده از تابع setns، وارد فضای mount کانتینر می‌شویم.

۴. مسیر مقصد نهایی داخل کانتینر محاسبه می‌شود. فرض بر این است که فایل‌سیستم کانتینر در مسیر /mnt مونت شده است. مسیر نهایی به صورت زیر خواهد بود:

```
/mnt + target_in_container
```

۵. مسیر مقصد داخل کانتینر در صورت نیاز ایجاد می‌شود.

۶. عملیات bind mount انجام می‌گیرد تا مسیر منبع روی میزبان به مسیر مقصد داخل کانتینر متصل شود:

```
mount(src, container_path, NULL, MS_BIND, NULL);
```

۷. در پایان، با استفاده از فایل ذخیره‌شده، فضای mount اصلی میزبان بازیابی می‌شود.

موارد استفاده این تابع معمولاً برای اشتراک‌گذاری فایل‌ها یا ابزارهای سخت‌افزاری بین میزبان و کانتینر استفاده می‌شود؛ برای مثال:

- اشتراک‌گذاری پوشه‌ای حاوی داده با کانتینر.
- در دسترس قرار دادن دیسک یا USB داخل کانتینر.
- مانیتور کردن لاگ‌های میزبان توسط کانتینر.

نکات مهم

- برای استفاده از setns و mount باید دسترسی root یا توانایی‌های CAP_SYS_ADMIN داشته باشید.
- مسیر mnt/ باید از قبل ریشه فایل‌سیستم کانتینر باشد.
- در صورت خطا در هر مرحله، فایل‌های namespace بسته می‌شوند و فضای اصلی بازیابی می‌گردد.

۱۵.۵ تابع main

این تابع نقطه شروع برنامه است که وظیفه دریافت دستورات کاربر و فراخوانی توابع مربوط به هر دستور را بر عهده دارد. علاوه بر این، یک پردازه فرزند جهت اجرای اسکریپت monitor.py برای نظارت بر فراخوانی‌های سیستمی نیز ایجاد می‌کند.

شرح:

۱. ابتدا با استفاده از تابع fork یک پردازه فرزند ایجاد می‌کند:

- در صورت موفقیت، پردازه فرزند با اجرای execlp اسکریپت monitor.py را به عنوان یک پردازه جداگانه اجرا می‌کند تا به صورت همزمان فراخوانی‌های سیستمی کانتینر را مانیتور کند.
- اگر اجرای اسکریپت با خطا مواجه شود، پیام خطا چاپ شده و پردازه فرزند با کد خروجی ۱ خاتمه می‌یابد.

۲. سپس کنترل به پردازه پدر بازمی‌گردد و بررسی می‌کند که آرگومان‌های ورودی به برنامه حداقل دو مقدار داشته باشند (نام برنامه و دستور).

۳. در صورت نبود دستور مناسب، راهنمای استفاده از برنامه همراه با لیست دستورات مجاز به کاربر نمایش داده می‌شود.

۴. مقدار دستور گرفته شده و آرگومان‌های باقی‌مانده به صورت جداگانه ذخیره می‌شوند تا برای فراخوانی توابع مرتبط استفاده شوند.

۵. با توجه به دستور، یکی از توابع زیر فراخوانی می‌شود:

- start: راه‌اندازی کانتینر جدید.
- list: لیست کردن کانتینرهای در حال اجرا.
- status: بررسی وضعیت کانتینر مشخص شده.
- stop: متوقف کردن کانتینر.
- inspect: نمایش متادیتا و اطلاعات کانتینر.
- freeze: فریز (متوقف کردن موقت) کانتینر با استفاده از .cgroup
- thaw: بازگردانی کانتینر از حالت فریز شده.
- rm: حذف کامل کانتینر و پاکسازی منابع آن.

۶. در صورت دریافت دستور mountdev، فرایند زیر طی می‌شود:

(آ) ابتدا بررسی می‌شود که تعداد آرگومان‌ها برای mountdev حداقل ۴ باشد؛ یعنی دستور به صورت زیر باشد:
mountdev <device_path> <mount_point_in_container> <pid1> [pid2] ...

(ب) مسیر دستگاه (device) و مسیر مقصد داخل کانتینر خوانده می‌شود.

(ج) ابتدا دستگاه روی میزبان در مسیر /mnt/shared_dev به صورت mount با گزینه MS_RELATIME مونت می‌شود. اگر مسیر دستگاه وجود نداشته باشد یا مونت با خطا مواجه شود، برنامه با پیغام خطا خاتمه می‌یابد.

(د) مسیر mnt/ روی میزبان به حالت shared تنظیم می‌شود تا تغییرات مونت در این مسیر به زیرشاخه‌ها و کانتینرها منتشر شود (propagation).

(ه) **(قسمت اضافه شده و بسیار مهم):** مسیر /mnt/shared_dev که دستگاه روی میزبان آنجا مونت شده نیز باید به حالت shared تغییر کند تا این مسیر هم بتواند تغییراتش را به کانتینرها منتقل کند.

(و) سپس برای هر PID که به عنوان آرگومان داده شده، تابع mount_shared_device فراخوانی می‌شود تا دستگاه به مسیر مقصد در فضای mount هر کانتینر مونت شود.

(ز) در نهایت، پیام موفقیت برای هر کانتینری که عملیات مونت با موفقیت انجام شده، چاپ می‌شود.

۷. اگر دستور وارد شده در میان دستورات شناخته‌شده نبود، پیام خطا به کاربر نمایش داده می‌شود.

نکات کلیدی

- اجرای monitor.py به صورت پردازه جداگانه باعث می‌شود که برنامه بتواند به صورت موازی فعالیت‌های کانتینر را تحت نظر داشته باشد.
- استفاده از fork و execlp در این بخش نشان‌دهنده طراحی سیستم چندپردازه‌ای و جداسازی وظایف است.
- تغییر حالت mnt/ و مسیر مشترک /mnt/shared_dev به shared بسیار مهم است تا mount propagation بین میزبان و کانتینرها به درستی انجام شود و دستگاه یا پوشه به صورت قابل دسترس در کانتینرها ظاهر شود.
- هر کانتینر با PID مشخص به طور مستقل مدیریت می‌شود و امکان مونت کردن مسیرهای مشترک برای چند کانتینر فراهم شده است.
- برنامه به صورت کامل پیام‌های خطا را مدیریت می‌کند و در صورت وجود خطا، علت آن را با perror یا fprintf گزارش می‌دهد.

۶ نظارت با eBPF

برای مشاهده و ثبت سیستم‌کال‌هایی مانند clone، mount، unshare و ... که توسط کانتینر فراخوانی می‌شوند، از اسکریپت Python مبتنی بر eBPF و کتابخانه BCC استفاده شده است.

ساختار فایل monitor.py

این فایل با استفاده از eBPF، برخی هایsyscall مهم مربوط به ایجاد و مدیریت کانتینر را شنود (trace) می‌کند. کد BPF ابتدا به صورت رشته تعریف شده است:

```
bpf_source = """
#include <uapi/linux/ptrace.h>

int trace_clone(struct pt_regs *ctx) {
    bpf_trace_printk("CLONE syscall by PID %d\\n",

    bpf_get_current_pid_tgid() >> 32);
    return 0;
}
...
"""
```

در اینجا از 32 « ()bpf_get_current_pid_tgid برای دریافت PID واقعی فرآیند استفاده شده است.

اتصال به هاkprobe

با استفاده از attach_kprobe سیستم‌کال‌های مشخصی را مانیتور می‌کنیم:

```
b.attach_kprobe(event="__x64_sys_clone", fn_name="trace_clone")
b.attach_kprobe(event="__x64_sys_unshare", fn_name="trace_unshare")
...
```

ذخیره در لاگ

خروجی مانیتور به صورت زنده در فایل متنی ذخیره می‌شود:

```
log_path = os.path.join(os.path.dirname(__file__), "container.log")
with open(log_path, "a") as logfile:
    while True:
        (task, pid, cpu, flags, ts, msg) = b.trace_fields()
        logfile.write(f"[{time.strftime('%H:%M:%S')}] PID {pid}: {msg}\n")
```

نحوه ادغام با برنامه اصلی کانتینر

در ابتدای تابع ()main برنامه C، این اسکریپت با استفاده از ()fork و ()execlp اجرا می‌شود:

```
pid_t monitor_pid = fork();
if (monitor_pid == 0) {
    execlp("python3", "python3", "monitor_syscalls.py", NULL);
    perror("Failed to launch syscall monitor");
    exit(1);
}
```

نمونه خروجی فایل لاگ

```
[14:27:05] PID 1123: CLONE syscall by PID 1123
[14:27:05] PID 1123: UNSHARE syscall by PID 1123
[14:27:06] PID 1123: MOUNT syscall by PID 1123
```

جمع‌بندی

ترکیب ابزار eBPF با کانتینرهای ساده، امکان مانیتورینگ سطح کرنل با کارایی بالا را فراهم می‌سازد. استفاده از monitor.py در شروع برنامه، به توسعه‌دهنده امکان بررسی و تحلیل رفتارهای سیستمی در زمان واقعی را می‌دهد.

۷ نتایج آزمایش ها

در این گزارش، مجموعه‌ای از تست‌های دستی جهت بررسی صحت عملکرد Runtime سفارشی کانتینر که بر پایه مفاهیمی مانند names- pace و cgroup طراحی شده است، ارائه می‌شود. این Runtime وظیفه اجرای فرایندها به صورت ایزوله، کنترل منابع و مدیریت چرخه حیات کانتینرها را بر عهده دارد.

در ابتدا باید اسکریپت rootfs ران شود تا مسیر rootfs ساخته شود و بعد از آن با کد c میتوان کانتینر را ساخت که خروجی log مانیتور نیز در همان فولدر ذخیره میشود.

مشخصات محیط تست

- **سیستم عامل میزبان:** Ubuntu ۰۴.۲۲
- **مسیر فایل اجرایی Runtime :** ~/Downloads/container/container
- **مسیر rootfs :** /tmp/my_container_rootfs

تست‌ها

در هر تست، هدف، دستور اجرا و خروجی مورد انتظار ذکر شده است.

۱.۷ تست ۱: اجرای اولیه کانتینر

دستور اجرا:

```
sudo ./container start /tmp/my_container_rootfs
```

خروجی مورد انتظار: نمایش PID کانتینر و پیام موفقیت‌آمیز بودن راه‌اندازی.
خروجی واقعی:

```
[parent] Container started with PID 5832
```

۲.۷ تست ۲: بررسی وضعیت کانتینر

دستور اجرا:

```
sudo ./container status 5832
```

انتظار: نمایش وضعیت «در حال اجرا» یا «متوقف شده».
خروجی:

```
Container with PID 5832 is running
```

۳.۷ تست ۳: مشاهده تنظیمات کانتینر

دستور:

```
sudo ./container inspect 5832
```

انتظار: نمایش تنظیمات منابع و مسیرها.
خروجی:

```
Metadata for container PID 5832:
pid=5832
rootfs=/tmp/my_container_rootfs
memory_limit = 100MB
cpu_quota = 50
overlay_upper = /tmp/container_5832_upper
overlay_work = /tmp/container_5832_work
mount_propagation = shared
```

۴.۷ تست ۴: فریز کردن کانتینر

دستور:

sudo ./container freeze 5832

انتظار: تغییر وضعیت کانتینر به حالت توقف .(T)
خروجی:

Container 5832 frozen

یادداشت: این دستور در اجرای دستی درست کار می‌کند، اما در اسکریپت ممکن است به دلیل مغایرت PID مانیتور شده خطا دهد.

۵.۷ تست ۵: thaw کردن کانتینر

دستور:

sudo ./container thaw 5832

انتظار: بازگشت کانتینر به حالت اجرا.
خروجی:

tContainer 5832 thawed

۶.۷ تست ۶: لیست کانتینر

دستور:

sudo ./container list

انتظار: لیستی از تمام کانتینر ها .
خروجی:

Running containers (PIDs): PID 5832

۷.۷ تست ۷: توقف کانتینر

دستور:

sudo ./container stop 5832

انتظار: خاتمه‌ی صحیح اجرای کانتینر.
خروجی:

sent SIGKILL to container 5832

۸.۷ تست ۸: حذف متادیتای کانتینر

دستور:

sudo ./container rm 5832

انتظار: حذف اطلاعات مرتبط با کانتینر از سیستم فایل.
خروجی: cgroup and records from removed ۵۸۳۲ Container dead. already is ۵۸۳۲ process

۹.۷ تست ۹: اجرای فایل باینری

دستور:
در اسکریپت ایجاد rootfs یک فایل باینری نیز در /tmp/my_contair_rootfs/bin/myprogram کپی میشود که وقتی در bash کانتینر اسم ان را بنویسیم اجرا میشود اسم این فایل باینری myprogram است و محتویات آن چاپ کردن یک جمله می باشد .
انتظار:
جمله binary! container the inside from hello
خروجی: binary! container the inside from hello

۱۰.۷ جمع‌بندی

با اجرای مجموعه تست‌های فوق، عملکرد Runtime پیاده‌سازی‌شده برای مدیریت کانتینرها مورد بررسی دقیق قرار گرفت. نتایج به‌دست‌آمده به صورت زیر قابل تحلیل است:

- فرآیند **راه‌اندازی اولیه کانتینر** به‌درستی انجام شده و PID اختصاص‌یافته به‌طور موفق ثبت گردیده است.
- دستورات مربوط به **وضعیت**، **بازبینی تنظیمات** و **فریز/ذوب کانتینر** نیز عملکرد صحیحی از خود نشان دادند. این موارد نشان‌دهنده پیاده‌سازی موفق کنترل وضعیت‌های اجرای فرایند (از جمله SIGSTOP و SIGCONT) می‌باشد.
- فرآیند **متوقف‌سازی کانتینر** با ارسال سیگنال SIGKILL با موفقیت به پایان رسید و نشان داد که Runtime توانایی خاتمه دادن ایمن به کانتینر را داراست.
- عملیات **حذف متادیتا** نیز بدون مشکل انجام شد که بیانگر پاکسازی صحیح منابع پس از اتمام اجرای کانتینر است.
- لازم به ذکر است که کلیه این تست‌ها به‌صورت دستی انجام شده‌اند تا از بروز مغایرت در PID فرایندها (که در حالت خودکار و اسکریپتی مشاهده می‌شد) جلوگیری شود.

در مجموع می‌توان نتیجه گرفت که Runtime طراحی‌شده، قابلیت‌های پایه‌ای مورد انتظار برای مدیریت چرخه حیات یک کانتینر (راه‌اندازی، ایزولاسیون، نظارت، توقف و پاکسازی) را به‌درستی پیاده‌سازی کرده و عملکرد آن از منظر عملیاتی رضایت‌بخش است.

۸ rootfs_script

اسکریپت Bash مورد بررسی، یک محیط ایزوله شبه‌ـ کانتینری ایجاد می‌کند. این محیط بر پایه دستورات پایه لینوکس و با استفاده از chroot ساخته شده است. در ادامه، هر بخش اسکریپت به تفصیل بررسی می‌شود.

۱.۸ مرحله ۰: تعریف مسیر rootfs

در ابتدا مسیر ریشه سیستم فایل کانتینر مشخص شده و اگر وجود نداشته باشد ساخته می‌شود

۲.۸ مرحله ۱: ساخت ساختار پوشه‌ها

دایرکتوری‌های ضروری سیستم مانند `/bin`، `/lib`، `/proc` و ... ساخته می‌شوند

۳.۸ مرحله ۲: کپی باینری‌ها

دستورات پایه‌ای مانند ،bash ،ls ،sh cat و hostname به محیط جدید منتقل می‌شوند

۴.۸ مرحله ۳: کپی کتابخانه‌ها

باینری‌ها برای اجرا نیاز به libraries shared دارند. برخی از این کتابخانه‌ها مانند .so.۶libc، .so.۲libdl کپی می‌شوند

۵.۸ مرحله ۴: bind-mount دایرکتوری dev/

برای فراهم کردن دسترسی به فایل‌های سیستمی مانند `/dev/null`، دایرکتوری dev/ از سیستم میزبان به rootfs mount می‌شود

۶.۸ مرحله ۵: mount کردن proc/ و sys/

برای اجرای ابزارهایی مثل ps یا مشاهده اطلاعات سیستم، این دایرکتوری‌ها باید mount شوند

۷.۸ مرحله ۶: ورود به محیط با chroot

با اجرای دستور ،chroot ریشه سیستم فایل به دایرکتوری کانتینر تغییر می‌کند و کاربر وارد آن می‌شود

۸.۸ مرحله ۷: خروج و پاک‌سازی

پس از خروج، هایmount ایجاد شده unmount می‌شوند

۹.۸ جمع‌بندی

این اسکریپت ساده، یک محیط ایزوله شده با ساختار پایه‌ای لینوکس ایجاد می‌کند. این محیط می‌تواند به عنوان پایه‌ای برای توسعه container واقعی‌تر با استفاده از ،هاnamespace هاcgroup و propagation mount نیز به کار رود.

۹ نقد و بررسی سیستم

در فرآیند توسعه‌ی این سیستم مجازی‌سازی ساده Container (Simple ،(Runtime چالش‌ها و محدودیت‌هایی شناسایی شد که در ادامه تحلیل شده و پیشنهاداتی برای بهبود ارائه می‌شود.

۱.۹ مشکلات و محدودیت‌ها

- **پیچیدگی در مدیریت فضای OverlayFS :** استفاده از OverlayFS برای ساخت محیط ایزوله نیازمند مدیریت دقیق مسیرهای upperdir، workdir و نقاط اتصال است. حذف ناکامل این فضاها پس از اجرای container ممکن است منجر به نشت منابع شود.
- **امنیت محدود در اجرای دستورات :** با وجود حذف قابلیت‌ها (drop_capabilities)، هنوز برخی تهدیدات امنیتی بالقوه مثل دسترسی به منابع مشترک یا امکان اجرای کد مخرب در فضای mnt/ باقی مانده است.
- **عدم استفاده از seccomp یا :AppArmorc** در این پیاده‌سازی، فیلترهای سیستمی مانند seccomp جهت محدودسازی هاsyscall اعمال نشده‌اند که می‌تواند سطح حملات احتمالی را افزایش دهد.
- **مدیریت ناکامل سیگنال‌ها و خطاها:** در مواقعی که اجرای فرآیند با خطا مواجه می‌شود (مثل خطای mount یا ،(chroot تمام منابع cgroup، overlayfs و ...) به درستی آزاد نمی‌شوند.

۲.۹ پیشنهادهای بهبود

- **افزودن پشتیبانی از Seccomp و AppArmor :** فیلتر کردن هاsyscall و استفاده از پروفایل‌های امنیتی باعث افزایش سطح ایمنی کانتینرها خواهد شد.
- **مدیریت جامع‌تر منابع:** بهینه‌سازی مکانیزم‌های حذف و پاک‌سازی (clean-up) برای cgroup، فایل‌های metadata و overlayfs به صورت خودکار و ایمن ضروری است.
- **استفاده از daemon سبک:** تعریف یک daemon مرکزی برای پیگیری وضعیت کانتینرها (با استفاده از Socket Unix یا REST API) می‌تواند کارایی و پایایی سیستم را افزایش دهد.
- **بهبود کنترل خطا و لاگ‌گیری:** اضافه کردن لاگ‌گیری دقیق برای عملیات mount، exec chroot و سایر مراحل بحرانی می‌تواند اشکال‌زدایی سیستم را تسهیل کند.
- **افزایش پایداری در برابر crash :** استفاده از مکانیزم‌های journaled یا transactional در نوشتن اطلاعات PID و metadata (مثلاً از طریق SQLite یا ساختارهای atomic فایل) می‌تواند از نشت وضعیت جلوگیری کند.
- **افزودن ابزار CLI گسترش‌پذیر:** گسترش رابط خط فرمان با قابلیت‌هایی مانند log مشاهده، metrics live و محدودسازی شبکه می‌تواند بهره‌وری توسعه‌دهنده را افزایش دهد.

در مجموع، این سیستم پایه‌ای مناسب برای درک مفاهیم مجازی‌سازی لینوکسی مانند namespace، cgroup chroot و mount فراهم می‌کند، ولی برای استفاده در محیط‌های عملیاتی نیازمند بهینه‌سازی‌های بیشتر و افزایش قابلیت اطمینان و امنیت است.