Linux namespaces and these things called containers

Martin Wagner

March 16, 2023

Outline

- Working from the inside out
- What are namespaces? Why were they created?
- How can they be used for constructing containers
- High level container tutorial & demo

Problem

- Global system resources are shared between all processes
  - List of mount points
  - PID numbers
- Major changes not possible without disruption of the entire system
  - Can't just unmount /
  - Only one process can be PID 1

Solution: namespaces

- Isolated version of some system resource
- Processes attached to a namespace have a different view on this resource

```
> ps -e | wc -l
251
> ./demo_1 # spawns shell in new namespace
> ps -e | wc -l
4
> echo $$
1
```

- Work a bit like “The Matrix”

Usecases

- Sandbox and isolation
  - Hide parts of the system from a process
  - Limits damage in case of vulnerability
- Containers
  - Package a known good environment for an application
  - Always setup this environment to ensure stability

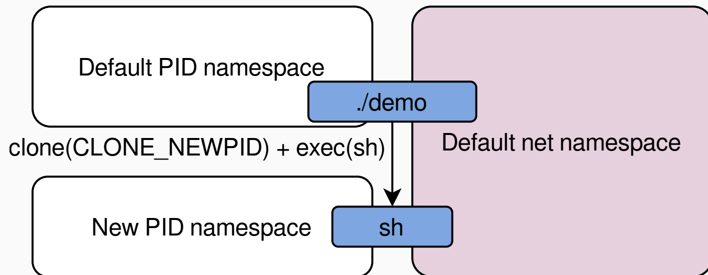
Usage i

- Every process is attached to namespace for every resource
- Eighth namespace types (mounts, pid, users, hostname, time, cgroup, ipc, net)
- Initial default namespaces exist

Usage ii

- Links in `/proc/*/ns/`
  - One for each namespace type
  - Point to the same inode for processes in the same namespace
- Syscalls
  - `clone` starts a new process in a new namespace
  - `unshare` moves caller in new namespace
  - `setns` takes fd to `/proc/*/ns/*` and moves caller into that namespace
- Tree shape: process in parent namespace can freely inspect child namespaces

Usage iii



Example: running a container

```c
int main(int argc, char *argv[]) {
  int flags = CLONE_NEWUSER | CLONE_NEWNS;
  int pid = clone(child_main, child_stack + sizeof(child_stack),
    flags | SIGCHLD, argv);

  waitpid(pid, NULL, 0);
  return 0;
}
```

Example ii

```c
int child_main() {
  mount("./container", "./containers", "bind", MS_BIND | MS_REC, "");
  mkdir("./container/oldrootfs", 0755);
  syscall(SYS_pivot_root, "./container", "./container/oldrootfs");
  chdir("/");

  umount2("/oldrootfs", MNT_DETACH);
  rmdir("/oldrootfs");

  return execvp("sh", NULL);
}
```

Example iii

```
$ ls /home
martin
$ ls ./container
bin   dev  home  lib64  opt   root  sbin  sys  usr
...
$ ./demo_2
# ls /home
hello_from_the_other_side
```

User namespaces

- By default, only root can create new namespaces
- Exception: user namespace
  - If enabled, unprivileged users can create new user namespace
  - User becomes root in new namespace and can create other namespaces
- Privileged syscalls available to users in user namespaces

Integrations

- Sandboxing
  - Unmount `/home` for system services
  - Hide running processes from a webserver
  - Examples: Firefox, Chrome, systemd
- Container runtime
  - Use all the available namespacing capabilities
  - Replace root filesystem with an application specific one
  - Examples: Podman, Docker
- Combined with other technologies like `seccomp` and `apparmor` for security

Limitations

- Shared kernel
    - Kernel vulnerabilities can break isolation
    - Can't run other operating systems or different kernel versions
- No replacement for VMs
- Increase the kernel's attack surface

Summary

- Partition of system resources
- Building block for sandboxing and containers
- Shared kernel, no replacement for VMs

Containers in CTF

- Container escape?
  - Not today :(
- Containers as a way to run challenges locally
- Containers as a way to run tools

Container terms

- Container runtime
  - docker, podman
- Container image
  - A filesystem image that can be used as basis for containers
- Container
  - An instance of a container image with some additional configs and state
- Container orchestration
  - Something manages (multiple) containers. `docker compose`, kubernetes

Running on non-linux

/shrug [1]

[1] I failed to typeset the real thing in pandoc

docker vs podman

- Docker is the older and more popular runtime
  - Requires a daemon running as root
  - Requires root privileges to interact with
  - Is what people expect you to run
- Podman is a new alternative
  - No daemon, no root privileges
  - Is what you should be using 90% of the time (on linux)
  - CLI tries to be compatible. Just switch the binary name

Getting started

```
podman run nginx
```

- Loads the `nginx` container image
- Starts the container with a random name
- Prints logs, unless started with `-d`

Useful options

```
podman run -it \        # interactive with tty
  # mount directory into the container
  -v $(pwd)/www:/usr/share/nginx/html \
  -p 1337:80 \          # expose port 80 as 0.0.0.0:1337
  --rm \                # delete the container after it exits
  --name=kitctf \       # give it an explicit name
  -e TEST=123 \         # Set environment variable
    nginx:alpine        # specify an image with a tag
```

Managing containers

- `podman ps -a`
  - List all containers (`-a` include stopped containers)
- `podman stop kitctf` & `podman start kitctf`
  - Manage created containers
- `podman rm kitctf`
  - Delete containers once you are done

Working with running containers

- We can enter running containers
  - `podman exec -it kitctf bash`
- We can copy files from and into containers
  - `podman cp kitctf:/some/file some-file`

Building images

- Container images need to come from somewhere
- Basically just a tar archive with all the files in the container
- Easiest way to build images: `Containerfile` / `Dockerfile`
  - File containing commands that should be executed to build the container
  - `podman build -t my-image-name .` to build container in current working directory
- Sometimes events provide a `Dockerfile` but no image

Containerfile (née Dockerfile)

```
FROM ubuntu:22.04

RUN apt update && apt install -y python3 python3-pip

COPY ./python-app /opt/app
WORKDIR /opt/app

RUN pip install -r requirements.txt

CMD ["python3", "main.py"]
```

Managing images

- List images on your system
  - `podman images`
- Pull an image from a remote registry
  - `podman pull ubuntu:20.04`
  - `podman pull registry.example.com/base/challenge`
- Delete a local image
  - `podman rmi ubuntu:20.04`

Running multiple containers

- Lots of way to manage containers
    - `docker-compose` / `podman-compose` is a popular choice
- `docker-compose.yml` defines which containers to run
    - `podman-compose up` start all container
    - `podman-compose down` stop & delete all containers
- `ps`, `exec`, `cp` also work with `podman-compose`

Minimal compose example

```yaml
services:
  python-app:
    build: .

  nginx:
    image: nginx:alpine
    volume:
      - ./www:/usr/share/nginx/html
    ports:
      - 127.0.0.1:1337:80
```

Most important command

`podman system prune`

- Delete all unused resources
- 10s of GB of storage

Useful tips for CTF

- Debugging in containers
  - Run your containers with `--privileged --security-opt seccomp=unconfiend` to run gdb in container
  - PID in container != PID outside of container. Be aware when attaching from the outside
- Mount your local version into the container
  - Allows easier changing / debugging of challenge files without rebuilding