Linux Primitives

Nati Cohen (@nocoot)
Avishai Ish-Shalom (@nukemberg)

What we know as Linux is actually GNU user space

Our apps interact with *GNU libc* and other userspace libraries

- All the system calls documented in man 2 are actually glibc wrappers
  - Some syscalls are not wrapped, and require usage of syscall()
- Libraries provide many basic mechanisms - e.g. *malloc()*, resolver
- Provides POSIX API, ISO C API, BSD/Unix compat
- There is more than one libc implementation
- We can package our own libc in a container or binary
- Container isolation works in the kernel level

Linux containers

- There is no “Linux container” primitive in the kernel
- A “container” is a group of processes associated with common namespaces/cgroups/root/etc

What’s on the menu tonight

- Processes
- Mounts
- chroot/pivot_root
- CoW storage
- Users
- Namespaces
- Memory management

Processes - Data Structures

Under the hood both threads and processes are *tasks*

- *task_struct* - ~ 170 fields, ~1k size. Some notable fields: *user, pid, tgid, *files, *fs, *nsproxy
- *fs_struct *fs* holds information on current root
- *pid* struct maps processes to one or more *tasks*

Processes - fork & exec

Traditionally *nixs created new processes using:

1. *fork()* - Duplicate the current **process**, VM is copy-on-write
2. *exec()* - Replace text/data/bss/stack with new program

- glibc's *fork()* and *pthread_create()* both call *clone()* syscall
- *clone()* creates a new task_struct from parent
    - controls resource sharing by flags (e.g. share VM, share/copy fd)

Users

From the kernel’s PoV, a user is an int parameter in various structs

- A process has several uid fields: *ruid*, suid, *euid*, *fsuid*
- No need to “add” users
- *useradd* manipulates /etc/passwd, /etc/shadow which are accesses by userspace tools
- User names are a userspace feature, the kernel doesn’t care
- No identity checks (NFS i’m looking at you)

Capabilities

Traditionally, UNIX is a monotheistic O/S: you are either god or mortal

- Many things require root privileges, e.g. ICMP, ports < 1024
- *setuid* binaries allow mortals to run ping, etc
  - but can be used for Privilege Escalation
- To avoid over-privileged processes, root power has been split to various CAPABILITIES
- Capabilities are associated with files and processes using extended attributes

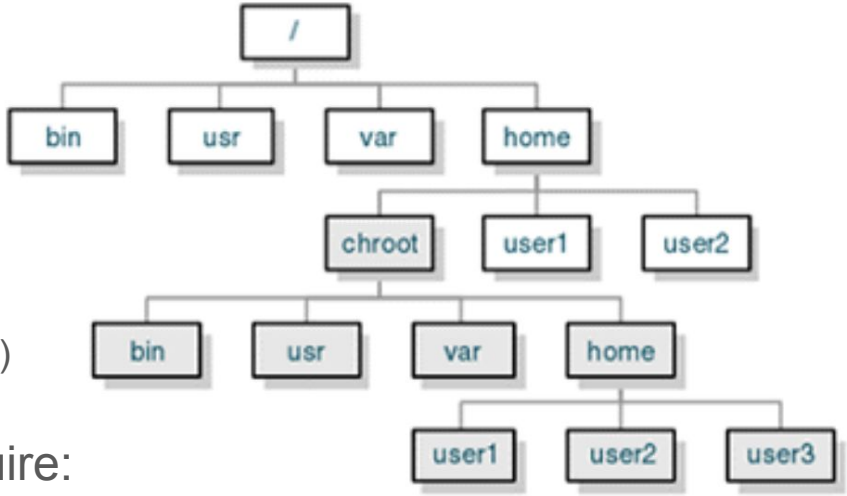
Mounts

Map a file system to a directory

- The Virtual File System provides a file system interface to userspace
- A mount maps some inode in the VFS tree to a file system
- bind mounts map inode in VFS to another inode (yes, files too!)
- Mounts can be:
  - shared - all replicas are the same
  - slave - only receives mount/umount events
  - private - doesn’t forward or receive propagations
  - unbindable - private + unbindable

chroot(new_root)

Change the root directory for a process



- Traditional uses:
  - “Jail” vulnerable processes (e.g. Apache, vsftpd)
  - Building packages (mock, debootstrap)
- Running processes in a chroot might require:
  - Special mounts- procfs, sysfs, devpts
  - Some devices- /dev/{null,zero,[u]random,stdin,stdout,stderr}
  - Dynamically loaded libraries- glibc, jre, libv8
  - Other dependencies- jars, eggs, gems, etc.
- Problem- too many techniques for chroot escaping
  - Common cause- old_root is still accessible

pivot_root(new_root, put_old)

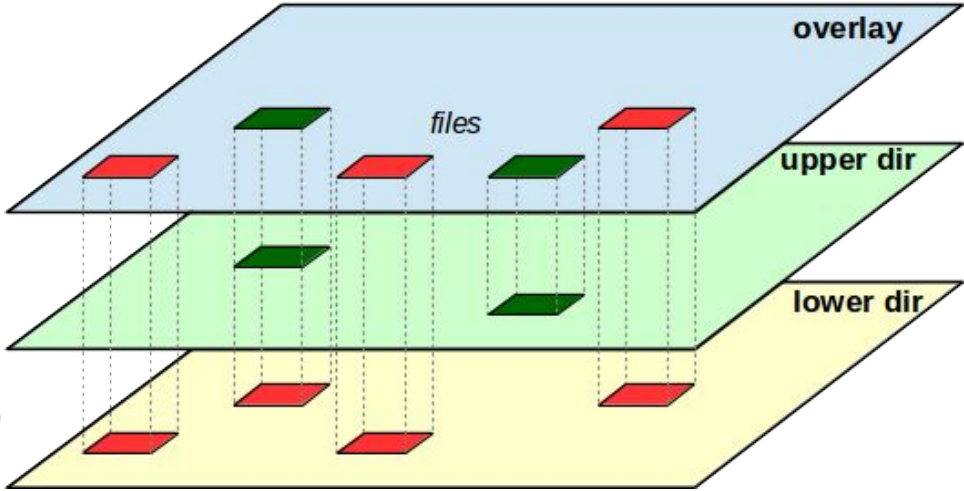
Change the root directory for **all processes in current mnt namespace**

- **Traditional use**: switch from initrd to root file system
- Allows us to umount() put_old
- **For containment**: performed in a new mount namespace
- Can prevent most chroot()'s "design flaws" when paired with
  - Device control group (e.g. no /dev/sda1 *duh*)
  - Capabilities (e.g. prevent creating device files like /dev/sda1)
  - seccomp (blocking some syscalls, e.g. pivot_root)

CoW storage

General idea: start with a common read branch, writes go to a different branch

- Several flavors: Layered filesystems, filesystem snapshots (btrfs), block level snapshots (LVM/DM)
- Original implementation of layered fs: UnionFS; Successor: AUFS
- Mainline kernel rejected AUFS and merged OverlayFS



cgroups

CGroups control, account and limit system resources

- Current implementation (V1) is being replaced by CGroups V2
- Filesystem based API (/sys/fs/cgroup mount)
- Hierarchy
- Accounting and resource limits - CPU, memory, blkio, network
- Control - device whitelist, freezer

Namespaces

Why?

- Consistent world view - needed for Checkpoint/Restore, Process migration
- Isolation, access control

Namespaces

The namespaces API:

- clone - create a new process in a new namespace
- unshare - create a new namespace for current process
- setns - set process namespace from existing ns fd
- /proc/<pid>/ns - fd to process namespaces

Namespaces

```
struct nsproxy {
        atomic_t count;
        struct uts_namespace *uts_ns;
        struct ipc_namespace *ipc_ns;
        struct mnt_namespace *mnt_ns;
        struct pid_namespace *pid_ns_for_children;
        struct net           *net_ns;
};
```

For PID namespace *task_active_pid_ns()* finds ns using *pid* and *upid* structs

PID namespace - How

```
struct upid {
        /* Try to keep pid_chain in the same cacheline as nr for find_vpid */
        int nr;
        struct pid_namespace *ns;
        struct hlist_node pid_chain;
};

struct pid
{
        atomic_t count;
        unsigned int level;   <-- New
        /* lists of tasks that use this pid */
        struct hlist_head tasks[PIDTYPE_MAX];
        struct rcu_head rcu;
        struct upid numbers[1];   <-- New
};
```

Before PID namespace

```
struct pid {
        atomic_t count;
        int nr;
        struct hlist_node pid_chain;
        struct hlist_head tasks[PIDTYPE_MAX];
        struct rcu_head rcu;
};
```

"We can solve any problem by introducing an extra level of indirection."

PID namespace behaviour

- setns, unshare - current process unchanged, forks will be in another namespace
- First process in ns is *init* - pid 1
- When *init* dies, all processes in ns die with it
- On parent death processes re-parent to *init* of the ns
- Processes forked from outside have ppid 0
- Userspace tools (e.g. ps) work with /proc, so we need mount ns too

PID namespaces form a hierarchy.

UID/user namespace

Map uid and gid numbers

- Allow non-root user to be root in the namespace
- Some actions still require “real” root privileges
- /proc/<pid>/uid_map
- Hierarchy of namespaces, up to 32 levels of nesting
- Interacts with other namespaces (e.g. mount namespace)

Memory management

What happens when memory runs out?

- With *vm.overcommit_memory=2 malloc()* returns NULL
- If using overcommit, *malloc()* may succeed, later *oom-killer* may be activated

OOM Killer heuristics find fattest, laziest SOB and kills it.

- Can be controlled by */proc/<pid>/oom_score_adj*

setrlimit(resource, rlim)

Traditional approach for per-task memory limiting (aka ulimit)

- Limit per task, children’s accounting “starts over”
  - Can’t control a group of tasks
- Limits stack, virtual memory and locked pages separately
- Page cache is shared

Memory Control Group

- Limit a group of tasks
- Limit **some** kernel memory (stack, sl[au]b, tcp/socket memory)
- Page cache is per-group (sort of)
- Knobs and gauges per-group
  - Enable/Disable OOM Killer, Control Swappiness (swap vs page cache eviction)
  - /proc/meminfo style accounting (vi memory.stat)
- Has some overhead (40b per-page on x86_64)
- Controlled using kernel parameter
  - cgroup_disable/cgroup_enable=memory
  - Disabled by default on Ubuntu due to performance hit

More info in the Linux kernel documentation for Memory Resource Controller

Memory Control Group - OOM

What happens when we allocate too much memory?

- Swap pages until *memory.memsw.limit_in_bytes* is hit
- Then activate cgroup oom-killer
- If oom-killer disabled, **block** the allocating process
- You can register for oom and memory pressure events

References

Glibc and the kernel user-space API [LWN]

Virtual File System [Kernel Docs]

Shared Subtrees [Kernel Docs]

Docker from Scratch workshop repo