New York Linux User Group

# Understanding user namespaces

Michael Kerrisk, man7.org © 2018
mtk@man7.org

New York Linux User Group
19 September 2018

# Outline

# Outline

# Who am I?

- Contributor to Linux *man-pages* project since 2000
  - Maintainer since 2004
  - Project provides ≈1050 manual pages, primarily documenting system calls and C library functions
- Author of a book on the Linux programming interface
- Trainer/writer/engineer
  - Lots of courses at *http://man7.org/training/*

# Time is short

- Normally, I would spend several hours on this topic
- Many details left out, but I hope to give an idea of big picture
- We'll go fast

# Outline

# (Traditional) superuser and set-UID-*root* programs

- Traditional UNIX privilege model divides users into two groups:
  - **Normal users**, subject to privilege checking based on UID (user ID) and GIDs (group IDs)
  - **Superuser** (UID 0) bypasses many of those checks
- Traditional mechanism for giving privilege to non-superusers is **set-UID-*root* program**

```
# chown root prog
# chmod u+s prog
```

  - When executed, **process assumes UID of file owner**
    - $\Rightarrow$ process gains privileges of superuser
  - Powerful, but dangerous

# The traditional privilege model is a problem

- Coarse granularity of traditional privilege model is a problem:
  - E.g., say we want to give user power to change system time
  - Must also give it power to do everything else *root* can do
  - ⇒ **No limit on possible damage** if program is compromised
- **Capabilities** are an attempt to solve this problem

# Background: capabilities

- Capabilities: divide power of superuser into small pieces
  - 38 capabilities as at Linux 4.19 (see *capabilities(7)*)
  - Examples:
    - `CAP_DAC_OVERRIDE`: bypass all file permission checks
    - `CAP_SYS_ADMIN`: do (too) many different sysadmin operations
    - `CAP_SYS_TIME`: change system time
- Instead of set-UID-*root* programs, have programs with one/a few attached capabilities
  - Attached using *setcap(8)* (needs `CAP_SETFCAP` capability!)
  - When program is executed ⇒ process gets those capabilities

# Background: capabilities

- **Summary**:
  - Processes can have capabilities (**subset** of power of *root*)
  - Files can have attached capabilities, which are given to process that executes program

# Outline

# Namespaces

- A namespace (NS) “wraps” some global system resource to provide resource isolation
- Linux supports multiple (currently, seven) NS types

# Each NS isolates some kind of resource(s)

- **Mount** NS: isolate mount point list
  - (CLONE_NEWNS; 2.4.19, 2002)
- **UTS** NS: isolate system identifiers (e.g., hostname)
  - (CLONE_NEWUTS; 2.6.19, 2006)
- **IPC** NS: isolate System V IPC and POSIX MQ objects
  - (CLONE_NEWIPC; 2.6.19, 2006)
- **PID** NS: isolate PID number space
  - (CLONE_NEWPID; 2.6.24, 2008)
- **Network** NS: isolate NW resources (firewall & routing rules, socket port numbers, /proc/net, /sys/class/net, ...)
  - (CLONE_NEWNET; ≈2.6.29, 2009)

# Each NS isolates some kind of resource(s)

- **User** NS: isolate user ID and group ID number spaces
  - (`CLONE_NEWUSER`; **3.8, 2013**)
- **Cgroup** NS: virtualize (isolate) certain cgroup pathnames
  - (`CLONE_NEWCGROUP`; 4.6, 2016)

# Namespaces

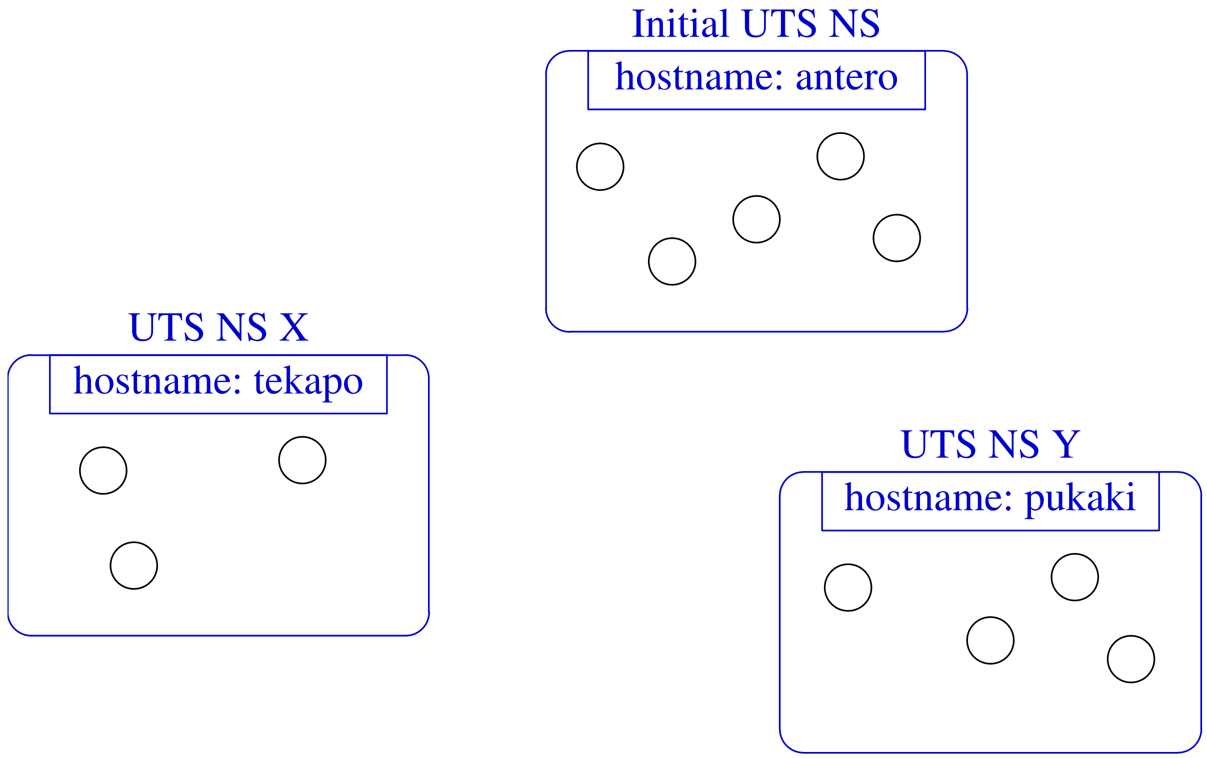
- For each NS type:
  - Multiple **instances** of NS may exist on a system
  - At system boot, there is one instance of each NS type–the **initial namespace**
  - A process resides in one NS instance (of each of NS types)
  - To processes inside NS instance, it appears that only they can see/modify corresponding global resource
    - (They unaware of other instances of resource)
- When new child process is created (*fork()*), it resides in same set of NSs as parent process
  - There are system calls (and commands) for creating new NSs and moving processes into NSs

# Namespaces example

Example: **UTS namespaces**

- **Isolate** certain system identifiers, including **hostname**
  - *hostname(1)*, *uname(1)*, *uname(2)*
- Running system may have multiple UTS NS instances
- Processes in same NS instance access (get/set) same hostname
- Each NS instance has its own hostname
  - Changes to hostname in one NS instance are invisible to other instances

# UTS namespace instances




Each UTS NS contains a set of processes (circles) which access (see/modify) same hostname

# Outline

# Some “magic” symlinks

- Each process has some symlink files in `/proc/`*`PID`*`/ns`

```
/proc/PID/ns/cgroup      # Cgroup NS instance
/proc/PID/ns/ipc         # IPC NS instance
/proc/PID/ns/mnt         # Mount NS instance
/proc/PID/ns/net         # Network NS instance
/proc/PID/ns/pid         # PID NS instance
/proc/PID/ns/user        # User NS instance
/proc/PID/ns/uts         # UTS NS instance
```

  - One symlink for each of the NS types

# Some "magic" symlinks

- Target of symlink tells us which NS instance process is in:

```
$ readlink /proc/$$/ns/uts
uts:[4026531838]
```

  - Content has form: *ns-type*: [*magic-inode-#*]
- Various uses for the `/proc/PID/ns` symlinks, including:
  - If processes show same symlink target, they are in same NS

# APIs and commands

- Programs can use various system calls to work with NSs:
  - *clone(2)*: create new process in new NS(s)
  - *unshare(2)*: create new NS/s and move caller into it/them
  - *setns(2)*: move calling process to another (existing) NS instance
  - Plus some special-purpose *ioctl()*s (see *ioctl_ns(2)*)
- There are analogous **shell commands**:
  - *unshare(1)*: create new NS(s) and execute a shell command in the NS(s)
  - *nsenter(1)*: enter existing NS(s) and execute a command

# The *unshare(1)* and *nsenter(1)* commands

*unshare(1)* and *nsenter(1)* have flags for specifying each NS type:

```
unshare [options] [command [arguments]]
  -C       Create new cgroup NS
  -i       Create new IPC NS
  -m       Create new mount NS
  -n       Create new network NS
  -p       Create new PID NS
  -u       Create new UTS NS
  -U       Create new user NS
```

```
nsenter [options] [command [arguments]]
  -t PID   PID of process whose NS(s) should be entered
  -C       Enter cgroup NS of target process
  -i       Enter IPC NS of target process
  -m       Enter mount NS of target process
  -n       Enter network NS of target process
  -p       Enter PID NS of target process
  -u       Enter UTC NS of target process
  -U       Enter user NS of target process
  -a       Enter all NSs of target process
```

# Privilege requirements for creating namespaces

- Creating **user** NS instances requires no privileges
- Creating instances of **other** (non-user) NS types requires privilege
  - `CAP_SYS_ADMIN`

# Demo

- Two terminal windows (*sh1*, *sh2*) in initial UTS NS

```
sh1$ hostname          # Show hostname in initial UTS NS
antero
```

- In *sh2*, create new UTS NS, and change hostname

```
sh2$ hostname          # Show hostname in initial UTS NS
antero
$ PS1='sh2# ' sudo unshare -u bash
sh2# hostname bizarro  # Change hostname
sh2# hostname          # Verify change
bizarro
```

# Demo

- In *sh1*, verify that hostname is unchanged:

```
sh1$ hostname
antero
```

- Compare /proc/PID/ns/uts symlinks in two shells

```
sh1$ readlink /proc/$$/ns/uts
uts:[4026531838]
```

```
sh2# readlink /proc/$$/ns/uts
uts:[4026532855]
```

  - The two shells are in different UTS NSs

# Demo

- From *sh1*, use *nsenter(1)* to create a new shell that is in same NS as *sh2*:

```
sh2# echo $$         # Discover PID of sh2
5912
```

```
sh1$ PS1='sh1# ' sudo nsenter -t 5912 -u
sh1# hostname
bizarro
sh1# readlink /proc/$$/ns/uts
uts:[4026532855]
```

  - Comparing the symlink value, we can see that this shell is in the second (`sh2#`) UTS NS

# Outline

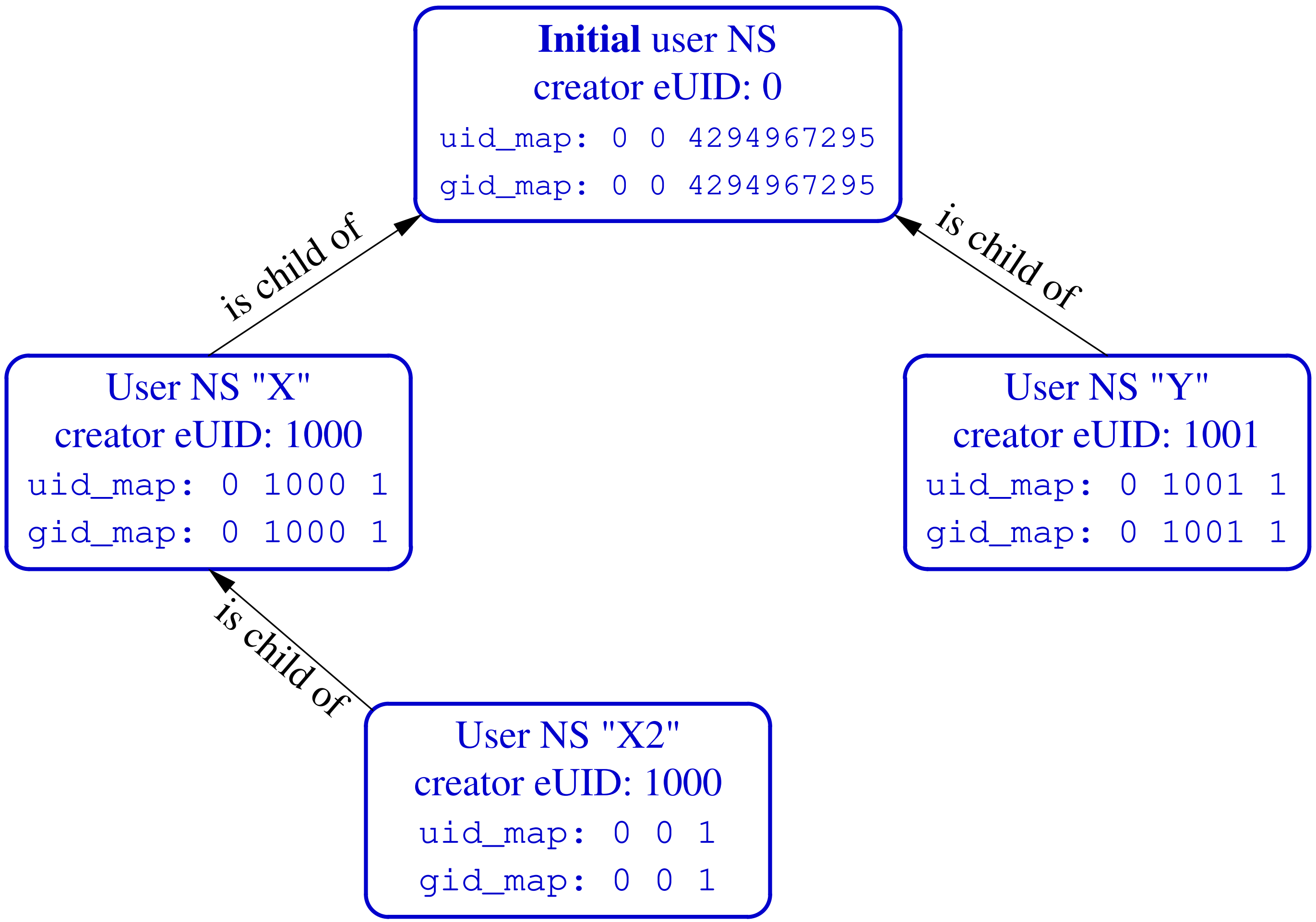
# What do user namespaces do?

- Allow per-namespace **mappings** of UIDs and GIDs
  - I.e., process's UIDs and GIDs inside NS may be different from IDs outside NS
- Interesting use case: process may have nonzero UID outside NS, and UID of 0 inside NS
  - Process has **root privileges *for operations inside user NS***
    - We revisit this point soon...

# Relationships between user namespaces

- User NSs have a hierarchical relationship:
  - A user NS can have zero or more child user NSs
  - Each user NS has parent NS, going back to initial user NS
  - Parent of a user NS == user NS of process that created this user NS
    - Using *clone(2)*, *unshare(2)*, or *unshare(1)*
- Parental relationship determines some rules we look at later

# A user namespace hierarchy



# The first process in a new user NS has root privileges

- When a new user NS is created (*unshare(1)*, *clone(2)*, *unshare(2)*), first process in NS has **all** capabilities
- That process has power of superuser!
- ... but only inside the user NS

# “Root privileges inside a user NS”

- What does “root privileges in a user NS” really mean?
- We’ve already seen that:
  - There are a number of NS types
  - Each NS type governs some global resource(s); e.g.:
    - UTS: hostname, NIS domain name
    - Network: IP routing tables, port numbers, `/proc/net`, ...
- What we will see is that:
  - Each non-user NS is “owned” by a particular user NS
  - “root privileges in a user NS” == root privileges on resources governed by non-user NSs owned by this user NS
    - And **only** on those resources

# Outline

# UID and GID mappings

- One of first steps after creating a user NS is to define UID and GID mappings for NS
  - The chain of mappings back to initial user NS allows kernel to know "true" UID and GID of processes in user NSs
    - So, for example, kernel can determine permissions for accessing files
- Mappings are defined by writing to 2 files: `/proc/PID/uid_map` and `/proc/PID/gid_map`
- For security reasons, there are **many** rules + restrictions on:
  - How/when files may be updated
  - Who can update the files
  - Way too many details to cover here...
    - See *user_namespaces(7)*

# UID and GID mappings

- Records written to/read from `uid_map` and `gid_map` have the form:

```
ID-inside-ns    ID-outside-ns    length
```

  - *ID-inside-ns* and *length* define range of IDs inside user NS that are to be mapped
  - *ID-outside-ns* defines start of corresponding mapped range in "outside" user NS

- Commonly these files are initialized with a single line containing "root mapping":

```
0     1000     1
```

  - One ID, 0, inside NS maps to ID 1000 in outer NS

# Example: creating a user NS with "root" mappings

- `unshare -U -r` creates user NS with root mappings (`-r`)
- Create a user NS with root mappings running new shell, and examine map files:

```
$ id                  # Show credentials in current shell
uid=1000(mtk) gid=1000(mtk) ...

$ PS1='uns2$ ' unshare -U -r bash
uns2$ cat /proc/$$/uid_map
         0       1000          1
uns2$ cat /proc/$$/gid_map
         0       1000          1
```

# Example: creating a user NS with "root" mappings

- Examine credentials and capabilities of new shell:

```
uns2$ id
uid=0(root) gid=0(root) groups=0(root) ...
uns2$ egrep '[UG]id|CapEff' /proc/$$/status
Uid:  0 0 0 0
Gid:  0 0 0 0
CapEff: 0000003fffffffff
```

  - `0x3fffffffff` is bit mask with all 38 capability bits set
    - *pscap* from *libcap-ng* project gives same info more readably

# Example: creating a user NS with “root” mappings

- Discover PID of shell in new user NS:

```
uns2$ echo $$
21135
```

- From a shell in **initial user NS**, examine credentials of that PID:

```
$ grep '[UG]id' /proc/21135/status
Uid:  1000  1000  1000  1000
Gid:  1000  1000  1000  1000
```

# I'm superuser! (But, you're a big fish in a little pond)

- From the shell in new user NS, let's try to change the hostname
  - Requires `CAP_SYS_ADMIN`

```
uns2$ hostname bizarro
hostname: you must be root to change the host name
```

- Shell is UID 0 (superuser) and has `CAP_SYS_ADMIN`
- What went wrong?
- The new shell is in new user NS, but **still resides in initial UTS NS**
  - (Remember: hostname is isolated/governed by UTS NS)
  - Let's look at this more closely...

# Outline

# More on capabilities

- Kernel grants initial process in new user NS a full set of capabilities
- But, those capabilities are available **only for operations on objects governed by the new user NS**

# More on capabilities

- **Each non-user NS instance is owned by some user NS instance**
  - When creating a new non-user NS, kernel associates **user NS of creating process** with new non-user NS
  - There are APIs to discover the ownership relationships
    - See *ioctl_ns(2)*
    - Code examples: `namespaces/pid_namespaces.go`, `namespaces/namespaces_of.go`
- If a process operates on resources governed by non-user NS:
  - Permission checks are done according to that process's capabilities in user NS that owns the non-user NS
- Goal of this scheme: safely deliver full capabilities inside a NS without allowing users to damage wider system

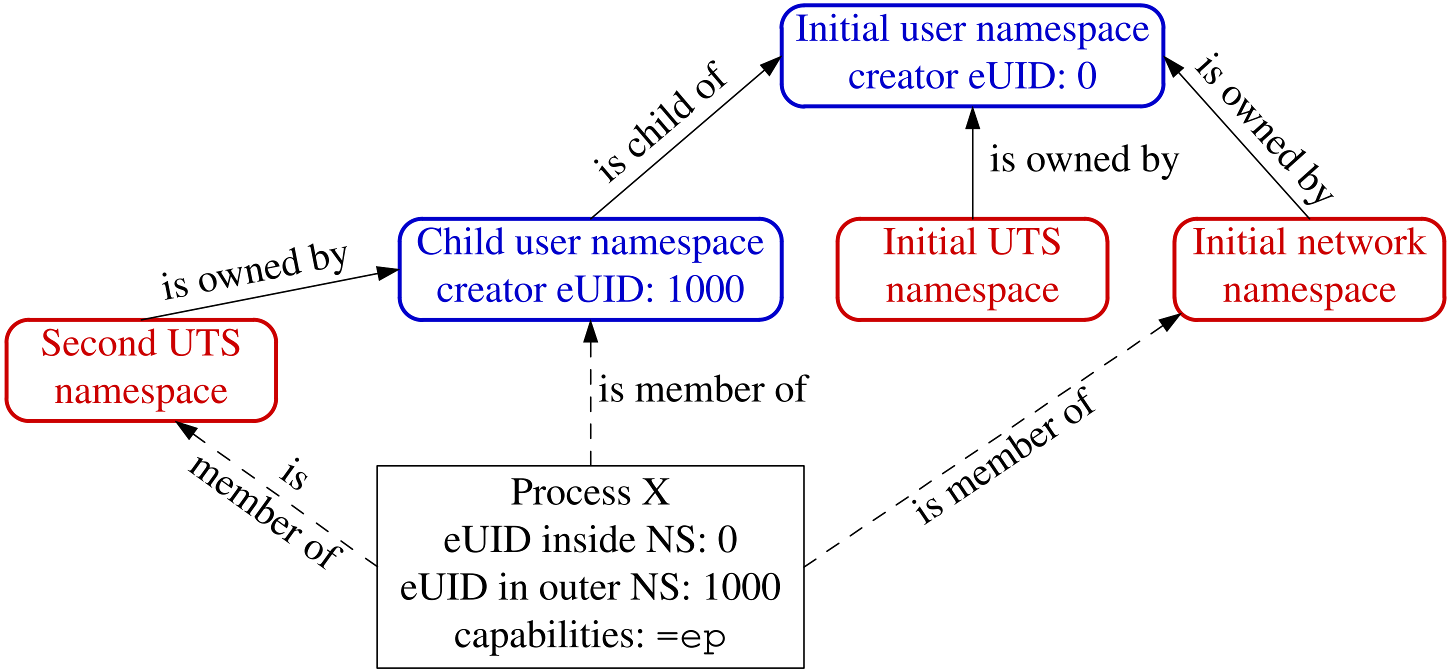
# Example

- Suppose we create a process in new user and UTS NSs, with root mappings for UID (and GID)

```
$ unshare -U -u -r bash
```

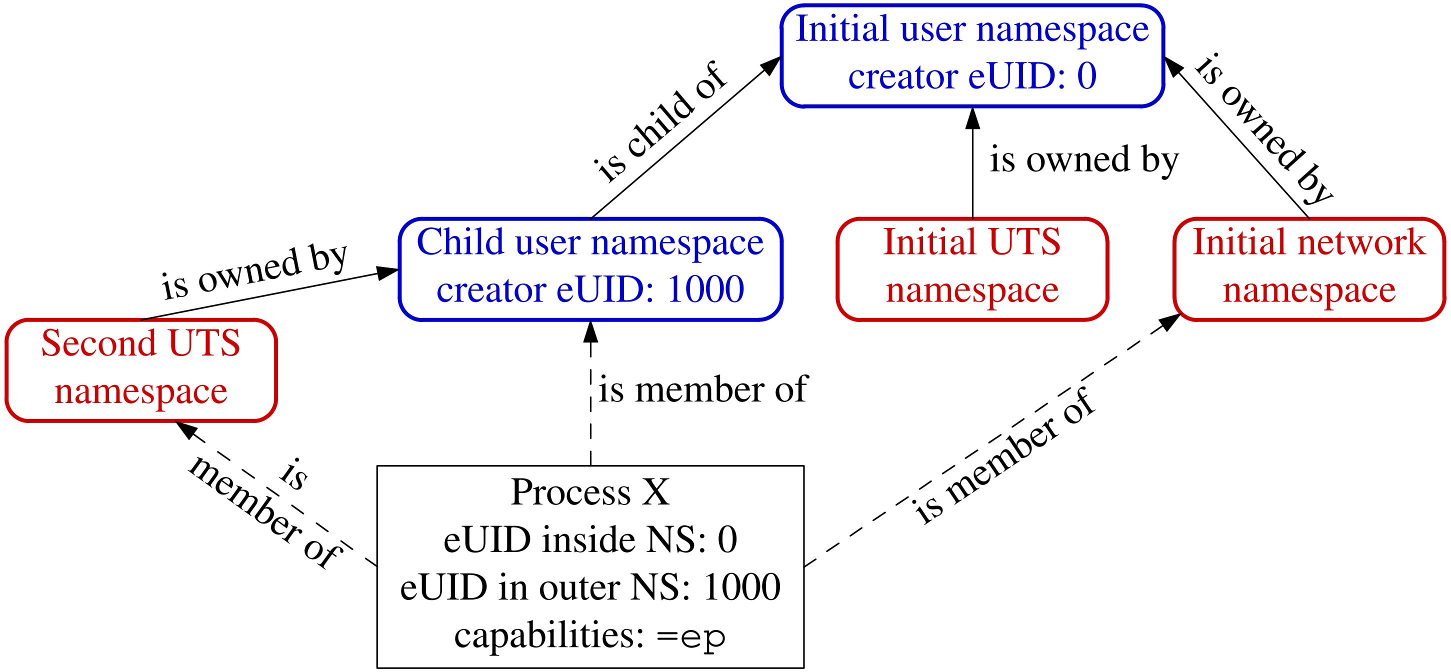
- See diagram

# More on capabilities–an example




- Suppose X tries to change hostname (`CAP_SYS_ADMIN`)
- X is in second **UTS** NS
- Permissions checked according to X's capabilities in user NS that owns that UTS NS ⇒ succeeds (X has capabilities in user NS)

# More on capabilities–an example




- Suppose X tries to bind to reserved socket port (`CAP_NET_BIND_SERVICE`)
- X is in initial **network** NS
- Permissions checked according to X's capabilities in user NS that owns network NS ⇒ attempt fails (no capabilities in initial user NS)

# Outline

# Combining user namesapces and other namespace types

- Earlier, we noted that `CAP_SYS_ADMIN` is needed to create nonuser NSs
- So, why can unprivileged user do this:

```
$ unshare -U -u -r bash
```

- Can do this, because kernel first creates user NS, giving child all privileges, so that UTS NS can also be created
- Equivalent to following, but without intervening child process:

```
$ unshare -U -r bash   # Child in new user NS
$ unshare -u bash      # Grandchild in new UTS NS
```

# What about resources not governed by namespaces?

- Some privileged operations relate to resources/features not (yet) governed by any namespace
  - E.g., system time, kernel modules
- Having all capabilities in a (noninitial) user NS doesn't grant power to perform operations on features not currently governed by any NS
  - E.g., can't change system time or load/unload kernel modules

# But what about accessing files (and other resources)?

- Suppose UID 1000 is mapped to UID 0 inside a user NS
- What happens when process with UID 0 inside user NS tries to access file owned by ("true") UID 0?
- When accessing files, IDs are mapped back to values in initial user NS
  - UID mappings don't allow us to bypass traditional UID/GID permission checks
  - Same principle for checks on other resources that have UID+GID owner
    - E.g., Various IPC objects

# Outline

# User namespaces are hard (even for kernel developers)

- Developer(s) of user NSs put much effort into ensuring capabilities couldn't leak from inner user NS to outside NS
  - Potential risk: some piece of kernel code might not be refactored to account for distinct user NSs
  - $\Rightarrow$ unprivileged user who gains all capabilities in child user NS might be able to do some privileged operation in **outer** NS
- User NS implementation touched a **lot** of kernel code
  - Perhaps there were/are some unexpected corner case that wasn't correctly handled?
  - A number of such cases have occurred (and been fixed)
  - Common cause: many kernel code paths that could formerly be exercised only by root can now be exercised by any user
    - Now, unprivileged users can test for weaknesses in kernel code paths that formerly could be accessed only by root

# Outline

# User namespaces allow interesting possibilities

- User NSs allow unprivileged processes access to functionality formerly reserved to *root*
  - But only inside the user NS!
- User NSs also have implications from a security perspective
  - Unprivileged attackers now have opportunities to test kernel code paths that formerly could be reached only with UID 0
    - Cf. the *setgroups()* vulnerability fixed in Linux 3.19

# User namespaces permit novel applications

- User NSs permit novel applications; for example:
  - Running Linux containers **without** *root* privileges
    - Docker, LXC
  - Chrome-style sandboxes without set-UID-*root* helpers
    - http://dev.chromium.org/developers/design-documents/sandbox
  - User namespace with single UID identity mapping ⇒ no superuser possible!
    - E.g., `uid_map:  1000 1000 1`

# User namespaces permit novel applications

- User NSs permit novel applications; more examples:
  - *chroot()*-based applications for process isolation
    - User NSs allow unprivileged process to create new mount NSs and use *chroot()*
  - *fakeroot*-type applications without `LD_PRELOAD`/dynamic linking tricks
    - *fakeroot(1)* is a tool that makes it appear that you are *root* for purpose of building packages (so packaged files are marked owned by *root*) (*http://fakeroot.alioth.debian.org/*)

# User namespaces permit novel applications

- User NSs permit novel applications; more examples:
  - Firejail: namespaces + seccomp + capabilities for generalized, **simplified** sandboxing of any application
    - https://firejail.wordpress.com/, https://lwn.net/Articles/671534/
- Flatpak: namespaces + seccomp + capabilities + cgroups for application packaging / sandboxing
  - Allows upstream project to provide packaged app with all necessary runtime dependencies
    - No need to rely on packaging in downstream distributions
    - Package once; run on any distribution
  - Desktop applications run seamlessly in GUI
  - http://flatpak.org/, https://lwn.net/Articles/694291/

# Namespaces: sources of further information

- My LWN.net article series *Namespaces in operation*
  - https://lwn.net/Articles/531114/
  - Many example programs and shell sessions...
- Man pages:
  - *namespaces(7)*, *cgroup_namespaces(7)*, *mount_namespaces(7)*, *pid_namespaces(7)*, *user_namespaces(7)*
  - *unshare(1)*, *nsenter(1)*
  - *capabilities(7)*
  - *clone(2)*, *unshare(2)*, *setns(2)*, *ioctl_ns(2)*
- "Linux containers in 500 lines of code"
  - https://blog.lizzie.io/linux-containers-in-500-loc.html

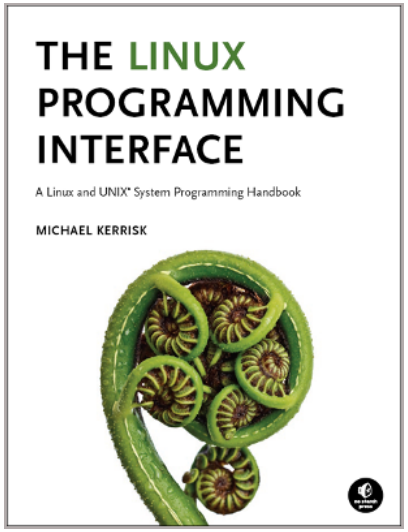
# Thanks!

Michael Kerrisk    mtk@man7.org    @mkerrisk

Slides at http://man7.org/conf/
Source code at http://man7.org/tlpi/code/

Training: Linux system programming, security and isolation APIs, and more; http://man7.org/training/

The Linux Programming Interface, http://man7.org/tlpi/



# Outline

# What are the rules that determine the capabilities that a process has in a given user namespace?

# User namespace hierarchies

- User NSs exist in a hierarchy
  - Each user NS has a parent, going back to initial user NS
- Parental relationship is established when user NS is created:
  - Parent of a new user NS is user NS of process that created new user NS
- Parental relationship is significant because it plays a part in determining capabilities a process has in user NS

# User namespaces and capabilities

- Whether a process has a capability inside a user NS depends on several factors:
  - Whether the capability is present in the process's (effective) capability set
  - Which user NS the process is a member of
  - The (effective) process's UID
  - The (effective) UID of the process that created the user NS
    - At creation time, **kernel records eUID of creator** as "owner UID" of user NS
  - The parental relationship between user NSs
  - (`namespaces/ns_capable.c` program encapsulates the rules shown on next slide–it answers the question, does process P have capabilities in namespace X?)

# Capability rules for user namespaces

1. A process has a capability in a user NS if:
   - it is a **member of the user NS**, and
   - **capability is present in its effective set**
   - **Note**: this rule doesn't grant that capability in **parent** NS
2. A process that has a capability in a user NS **has the capability in all descendant user NSs** as well
   - I.e., members of user NS are not isolated from effects of privileged process in parent/ancestor user NS
3. (All) processes in **parent** user NS that have **same eUID** as eUID of creator of user NS have all capabilities in the NS
   - At creation time, **kernel records eUID of creator** as "owner UID" of user NS
   - By virtue of previous rule, capabilities also propagate into all descendant user NSs