Understanding user namespaces

Michael Kerrisk, man7.org © 2019

mtk@man7.org @mkerrisk #man7training

4 September 2019, Kongsberg
<table>
<thead>
<tr>
<th></th>
<th>Outline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>Some background: capabilities</td>
</tr>
<tr>
<td>3</td>
<td>Namespaces</td>
</tr>
<tr>
<td>4</td>
<td>Namespaces example: UTS namespaces</td>
</tr>
<tr>
<td>5</td>
<td>Namespace APIs and commands</td>
</tr>
<tr>
<td>6</td>
<td>User namespaces overview</td>
</tr>
<tr>
<td>7</td>
<td>User namespaces: UID and GID mappings</td>
</tr>
<tr>
<td>8</td>
<td>User namespaces and capabilities</td>
</tr>
<tr>
<td>9</td>
<td>Use cases</td>
</tr>
<tr>
<td>10</td>
<td>PS: a few more details</td>
</tr>
<tr>
<td>11</td>
<td>PS: when does a process have capabilities in a user NS?</td>
</tr>
<tr>
<td>12</td>
<td>PS: security issues</td>
</tr>
<tr>
<td>Chapter</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>Some background: capabilities</td>
</tr>
<tr>
<td>3</td>
<td>Namespaces</td>
</tr>
<tr>
<td>4</td>
<td>Namespaces example: UTS namespaces</td>
</tr>
<tr>
<td>5</td>
<td>Namespace APIs and commands</td>
</tr>
<tr>
<td>6</td>
<td>User namespaces overview</td>
</tr>
<tr>
<td>7</td>
<td>User namespaces: UID and GID mappings</td>
</tr>
<tr>
<td>8</td>
<td>User namespaces and capabilities</td>
</tr>
<tr>
<td>9</td>
<td>Use cases</td>
</tr>
<tr>
<td>10</td>
<td>PS: a few more details</td>
</tr>
<tr>
<td>11</td>
<td>PS: when does a process have capabilities in a user NS?</td>
</tr>
<tr>
<td>12</td>
<td>PS: security issues</td>
</tr>
</tbody>
</table>
Who am I?

- Maintainer to Linux *man-pages* project since 2004
  - \(\approx 1050\) pages, mainly for system calls & C library functions
    - (I wrote a lot of those pages…)
- Author of a book on the Linux programming interface
- **Trainer** / writer / engineer
  - [http://man7.org/training/](http://man7.org/training/)
- Email: mtk@man7.org
  - Twitter: @mkerrisk
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
  - ⚠️ Save questions until the end please
- ⚠️ My presumption: you attended the preceding presentation (“Linux namespaces”)
  - Some slides from that presentation are duplicated here
    - Just for completeness (in case you didn’t attend that presentation)
    - (But, I’ll skip those slides)
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Some background: capabilities</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>Namespaces</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>Namespaces example: UTS namespaces</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>Namespace APIs and commands</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>User namespaces overview</td>
<td>29</td>
</tr>
<tr>
<td>7</td>
<td>User namespaces: UID and GID mappings</td>
<td>34</td>
</tr>
<tr>
<td>8</td>
<td>User namespaces and capabilities</td>
<td>41</td>
</tr>
<tr>
<td>9</td>
<td>Use cases</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>PS: a few more details</td>
<td>56</td>
</tr>
<tr>
<td>11</td>
<td>PS: when does a process have capabilities in a user NS?</td>
<td>60</td>
</tr>
<tr>
<td>12</td>
<td>PS: security issues</td>
<td>65</td>
</tr>
</tbody>
</table>
Traditional UNIX privilege model divides users into two groups:

- **Normal users**, subject to privilege checking based on UIDs and GIDs
- **Superuser** (UID 0) bypasses many of those checks

Traditional mechanism for giving privilege to unprivileged users is **set-UID-root program**

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

- When executed, **process assumes UID of file owner**
  - process gains privileges of superuser
- Powerful, but dangerous
Traditional UNIX privilege model divides users into two groups:

- **Normal users**, subject to privilege checking based on UIDs and GIDs
- **Superuser** (UID 0) bypasses many of those checks

Traditional mechanism for giving privilege to unprivileged users is **set-UID-root program**

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

- When executed, **process assumes UID of file owner**
  - ⇒ 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 a program the power to change system time
    - Must also give it power to do everything else root can do
  - $\Rightarrow$ No limit on possible damage if program is compromised

- Capabilities are an attempt to solve this problem
The traditional privilege model is a problem

- 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
  
  \[ \Rightarrow \textbf{No limit on possible damage} \text{ if program is compromised} \]

- **Capabilities** are an attempt to solve this problem
Capabilities: divide power of superuser into small pieces

- 38 capabilities as at Linux 5.3 (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
- Program is weaker than set-UID-`root` program
  ⇒ less dangerous if compromised
Background: capabilities

- Capabilities: divide power of superuser into small pieces
  - 38 capabilities as at Linux 5.3 (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
    - Program is weaker than set-UID-root program
      - ⇒ less dangerous if compromised
Capabilities: divide power of superuser into small pieces
- 38 capabilities as at Linux 5.3 (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
- Program is weaker than set-UID-`root` program
  ⇒ less dangerous if compromised
Summary:

- Processes can have capabilities (subset of power of root)
- Files can have attached capabilities, which are given to process that executes program
- Privileged binaries/processes using capabilities are less dangerous if compromised
3 Namespaces

4 Namespaces example: UTS namespaces

5 Namespace APIs and commands

6 User namespaces overview

7 User namespaces: UID and GID mappings

8 User namespaces and capabilities

9 Use cases

10 PS: a few more details

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

12 PS: security issues
A namespace (NS) “wraps” some global system resource to provide resource isolation.

- Linux supports multiple NS types
  - Seven currently, and counting...
Namespaces

- A namespace (NS) “wraps” some global system resource to provide resource isolation
- Linux supports multiple NS types
  - Seven currently, and counting...
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)
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)
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)
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)
Outline

1 Introduction 3
2 Some background: capabilities 6
3 Namespaces 11
4 Namespaces example: UTS namespaces 16
5 Namespace APIs and commands 20
6 User namespaces overview 29
7 User namespaces: UID and GID mappings 34
8 User namespaces and capabilities 41
9 Use cases 50
10 PS: a few more details 56
11 PS: when does a process have capabilities in a user NS? 60
12 PS: security issues 65
UTS namespaces (CLONE_NEWUTS)

- UTS NSs are simplest NS, 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 (CLONE_NEWUTS)

- UTS NSs are simplest NS, 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 (CLONE_NEWUTS)**

- UTS NSs are simplest NS, 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 (CLONE_NEWUTS)

- UTS NSs are simplest NS, 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”
- 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
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
UTS namespaces (CLONE_NEWUTS)

- 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</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/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:

  ```
  $ 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
Programs can use various system calls to work with NSs:

- `clone(2)`: create new (child) 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

There are analogous shell commands:
- `unshare(1)`: create new NS(s) and execute a command in the NS(s)
- `nsenter(1)`: enter existing NS(s) and execute a command
APIs and commands

- Programs can use various system calls to work with NSs:
  - `clone(2)`: create new (child) 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

- There are analogous **shell commands**:
  - `unshare(1)`: create new NS(s) and execute a 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 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
   -u     Enter UTS 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** (nonuser) 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
```

Used *sudo* because we need privilege (**CAP_SYS_ADMIN**) to create a UTS NS
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:[402653 1838]

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

The two shells are in different UTS NSs
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=’sh3# ’ sudo nsenter -t 5912 -u
sh3# hostname
bizarro
sh3# readlink /proc/$$/ns/uts
uts:[4026532855]
```

- Comparing the symlink values, we can see that this shell (*sh3#*) is in the second (*sh2#*) UTS NS
Outline

1 Introduction 3
2 Some background: capabilities 6
3 Namespaces 11
4 Namespaces example: UTS namespaces 16
5 Namespace APIs and commands 20
6 User namespaces overview 29
7 User namespaces: UID and GID mappings 34
8 User namespaces and capabilities 41
9 Use cases 50
10 PS: a few more details 56
11 PS: when does a process have capabilities in a user NS? 60
12 PS: security issues 65
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
    - Understanding what that means is our goal...
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**
  - Understanding what that means is our goal...
User NSs have a **hierarchical relationship**:

- 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 about how capabilities work
  - (End slides)
User NSs have a **hierarchical relationship**:

**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 about how capabilities work

(End slides)
User NSs have a **hierarchical relationship**:  

Parent of a user NS $\Rightarrow$ user NS of process that created this user NS  
- Using `clone(2)`, `unshare(2)`, or `unshare(1)`  

Parental relationship determines some rules about how capabilities work  
- (End slides)
A user namespace hierarchy

Initial user NS
creator eUID: 0
uid_map: 0 0 4294967295
gid_map: 0 0 4294967295

User NS "X"
creator eUID: 1000
uid_map: 0 1000 1
gid_map: 0 1000 1

is child of

User NS "Y"
creator eUID: 1001
uid_map: 0 1001 1
gid_map: 0 1001 1

is child of

User NS "X2"
creator eUID: 1000
uid_map: 0 0 1
gid_map: 0 0 1

is child of

Initial user NS
creator eUID: 0
uid_map: 0 0 4294967295
gid_map: 0 0 4294967295
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.
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
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Some background: capabilities</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>Namespaces</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>Namespaces example: UTS namespaces</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>Namespace APIs and commands</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>User namespaces overview</td>
<td>29</td>
</tr>
<tr>
<td>7</td>
<td>User namespaces: UID and GID mappings</td>
<td>34</td>
</tr>
<tr>
<td>8</td>
<td>User namespaces and capabilities</td>
<td>41</td>
</tr>
<tr>
<td>9</td>
<td>Use cases</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>PS: a few more details</td>
<td>56</td>
</tr>
<tr>
<td>11</td>
<td>PS: when does a process have capabilities in a user NS?</td>
<td>60</td>
</tr>
<tr>
<td>12</td>
<td>PS: security issues</td>
<td>65</td>
</tr>
</tbody>
</table>
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 + restrictions on:
- How/when files may be updated
- Who can update the files
- Way too many details to cover here...
  - See `user_namespaces(7)`
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 + restrictions on:
- How/when files may be updated
- Who can update the files
- Way too many details to cover here...
  - See `user_namespaces(7)`
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 + 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:

<table>
<thead>
<tr>
<th>ID-inside-ns</th>
<th>ID-outside-ns</th>
<th>length</th>
</tr>
</thead>
</table>

  - `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
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
- Create a user NS with root mappings running new shell, and examine map files:

```bash
$ 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

- `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
```
Example: creating a user NS with “root” mappings

- Examine credentials and capabilities of new shell:

```bash
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: 0000003fffffffffff # Hex bit mask
```

- **0x3fffffffffff** is bit mask with all 38 capability bits set
  - `getpcaps` from `libcap` project gives same info more readably
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
  $ grep '\[UG\]id' /proc/21135/status
  Uid: 1000 1000 1000 1000 1000
  Gid: 1000 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...
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...
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

1 Introduction 3
2 Some background: capabilities 6
3 Namespaces 11
4 Namespaces example: UTS namespaces 16
5 Namespace APIs and commands 20
6 User namespaces overview 29
7 User namespaces: UID and GID mappings 34
8 User namespaces and capabilities 41
9 Use cases 50
10 PS: a few more details 56
11 PS: when does a process have capabilities in a user NS? 60
12 PS: security issues 65
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, NIS domain name
  - Mount: set of mount points
  - 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
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
  - Mount: set of mount points
  - 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
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
  - Mount: set of mount points
  - 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

Example scenario; **X was created with**: `unshare -Ur -u <prog>

- X is in new user NS, with root mappings, and has all capabilities
- 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., network NS)
User namespaces and 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 that user NS)
### User namespaces and 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)
Discovering namespace relationships

- There are APIs to discover parental relationships between user NSs and ownership relationships between user NSs and nonuser NSs
  - See `ioctl_ns(2)`,
    http://blog.man7.org/2016/12/introspecting-namespace-relationships.html
  - Code example: `namespaces/namespaces_of.go`
    - Shows namespace memberships of specified processes, in context of user NS hierarchy
Demo
Discovering namespace relationships

- Commands to replicate scenario shown in previous slides:

```bash
$ echo $$  # PID of a shell in initial user NS
  327
$ unshare -Ur -u sh  # Create new user and UTS NSs
# echo $$  # PID of shell in new NSs
  353
```

- Inspect with `namespaces/namespaces_of.go` program:

```bash
$ go run namespaces_of.go --namespaces=net,uts 327 353
user {3 4026531837} <UID: 0>
  [ 327 ]
  net {3 4026532008}
    [ 327 353 ]
  uts {3 4026531838}
    [ 327 ]
user {3 4026532760} <UID: 1000>
  [ 353 ]
  uts {3 4026532761}
    [ 353 ]
```

- Shells are in same network NS, but different UTS+user NSs
- Second UTS NS is owned by second user NS
Discovering namespace relationships

- Commands to replicate scenario shown in previous slides:

```bash
$ echo $$  # PID of a shell in initial user NS
327
$ unshare -Ur -u sh  # Create new user and UTS NSs
# echo $$  # PID of shell in new NSs
353
```

- Inspect with `namespaces/namespaces_of.go` program:

```bash
$ go run namespaces_of.go --namespaces=net,uts 327 353
user {3 4026531837} <UID: 0>
[ 327 ]
  net {3 4026532008}
    [ 327 353 ]
  uts {3 4026531838}
    [ 327 ]
user {3 4026532760} <UID: 1000>
[ 353 ]
  uts {3 4026532761}
    [ 353 ]
```

- Shells are in same network NS, but different UTS+user NSs
- Second UTS NS is owned by second user NS
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
    - Set-UID-*root* helpers are (were) used to set up sandbox
    - [https://chromium.googlesource.com/chromium/src/+/master/docs/design/sandbox.md](https://chromium.googlesource.com/chromium/src/+/master/docs/design/sandbox.md)
  - 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
    - (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), user_namespaces(7), mount_namespaces(7), pid_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”
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)`, `user_namespaces(7)`, `mount_namespaces(7)`, `pid_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”
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)`, `user_namespaces(7)`, `mount_namespaces(7)`, `pid_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”
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/
Understanding user namespaces

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/
Outline

1 Introduction 3
2 Some background: capabilities 6
3 Namespaces 11
4 Namespaces example: UTS namespaces 16
5 Namespace APIs and commands 20
6 User namespaces overview 29
7 User namespaces: UID and GID mappings 34
8 User namespaces and capabilities 41
9 Use cases 50
10 PS: a few more details 56
11 PS: when does a process have capabilities in a user NS? 60
12 PS: security issues 65
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
```
Earlier, we noted that `CAP_SYS_ADMIN` is needed to create nonuser NSs.

So, why can unprivileged user do this:

```bash
$ 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:

```bash
$ 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
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 a process with UID 0 inside a user NS tries to access a file owned by ("true") UID 0?
- When accessing files, IDs are mapped back to values in the initial user NS.
  - There is a chain of user NSs starting at the NS of the process and going back to the initial NS.
  - Examining the mappings in this chain allows the kernel to know "true" UID and GID of processes in user NSs.
  - Same principle for checks on other resources that have UID+GID owner,
    - E.g., Various IPC objects.
But what about accessing files (and other resources)?

- Suppose UID 1000 is mapped to UID 0 inside a user NS.
- What happens when a process with UID 0 inside a user NS tries to access a file owned by (“true”) UID 0?
- When accessing files, IDs are mapped back to values in the initial user NS:
  - There is a chain of user NSs starting at the NS of the process and going back to the initial NS.
  - Examining the mappings in this chain allows the kernel to know “true” UID and GID of processes in user NSs.
  - Same principle for checks on other resources that have UID+GID owner:
    - E.g., Various IPC objects.

©2019, Michael Kerrisk  @mkerrisk  Understanding user namespaces 59 / 66
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2  Some background: capabilities</td>
<td>6</td>
</tr>
<tr>
<td>3  Namespaces</td>
<td>11</td>
</tr>
<tr>
<td>4  Namespaces example: UTS namespaces</td>
<td>16</td>
</tr>
<tr>
<td>5  Namespace APIs and commands</td>
<td>20</td>
</tr>
<tr>
<td>6  User namespaces overview</td>
<td>29</td>
</tr>
<tr>
<td>7  User namespaces: UID and GID mappings</td>
<td>34</td>
</tr>
<tr>
<td>8  User namespaces and capabilities</td>
<td>41</td>
</tr>
<tr>
<td>9  Use cases</td>
<td>50</td>
</tr>
<tr>
<td>10 PS: a few more details</td>
<td>56</td>
</tr>
<tr>
<td><strong>11 PS: when does a process have capabilities in a user NS?</strong></td>
<td>60</td>
</tr>
<tr>
<td>12 PS: security issues</td>
<td>65</td>
</tr>
</tbody>
</table>
What are the rules that determine the capabilities that a process has in a given user namespace?
User namespaces exist in a hierarchy

Each user namespace has a parent, going back to the initial user namespace.

Parental relationship is established when a user namespace is created:

- Parent of a new user namespace is the user namespace of the process that created the new user namespace.

Parental relationship is significant because it plays a part in determining capabilities a process has in a 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

(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
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

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

(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
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
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
<table>
<thead>
<tr>
<th></th>
<th>Outline</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Some background: capabilities</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>Namespaces</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>Namespaces example: UTS namespaces</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>Namespace APIs and commands</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>User namespaces overview</td>
<td>29</td>
</tr>
<tr>
<td>7</td>
<td>User namespaces: UID and GID mappings</td>
<td>34</td>
</tr>
<tr>
<td>8</td>
<td>User namespaces and capabilities</td>
<td>41</td>
</tr>
<tr>
<td>9</td>
<td>Use cases</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>PS: a few more details</td>
<td>56</td>
</tr>
<tr>
<td>11</td>
<td>PS: when does a process have capabilities in a user NS?</td>
<td>60</td>
</tr>
<tr>
<td>12</td>
<td>PS: security issues</td>
<td>65</td>
</tr>
</tbody>
</table>
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

⇒ unprivileged user who gains all capabilities in child 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
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 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
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 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