| 1 | Introduction | 3 |
| 2 | Namespaces | 7 |
| 3 | An example: UTS namespaces | 11 |
| 4 | Namespaces commands | 15 |
| 5 | Namespaces demonstration (UTS namespaces) | 20 |
| 6 | Some background: capabilities | 24 |
| 7 | User namespaces overview | 29 |
| 8 | User namespaces: UID and GID mappings | 33 |
| 9 | User namespaces and capabilities | 40 |
| 10 | Use cases and further information | 51 |
| 11 | PS: when does a process have capabilities in a user NS? | 56 |
| 12 | PS: a few more details | 61 |
Why is this interesting?

- User namespaces are cornerstone of unprivileged containers
  - But also many other Linux tools
    - Flatpak / Snap
    - Firejail
    - Modern browser sandboxes
    - Etc.
Who?

- Linux *man-pages* project
    - Approx. 1060 pages documenting syscalls and C library
  - Contributor since 2000
  - Maintainer 2004-2020
  - Comaintainer 2020-2021

- I wrote a book
- Trainer/writer/engineer
  - [http://man7.org/training/](http://man7.org/training/)
- mtk@man7.org, @mkerrisk

![The Linux Programming Interface](image)
Time is short

- Normally, I would spend several hours on this topic
- Many details left out, but I hope to convey the big picture
- We’ll go fast
Outline

1 Introduction 3

2 Namespaces 7

3 An example: UTS namespaces 11

4 Namespaces commands 15

5 Namespaces demonstration (UTS namespaces) 20

6 Some background: capabilities 24

7 User namespaces overview 29

8 User namespaces: UID and GID mappings 33

9 User namespaces and capabilities 40

10 Use cases and further information 51

11 PS: when does a process have capabilities in a user NS? 56

12 PS: a few more details 61
Before looking specifically at user namespaces, what is a namespace (NS) more generally?

A namespace “wraps” some global system resource to provide resource isolation

Linux supports multiple NS types
  - Eight currently, and counting...
Each NS isolates some kind of resource(s)

- Each NS type isolates some kind of resource(s):
  - **UTS** NSs: isolate system identifiers (e.g., hostname)
  - **Mount** NSs: isolate mount point list
  - **IPC** NSs: isolate interprocess communication resources
  - **PID** NSs: isolate PID number space
  - **Network** NSs: isolate NW resources
    - Firewall & routing rules, socket port numbers, /proc/net, /sys/class/net, ...
  - And so on....
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 are unaware of other instances of resource)

This is a bit abstract so far; let’s look at concrete example...
Outline

1 Introduction 3
2 Namespaces 7
3 An example: UTS namespaces 11
4 Namespaces commands 15
5 Namespaces demonstration (UTS namespaces) 20
6 Some background: capabilities 24
7 User namespaces overview 29
8 User namespaces: UID and GID mappings 33
9 User namespaces and capabilities 40
10 Use cases and further information 51
11 PS: when does a process have capabilities in a user NS? 56
12 PS: a few more details 61
UTS namespaces

- UTS NSs are simple, and so provide an easy example
- Isolate two system identifiers returned by `uname(2)`
  - `nodename`: system hostname (set by `sethostname(2)`)
  - `domainname`: NIS domain name (set by `setdomainname(2)`)
- Container configuration scripts might tailor their actions based on these IDs
  - E.g., nodename could be used with DHCP, to obtain IP address for container
- “UTS” comes from `struct utsname` argument of `uname(2)`
  - Structure name derives from “UNIX Timesharing System”
UTS namespaces

- Running system may have multiple UTS NS instances
- Processes within single instance access (get/set) same `nodename` and `domainname`
- Each NS instance has its own `nodename` and `domainname`
  - Changes to `nodename` and `domainname` in one NS instance are invisible to other instances
Each UTS NS contains a set of processes (the circles) which see/modify same hostname (and domain name, not shown)
Some “magic” symlinks

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

<table>
<thead>
<tr>
<th>Symlink Path</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/proc/PID/ns/cgroup</code></td>
<td># Cgroup NS instance</td>
</tr>
<tr>
<td><code>/proc/PID/ns/ipc</code></td>
<td># IPC NS instance</td>
</tr>
<tr>
<td><code>/proc/PID/ns/mnt</code></td>
<td># Mount NS instance</td>
</tr>
<tr>
<td><code>/proc/PID/ns/net</code></td>
<td># Network NS instance</td>
</tr>
<tr>
<td><code>/proc/PID/ns/pid</code></td>
<td># PID NS instance</td>
</tr>
<tr>
<td><code>/proc/PID/ns/time</code></td>
<td># Time NS instance</td>
</tr>
<tr>
<td><code>/proc/PID/ns/user</code></td>
<td># User NS instance</td>
</tr>
<tr>
<td><code>/proc/PID/ns/uts</code></td>
<td># UTS NS instance</td>
</tr>
</tbody>
</table>

- One symlink for each of the NS types
Some “magic” symlinks

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

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

- Content has form: `ns-type : [magic_inode-#]`
  - `(inode-# comes from internally mounted NS filesystem)`

- Various uses for these symlinks, including:
  - If processes show same symlink target, they are in same NS
There are shell commands for working with namespaces...

unshare(1) creates new NSs and executes a command in those NSs:

```bash
unshare [options] [command [arg...]]
```

- command defaults to sh

nsenter(1) steps into already existing NS(s) and executes a command:

```bash
nsenter [options] [command [arg...]]
```

- command defaults to sh
The *unshare(1)* and *nsenter(1)* commands

*unshare(1)* and *nsenter(1)* have options for specifying NS types:

```
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
-T  Create new time NS
-u  Create new UTS NS
-U  Create new user NS
```

```
nsenter [options] [command [arguments]]
-t PID  PID of process whose NSs 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
-T     Enter time NS of target process
-u     Enter UTS NS of target process
-U     Enter user NS of target process
-a     Enter all NSs of target process
```
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>Namespaces</td>
<td>7</td>
</tr>
<tr>
<td>An example: UTS namespaces</td>
<td>11</td>
</tr>
<tr>
<td>Namespaces commands</td>
<td>15</td>
</tr>
<tr>
<td>Namespaces demonstration (UTS namespaces)</td>
<td>20</td>
</tr>
<tr>
<td>Some background: capabilities</td>
<td>24</td>
</tr>
<tr>
<td>User namespaces overview</td>
<td>29</td>
</tr>
<tr>
<td>User namespaces: UID and GID mappings</td>
<td>33</td>
</tr>
<tr>
<td>User namespaces and capabilities</td>
<td>40</td>
</tr>
<tr>
<td>Use cases and further information</td>
<td>51</td>
</tr>
<tr>
<td>PS: when does a process have capabilities in a user NS?</td>
<td>56</td>
</tr>
<tr>
<td>PS: a few more details</td>
<td>61</td>
</tr>
</tbody>
</table>
Start two terminal windows \((sh1, sh2)\) in initial UTS NS

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

sh2$ hostname
bienne
```

In \(sh2\), create new UTS NS, and change hostname

```
$ SUDO_PS1='sh2#' sudo unshare -u bash --norc
sh2# hostname langwied  # Change hostname
sh2# hostname             # Verify change
langwied
```

\textbf{\textit{sudo(8)}} because we need privilege (\texttt{CAP_SYS_ADMIN}) to create a UTS NS

- We set \texttt{SUDO_PS1} so shell has a distinctive prompt. Setting this environment variable causes \textit{sudo(8)} to set \texttt{PS1} for the command that it executes. (\texttt{PS1} defines the prompt displayed by the shell.) The \texttt{bash --norc} option prevents the execution of shell start-up scripts that might modify \texttt{PS1}. 
In *sh1*, verify that hostname is unchanged:

```
sh1$ hostname
bienne
```

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

```
sh1$ readlink /proc/$$/ns/uts
uts:[402653 1838]
```

```
sh2# readlink /proc/$$/ns/uts
uts:[402653 2855]
```

The two shells are in different UTS NSs
• Discover the PID of \textit{sh2}:

\begin{verbatim}
sh2# echo $$
5912
\end{verbatim}

• From \textit{sh1}, use \textit{nsenter(1)} to create a new shell that is in the same NS as \textit{sh2}:

\begin{verbatim}
sh1$ SUDO_PS1='sh3# ' sudo nsenter -t 5912 -u
sh3# hostname
langwied
sh3# readlink /proc/$$/ns/uts
uts:[4026532855]
\end{verbatim}

• Comparing the symlink values, we can see that this shell (\textit{sh3#}) is in the second (\textit{sh2#}) UTS NS
Outline

1 Introduction 3
2 Namespaces 7
3 An example: UTS namespaces 11
4 Namespaces commands 15
5 Namespaces demonstration (UTS namespaces) 20
6 Some background: capabilities 24
7 User namespaces overview 29
8 User namespaces: UID and GID mappings 33
9 User namespaces and capabilities 40
10 Use cases and further information 51
11 PS: when does a process have capabilities in a user NS? 56
12 PS: a few more details 61
(Traditional) superuser and set-UID-\textit{root} programs

- We need a brief understanding of capabilities...
- Traditional UNIX privilege model divides users into two groups:
  - \textbf{Normal users}, subject to privilege checking based on UIDs and GIDs
  - \textbf{Superuser} (UID 0) bypasses many of those checks
- Traditional mechanism for giving privilege to unprivileged users is \textbf{set-UID-\textit{root} program}

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

- When executed, \textbf{process assumes UID of file owner}
  - \(\Rightarrow\) process gains privileges of superuser
- Powerful... but dangerous
Coarse granularity of traditional privilege model is a problem:
  E.g., say we want to give a program the 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
Capabilities: **divide power of superuser into small pieces**
- 41 capabilities as at Linux 6.4 (see `capabilities(7)``

Examples:
- **CAP_DAC_OVERRIDE**: bypass all file permission checks
- **CAP_SYS_ADMIN**: do (too) many different sysadmin tasks
- **CAP_SYS_TIME**: change system time

Instead of set-UID-*root* programs, have programs with one/a few attached capabilities
- Attached using `setcap(8)`
- When program is executed ⇒ process gets those capabilities
- Program is **weaker** than set-UID-*root* program
  - ⇒ **less dangerous if compromised**
Summary:
- Processes can have capabilities (subset of power of root)
- Programs can have attached capabilities, which are given to processes that executes those programs
- Privileged programs/processes using capabilities are less dangerous if compromised
Outline

1 Introduction 3
2 Namespaces 7
3 An example: UTS namespaces 11
4 Namespaces commands 15
5 Namespaces demonstration (UTS namespaces) 20
6 Some background: capabilities 24
7 User namespaces overview 29
8 User namespaces: UID and GID mappings 33
9 User namespaces and capabilities 40
10 Use cases and further information 51
11 PS: when does a process have capabilities in a user NS? 56
12 PS: a few more details 61
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 has nonzero UID outside NS, and UID of 0 inside NS
  - Process has **root privileges for operations inside user NS**
    - Understanding what that means is our goal...
Relationships between user namespaces

- User NSs have a **hierarchical relationship**:
  - Each user NS (except initial user NS) has a parent user NS

- **Parent of a user NS** $==$ user NS of process that created this user NS

- Parental relationship determines some rules about how capabilities work
  - (End slides)
The first process in a new user NS has *root* privileges

- When a new user NS is created, first process in NS has **all** capabilities
  - Creation is done using `unshare(1)`, `clone(2)`, or `unshare(2)`
- That process has superuser powers!
- ... but only inside the user NS
Outline

1 Introduction 3
2 Namespaces 7
3 An example: UTS namespaces 11
4 Namespaces commands 15
5 Namespaces demonstration (UTS namespaces) 20
6 Some background: capabilities 24
7 User namespaces overview 29
8 User namespaces: UID and GID mappings 33
9 User namespaces and capabilities 40
10 Use cases and further information 51
11 PS: when does a process have capabilities in a user NS? 56
12 PS: a few more details 61
One of first steps after creating a user NS is to define **UID and GID mappings** for NS

Defined by writing to 2 files: `/proc/PID/uid_map` and `/proc/PID/gid_map`

For security reasons, there are **many rules** governing:
- **How** / **when** files may be updated
- **Who** can update the files
- Way too many details to cover here...
  - See `user_namespaces(7)`
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
```

- I.e., UID 0 inside NS maps to unprivileged UID in outer NS
Example: creating a user NS with “root” mappings

- `unshare -U -r` creates user NS with root mappings
- 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
```

- ($$ is PID of the shell)
Example: creating a user NS with “root” mappings

- Examine credentials of new shell:

  ```
  uns2$ id
  uid=0(root) gid=0(root) groups=0(root) ...
  ```

- Examine capabilities of new shell:

  ```
  uns2$ grep -E 'CapPrm|CapEff' /proc/$$/status
  CapPrm: 000001fffffffffffff # Hex bit mask
  CapEff: 000001fffffffffffff
  ```

  - 0xffffffffffffffff is bit mask with all capability bits set
  - `getpcaps` gives same info more readably:

    ```
    uns2$ getpcaps $$
    21135: =ep
    ```

    - `=ep` means all permitted and effective capabilities
Example: creating a user NS with “root” mappings

- Discover PID of shell in new user NS:

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

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

  ```bash
  $ ps -o 'uid,gid,pid' 21135
  UID    GID    PID
  1000    1000  21135
  ```
I’m superuser, right?

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

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

- What went wrong?
  - After all, that shell has **all** capabilities
  - 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...
Kernel grants all capabilities to initial process in new user NS of capabilities.

But, those capabilities are available only for operations on objects governed by the new user NS.

But what does that 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
  - Mount: mount list
  - Network: IP routing tables, port numbers, /proc/net, ...

Adding to this: **each nonuser NS instance is owned by some user NS instance**

- When creating new nonuser NS, kernel marks that NS as owned by **user NS of process creating the new NS**
- If a process operates on resources governed by nonuser NS:
  - Permission checks are done according to that process’s capabilities in user NS that owns the nonuser NS
To illustrate, let’s look at set-up resulting from command:

```
unshare -Ur -u <prog>
```

(Create process running `prog` in new user NS with root mappings + new UTS NS)
User namespaces and capabilities—an example

- X is in a new UTS NS, which is owned by new user NS
- X is in initial instance of all other NS types (e.g., NW NS)

©2023, Michael Kerrisk @mkerrisk

Understanding user namespaces 44 / 64
Suppose X tries to change hostname (CAP_SYS_ADMIN)

- X is in second UTS NS
- Privilege checked according to X’s capabilities in user NS that owns that UTS NS \( \Rightarrow \) succeeds (X has capabilities in that user NS)
Suppose X tries to turn NW device up/down (\texttt{CAP\_NET\_ADMIN})

- X is in initial \textbf{network} NS

- Privilege checked according to X’s capabilities in user NS that owns network NS \(\Rightarrow\) attempt fails (no capabilities in initial user NS)
Containers and namespaces

- “Superuser” process in a container has **root power over resources governed by non-user NSs owned by container’s user NS**
- And does **not** have privilege in outside user NS
  - (E.g., can’t change mounts seen by processes outside container)
Discovering namespace relationships

- There are APIs to discover:
  - Parental relationships between user NSs
  - Ownership relationships between user NSs
  - See `ioctl_ns(2)`

- Code example: `namespaces/namespaces_of.go`
  - Shows NS memberships of specified processes, in context of user NS hierarchy
  - Better example: https://github.com/TheDiveO/lxkns
Demo: effect of capabilities in a user NS

Create a shell in new user and UTS NSs:

```
$ unshare -Ur -u bash
# getpcaps $$
353: =ep
# Shell has all capabilities in its user NS
```

Since this shell has all capabilities in user NS that owns its UTS NS, we can change hostname:

```
# hostname
bienne
# hostname langwied
# hostname
langwied
```

But, this shell is in a network NS owned by initial user NS, and so can’t turn a NW device down:

```
# ip link set dev lo down
RTNETLINK answers: Operation not permitted
```
Discovering namespace relationships

- Inspect with namespaces/namespaces_of.go program:

```bash
$ echo $$ # PID of a shell in initial user NS
327
$ go run namespaces_of.go --namespaces=net,uts 327 353
user {4 4026531837} <UID: 0>
  [ 327 ]
  net {4 4026532008}
    [ 327 353 ]
  uts {4 4026531838}
    [ 327 ]
  user {4 4026532760} <UID: 1000>
    [ 353 ]
    uts {4 4026532761}
      [ 353 ]
```

- Indentation indicates user NS ownership / parental relationship between user NSs
- Shells are in same network NS, but different UTS+user NSs
- Second UTS NS is owned by second user NS
- {...} shows unique NS identifier (device ID + inode #)
Applications of user namespaces

User NSs permit many interesting applications; for example:

- **Running Linux containers without *root* privileges**
  - Docker, LXC
- **Chrome-style **sandboxing** of browser renderer process**
  - Sandbox renderer process, because it is an attack target
  - Formerly, use of set-UID-*root* helpers was required
    - [https://chromium.googlesource.com/chromium/src/+/master/docs/design/sandbox.md](https://chromium.googlesource.com/chromium/src/+/master/docs/design/sandbox.md)
- **User NS with single UID identity mapping** $\Rightarrow$ no superuser possible!
  - E.g., `uid_map: 1000 1000 1`
Applications of user namespaces

- **Firejail**: namespaces + seccomp + capabilities for generalized, **simplified sandboxing** of any application
  - Predefined sandboxing profiles exist for 1000+ common apps (Chrome, LibreOffice, VLC, *tar*, *vim*, *emacs*, ...

  - [https://firejail.wordpress.com/](https://firejail.wordpress.com/), [https://lwn.net/Articles/671534/](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/](http://flatpak.org/), [https://lwn.net/Articles/694291/](https://lwn.net/Articles/694291/)

- Ubuntu *Snap* is a similar concept
Further information

- My LWN.net article series *Namespaces in operation*
  - https://lwn.net/Articles/531114/
  - Many example programs and shell sessions...

- Manual pages:
  - `namespaces(7), user_namespaces(7)`, etc.
  - `unshare(1), nsenter(1)`
  - `capabilities(7)`
  - `clone(2), unshare(2), setns(2), ioctl_ns(2)`

- “Linux containers in 500 lines of code”
  - (But note: uses cgroups v1)
Thanks!

Michael Kerrisk, Trainer and Consultant
http://man7.org/training/

mtk@man7.org  @mkerrisk

Slides at http://man7.org/conf/
Source code at http://man7.org/tlpi/code/
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
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
- (The namespaces/ns_capable.c program encapsulates the rules shown on next slide—it answers the question, does process P have capabilities in namespace X?)
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

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

Any process in parent user NS that has 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, process also has capabilities in all descendant user NSs
Outline

1 Introduction 3
2 Namespaces 7
3 An example: UTS namespaces 11
4 Namespaces commands 15
5 Namespaces demonstration (UTS namespaces) 20
6 Some background: capabilities 24
7 User namespaces overview 29
8 User namespaces: UID and GID mappings 33
9 User namespaces and capabilities 40
10 Use cases and further information 51
11 PS: when does a process have capabilities in a user NS? 56
12 PS: a few more details 61
Earlier, we noted that `CAP_SYS_ADMIN` is needed to create nonuser NSs

So, why can unprivileged user do the following?

$ unshare -U -u -r bash

Can do this, because kernel first creates user NS, giving process 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 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
  - There is a chain of user NSs starting at NS of process and going back to initial NS
  - Examining the mappings in this chain allows kernel to know "true" UID and GID of a process
  - Same principle for checks on other resources that have UID+GID owner
    - E.g., various IPC objects