Linux Capabilities and Namespaces

User Namespaces

Michael Kerrisk, man7.org © 2026

mtk@man7.org

January 2026

Outline

Rev: # d49d0b54be95

Outline

Preamble

- For even more detail than presented here, see my articles:
  - *Namespaces in operation, part 5: user namespaces*, `https://lwn.net/Articles/532593/`
  - *Namespaces in operation, part 6: more on user namespaces*, `https://lwn.net/Articles/540087/`
  - ⚠ See my notes in comments section for some updates
- And *user_namespaces(7)* manual page

Introduction

- Milestone release: Linux 3.8 (Feb 2013)
  - User NSs can now be created by unprivileged users...
- 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*
    - We will learn what this means...

man7.org

Relationships between user namespaces

- User NSs have a hierarchical relationship:
  - A user NS can have 0 or more child user NSs
  - Each user NS has parent NS, going back to initial user NS
    - Initial user NS == sole user NS that exists at boot time
  - Maximum nesting depth for user NSs is 32†
    - (To prevent extremely long chains of descent, since these need to be traversed)
  - Parent of a user NS == user NS of process that created this user NS using *clone()* or *unshare()*
- Parental relationship determines some rules about how capabilities work in NSs (later...)

†In other words, 32 nested levels below the initial user namespace, meaning 33 levels in total. However, There is an off-by-one error in the kernel code, so that the maximum number of levels (including the initial user namespace) is actually 34!

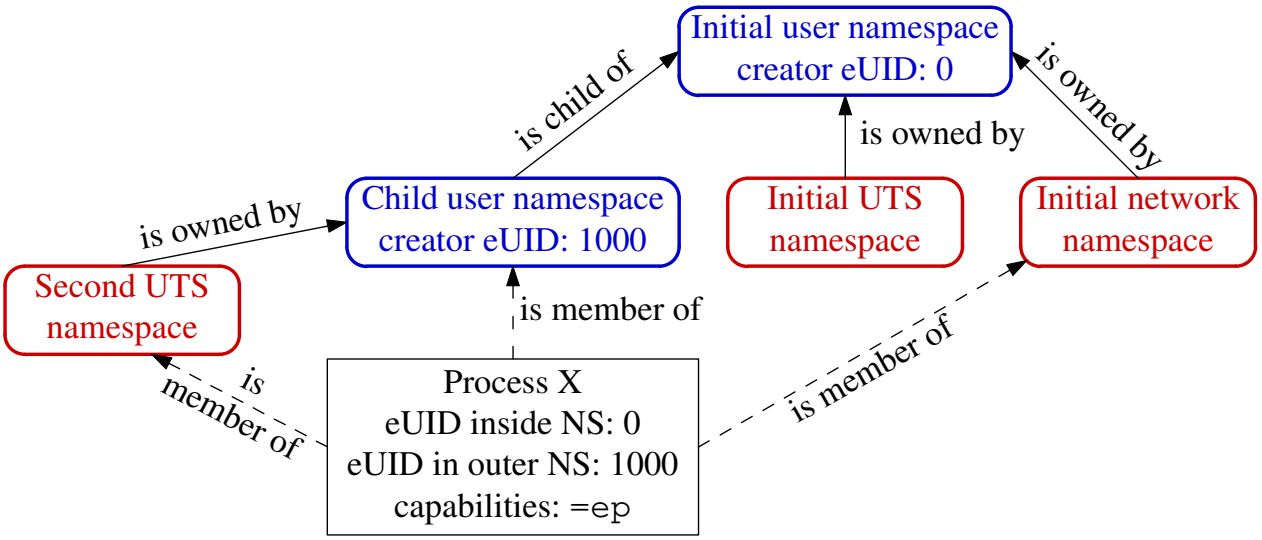
man7.org

"Root privileges inside a user NS"

- What does "*root* privileges in a user NS" 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 mounts
    - Network: IP routing tables, port numbers, `/proc/net`, ...
- What we will see is that:
  - There is an ownership relationship between user NSs and non-user NSs
    - I.e., each non-user NS is "owned" by a particular user NS
  - "*root* privileges in a user NS" == *root* privileges on (only) resources governed by non-user NSs owned by this user NS
    - And on resources associated with descendant user NSs...

man7.org

User namespaces "govern" other namespace types



- Understanding this picture is our ultimate goal...

man7.org

Outline

Creating and joining a user NS

- New user NS is created with `CLONE_NEWUSER` flag
  - *clone()* ⇒ child is made a member of new user NS
  - *unshare()* ⇒ caller is made a member of new user NS
- Can join an existing user NS using *setns()*
  - Process must have `CAP_SYS_ADMIN` capability in target NS
    - (The capability requirement will become clearer later)

User namespaces and capabilities

- A process gains a full set of permitted and effective capabilities in the new/target user NS when:
  - It is the child of *clone()* that creates a new user NS
  - It creates and joins a new user NS using *unshare()*
  - It joins an existing user NS using *setns()*
- But, process has no capabilities in parent/previous user NS
  - ⚠ Even if it was *root* in that NS!

man7.org

Example: `namespaces/demo_userns.c`

```
./demo_userns
```

- (Very) simple user NS demonstration program
- Uses *clone()* to create child in new user NS
- Child displays its UID, GID, and capabilities

man7.org

Example: namespaces/demo_userns.c

```
#define STACK_SIZE (1024 * 1024)

int main(int argc, char *argv[]) {
    char *stack = mmap(..., STACK_SIZE);    /* Allocate memory for
                                               child's stack */
    pid_t pid = clone(childFunc, stack + STACK_SIZE,
                      CLONE_NEWUSER | SIGCHLD, argv[1]);
    printf("PID of child: %ld\n", (long) pid);

    munmap(stack, STACK_SIZE);              /* Deallocate stack */

    waitpid(pid, NULL, 0);
    exit(EXIT_SUCCESS);
}
```

- Use *clone()* to create a child in a new user NS
  - Child will execute *childFunc()*, with argument *argv[1]*
- Printing PID of child is useful for some demos...
- Wait for child to terminate

man7.org

Example: namespaces/demo_userns.c

```
static int childFunc(void *arg) {
    for (;;) {
        printf("eUID = %ld; eGID = %ld; ",
                (long) geteuid(), (long) getegid());

        cap_t caps = cap_get_proc();
        char *str = cap_to_text(caps, NULL);
        printf("capabilities: %s\n", str);
        cap_free(caps);
        cap_free(str);

        if (arg == NULL)
            break;
        sleep(5);
    }
    return 0;
}
```

- Display PID, effective UID + GID, and capabilities
- If *arg* (*argv[1]*) was NULL, break out of loop
- Otherwise, redisplay IDs and capabilities every 5 seconds

man7.org

Example: namespaces/demo_userns.c

```
$ id -u      # Display effective UID of shell process
1000
$ id -g      # Display effective GID of shell process
1000
$ ./demo_userns
eUID = 65534; eGID = 65534; capabilities: =ep
```

Upon running the program, we'll see something like the above

- Program was run from unprivileged user account
- =ep means child process has a full set of permitted and effective capabilities

man7.org

Example: namespaces/demo_userns.c

```
$ id -u      # Display effective UID of shell process
1000
$ id -g      # Display effective GID of shell process
1000
$ ./demo_userns
eUID = 65534; eGID = 65534; capabilities: =ep
```

Displayed UID and GID are "strange"

- System calls such as *geteuid()* and *getegid()* always return credentials as they appear inside user NS where caller resides
- But, no mapping has yet been defined to map IDs outside user NS to IDs inside NS
- ⇒ when a UID is unmapped, system calls return value in /proc/sys/kernel/overflowuid
  - Unmapped GIDs ⇒ /proc/sys/kernel/overflowgid
  - Default value, 65534, chosen to be same as NFS nobody ID

man7.org

Outline

UID and GID mappings

- One of first steps after creating a user NS is to define UID and GID mapping for NS
- Mappings for a user NS are defined by writing to 2 files: `/proc/PID/uid_map` and `/proc/PID/gid_map`
  - Each process in user NS has these files; writing to files of *any* process in the user NS suffices
  - Initially, these files are empty

UID and GID mappings

- Records written to/read from `uid_map` and `gid_map` have this 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

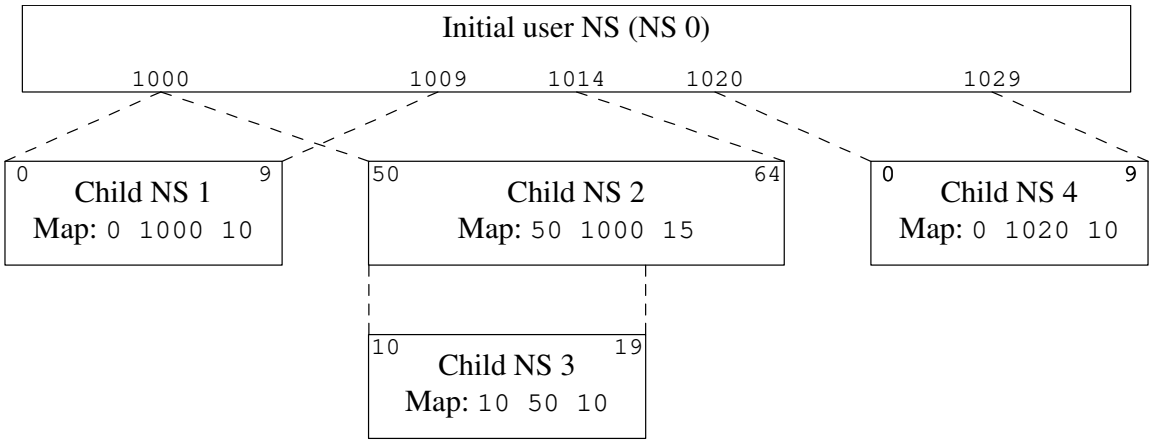
- E.g., following says that IDs 0...9 inside user NS map to IDs 1000...1009 in outside user NS

```
0      1000      10
```

- ⚠ To properly understand *ID-outside-ns*, we must first look at a picture

man7.org

Understanding UID and GID maps



- "What does ID X in namespace Y map to in namespace Z?" means "what is the equivalent ID (if any) in namespace Z?"
- What does ID 5 in NS 1 map to in the initial NS (NS 0)?
- What does ID 5 in NS 1 map to in NS 2 and NS 3?
- What does ID 15 in NS 3 map to in NS 2 and NS 1?
- What does the UID 0 in NS 4 map to in NS 1?

man7.org

Interpretation of *ID-outside-ns*

- ⚠ Interpretation(*) of *ID-outside-ns* depends on whether "opener" and *PID* are in same user NS
  - **"opener"** == process that is opening + reading/writing map file
  - ***PID*** == process whose map file is being opened

(*) Note: contents of `uid_map`/`gid_map` are generated on the fly by the kernel, and can be different in different processes

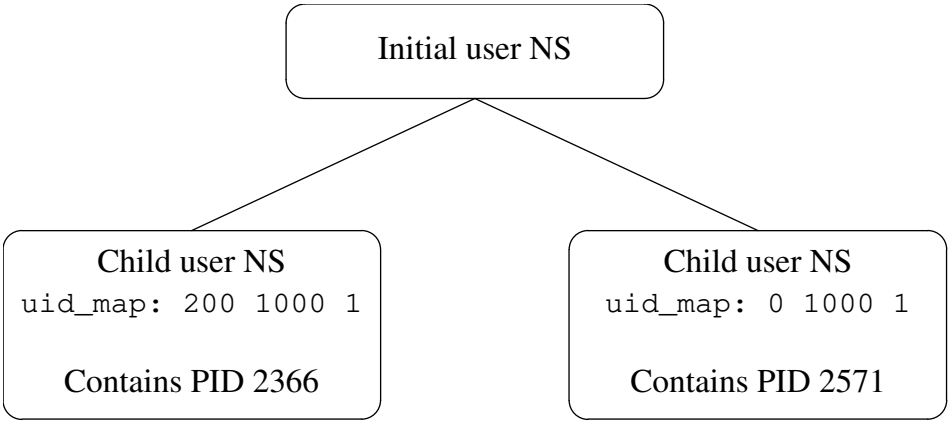
man7.org

Interpretation of *ID-outside-ns*

- If "opener" and *PID* are in **same user NS**:
  - *ID-outside-ns* interpreted as **ID in parent user NS** of *PID*
  - Common case: process is writing its own mapping file
- If "opener" and *PID* are in **different user NSs**:
  - *ID-outside-ns* interpreted as **ID in opener's user NS**
  - Equivalent to previous case, if "opener" is (parent) process that created user NS using *clone()*
- ⚠ Only *ID-outside-ns* is interpreted; *ID-inside-ns* and *length* are always treated literally

man7.org

Quiz: reading /proc/PID/uid_map



- If PID 2366 reads /proc/2571/uid_map, what should it see?
  - 0 200 1
- If PID 2571 reads /proc/2366/uid_map, what should it see?
  - 200 0 1

Example: updating a mapping file

- Let's run demo_userns with an argument, so it loops:

```
$ id -u      # Display user ID of shell
1000
$ id -G      # Display group IDs of shell
1000 10
$ ./demo_userns x
PID of child: 2810
eUID = 65534; eGID = 65534; capabilities: =ep
```

- Then we switch to another terminal window (i.e., a shell process in parent user NS), and write a UID mapping:

```
echo '0 1000 1' > /proc/2810/uid_map
```

- Returning to window where we ran demo_userns, we see:

```
eUID = 0; eGID = 65534; capabilities: =ep
```

Example: updating a mapping file

- But, if we go back to second terminal window, to create a GID mapping, we encounter a problem:

```
$ echo '0 1000 1' > /proc/2810/gid_map
bash: echo: write error: Operation not permitted
```

- There are **(many) rules** governing updates to mapping files
  - Inside the new user NS, user is getting full capabilities
  - **It is critical that capabilities can't leak**
    - I.e., user must not get more privileges than they would otherwise have **outside the NS**

man7.org

Validity requirements for updating mapping files

If any of these rules are violated, *write()* fails with `EINVAL`:

- There is a limit on the number of lines that may be written
  - Since Linux 4.15 (2017): up to 340 lines
    - 340 * 12-byte records: can fit in 4KiB
  - Linux 4.14 and earlier: up to 5 lines
    - An arbitrarily chosen limit that was expected to suffice, but eventually it became an issue for some use cases
    - 5 * 12-byte records: small enough to fit in a 64B cache line
- Each line contains 3 valid numbers, with *length* > 0, and a newline terminator
- The ID ranges specified by the lines may not overlap
  - (Because that would make IDs ambiguous)

man7.org

Permission rules for updating mapping files

If any of these "permission" rules are violated when updating `uid_map` and `gid_map` files, *write()* fails with `EPERM`:

- Each map may be **updated only once**
- Writer must be in target user NS or in parent user NS
- The mapped IDs must have a mapping in parent user NS
- Writer must have following **capability in target user NS**
  - `CAP_SETUID` for `uid_map`
  - `CAP_SETGID` for `gid_map`

man7.org

Permission rules for updating mapping files

As well as preceding rules, one of the following also applies:

- **Either**: writer has `CAP_SETUID` (for `uid_map`) or `CAP_SETGID` (for `gid_map`) **capability in parent user NS**:
  - ⇒ no further restrictions apply (more than one line may be written, and arbitrary UIDs/GIDs may be mapped)
- **Or**: otherwise, all of the following restrictions apply:
  - **Only a single line** may be written to `uid_map` (`gid_map`)
  - That line **maps only the writer's eUID** (eGID)
    - Usual case: we are writing a mapping for eUID/eGID of process that created the NS
  - eUID of writer must match eUID of creator of NS
    - (e**U**ID restriction also applies for `gid_map`)
  - For `gid_map` only: corresponding `/proc/PID/setgroups` file must have been previously updated with string "deny"
    - We revisit reasons later

man7.org

Example: updating a mapping file

- Going back to our earlier example:

```
$ echo '0 1000 1' > /proc/2810/gid_map
bash: echo: write error: Operation not permitted
$ echo 'deny' > /proc/2810/setgroups
$ echo '0 1000 1' > /proc/2810/gid_map
$ cat /proc/2810/gid_map
         0       1000          1
```

  - After writing "deny" to `/proc/PID/setgroups` file, we can update `gid_map`

- Upon returning to window running `demo_userns`, we see:

```
eUID = 0; eGID = 0; capabilities: =ep
```

Outline

Exercises

1. Try replicating the steps shown earlier on your system:
   a. [Ubuntu only] If you are using Ubuntu 24.04 or later, you may need to disable an AppArmor setting that disables the creation of user namespaces by unprivileged users. First, check whether the setting is already turned off (0), using the following command:

```
$ sudo sysctl kernel.apparmor_restrict_unprivileged_userns
```

   If the setting is not off (0), you can turn it off using the following command:

```
$ sudo sysctl -w kernel.apparmor_restrict_unprivileged_userns=0
```

   b. **Use the *id(1)* command to discover your UID and GID; you will need this information for a later step**.
   c. Run the `namespaces/demo_userns.c` program with an argument (any string), so it loops. Verify that the child process has all capabilities.
   d. Inspect (*readlink(1)*) the `/proc/PID/ns/user` symlink for the `demo_userns` child process and compare it with the `/proc/PID/ns/user` symlink for a shell running in the initial user namespace (for the latter, simply open a new shell window on your desktop). You should find that the two processes are in different user namespaces.

[Exercise continues on the next slide]

Exercises

   e. From a shell in the initial user NS, define UID and GID maps for the `demo_userns` child process (i.e., **for the UID and GID that you discovered in the earlier step**). Map the *ID-outside-ns* value for both IDs to IDs of your choice in the inner NS.
      - This step will involve writing to the `uid_map`, `setgroups`, and `gid_map` files in the `/proc/PID` directory.
   f. Verify that the UID and GID displayed by the looping `demo_userns` program have changed.
   g. Try changing the UID map my one more writing to the `uid_map` file. What happens when you try to do this? From a security perspective, why do you think that this happens?

2. What are the contents of the UID and GID maps of a process in the initial user namespace?

```
$ cat /proc/1/uid_map
```

3. The script `namespaces/show_non_init_uid_maps.sh` shows the processes on the system that have a UID map that is different from the *init* process (PID 1). Included in the output of this script are the capabilities of each processes. Run this script to see examples of such processes. As well as noting the UID maps that these processes have, observe the capabilities of these processes.

Outline

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?
  - UID 0 in initial user NS ("true UID 0") is sometimes called **global root**
- 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., System V IPC objects, POSIX IPC objects, UNIX domain sockets

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 correctly 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
  - Maybe some corner case(s) that weren't correctly handled...
  - One early case was discovered and fixed in Linux 3.19
    - *The trouble with dropping groups*, `https://lwn.net/Articles/621612/`
  - Fix required changes to user-space code that updated `gid_map` files
    - E.g., `userns_child_exec.c`

man7.org

setgroups() and /proc/PID/gid_map

- Consider a file with permissions `rw----r--`
  - What do these permissions mean?
  - Process with eUID != file-UID, but with eGID or supplementary GIDs matching file-GID gets no file access
  - Sometimes used to deny file access to class of users
    - A rare use case, but really does occur
- *setgroups(2)* allows a process to drop supplementary GIDs
  - But that's okay: `CAP_SETGID` is required
- However, starting in Linux 3.8, unprivileged user could create user NS where *clone()* child obtained full capabilities
  - Including `CAP_SETGID`!
    - ⇒ *setgroups()* can now be used to add/remove supp. GIDs
  - Unprivileged users now had path to call *setgroups()* and potentially access files that they should not

man7.org

setgroups() and /proc/PID/gid_map: the fix

- A new (writable) `/proc/PID/setgroups` file was added; two values are permitted:
  - "`allow`": processes in user NS of *PID* may call *setgroups()*
    - Process must have `CAP_SETGID` in user NS in order to call *setgroups()*
  - "`deny`": processes in user NS of *PID* may not call *setgroups()*
  - Default value:
    - Default value in initial user NS is "`allow`"
    - New user NS inherits setting from parent user NS
  - *User namespaces and setgroups()*, `https://lwn.net/Articles/626665/`

man7.org

setgroups() and /proc/PID/gid_map: the fix

- Linux 3.19 added two new restrictions:
  - Calling *setgroups()* is not permitted if /proc/[pid]/gid_map has not yet been set
    - Calling *setgroups()* had been possible without a valid map!
  - "deny" must be written to setgroups file before **unprivileged** process can update gid_map
    - Unprivileged process == process that does not have CAP_SETGID capability in **parent** user NS
    - Setting /proc/PID/setgroups to "deny" is irreversible
  - Takes us back to traditional situation: there is no pathway whereby unprivileged processes can call *setgroups()*
- Existence of setgroups file allows backward compatibility
  - I.e., application can discover if it is running on a kernel that imposes these restrictions
- See code in namespaces/userns_child_exec.c

Other security issues

- Other security issues have been uncovered from time to time
- One cause: unprivileged users now have access to system calls/code paths (and bugs in those paths...) formerly available only to superuser
- Examples:
  - *User namespaces + overlayfs = root privileges*, https://lwn.net/Articles/671641/, http://www.halfdog.net/Security/2015/UserNamespaceOverlayfsSetuidWriteExec/
  - http://seclists.org/fulldisclosure/2016/Feb/123
  - https://www.openwall.com/lists/oss-security/2022/01/18/7, https://coder.com/blog/statement-on-the-recent-cve-2022-0185-vulnerability, https://access.redhat.com/security/cve/CVE-2022-0185, https://www.willsroot.io/2022/01/cve-2022-0185.html (nice write-up!)

Other security issues

- Because of concerns that further vulnerabilities may be discovered, some (older) distros:
  - Disabled user NSs (`CONFIG_USER_NS=n`)
  - Patched kernel to disable user NSs by default, and had a `/proc` interface that *root* can use to enable user NSs
    - E.g., some distro releases (Debian, Ubuntu, Arch) added a file, `/proc/sys/kernel/unprivileged_userns_clone`,
    - If file value was 0, creation of user NS by unprivileged users was disallowed
    - Used where admin knows unprivileged users should never need to run containers
  - By now, most distro kernels allow unprivileged user NSs by default

Outline

Combining user namespaces with other namespaces

- Creating other (non-user) NSs requires `CAP_SYS_ADMIN`
- Creating user NSs requires no capabilities
  - And process in new user NS gets full capabilities
- $\Rightarrow$ We can create a user NS, and then create other NS types inside that user NS
  - I.e., two *clone()* or *unshare()* calls
- Actually, we can achieve desired result in one call; e.g.:

```
clone(child_func, stackptr, CLONE_NEWUSER | CLONE_NEWUTS, arg);
// or
unshare(CLONE_NEWUSER | CLONE_NEWUTS);
```

  - Kernel **creates user NS first**, then other NS types
    - And the other NSs are owned by the user NS

man7.org

Outline

Applications of user namespaces

User NSs permit many interesting applications; for example:

- Running Linux containers **without** *root* privileges
  - Docker, LXC, Podman
- Chrome-style sandboxes without set-UID-*root* helpers
  - Chrome browser sandboxes renderer process, since this is a target of attack
  - Formerly, use of set-UID-*root* helpers was required
  - `https://chromium.googlesource.com/chromium/src/+/master/docs/design/sandbox.md`
- User namespace with single UID identity mapping ⇒ no superuser possible!
  - `uid_map: 1000 1000 1`
    - (E.g., Firefox and Chrome browsers use this technique)

Applications of user namespaces

- *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*) (`https://wiki.debian.org/FakeRoot`)

Applications of user namespaces

- Firejail: namespaces + seccomp + capabilities + cgroups for generalized, **simplified sandboxing** of any application
  - Predefined sandboxing profiles exist for 1000+ common apps (Chromium, LibreOffice, VLC, *tar*, *vim*, *emacs*, ...)
  - `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/`
  - Ubuntu *Snap* is a similar concept

man7.org

Outline

Review questions

1. Suppose that there are three child namespaces (A, B, and C) underneath the initial user namespace, as follows:
   - In namespace A, the UID map is "`10 1000 10`", and there is a member process with PID 1111.
   - In namespace B, the UID map is "`50 1000 1`" , and there is a member process with PID 2222.
   - In namespace C, the UID map is "`0 2000 1`", and there is a member process with PID 3333.

   What three numbers will be seen in each of the following cases:
   - Suppose process 1111 reads `/proc/2222/uid_map`. What will it see?
   - Suppose process 2222 reads `/proc/1111/uid_map`. What will it see?
   - Suppose process 1111 reads `/proc/self/uid_map`. What will it see?
   - Suppose process 1111 reads `/proc/3333/uid_map`. What will it see?

This page intentionally blank