Outline

7  User Namespaces 7-1
7.1 Overview of user namespaces 7-3
7.2 Creating and joining a user NS 7-9
7.3 User namespaces: UID and GID mappings 7-17
7.4 User namespaces, execve(), and user ID 0 7-31
7.5 Accessing files; file-related capabilities 7-48
7.6 Security issues 7-55
7.7 Use cases 7-62
7.8 Combining user namespaces with other namespaces 7-68
Outline

7 User Namespaces 7-1
7.1 Overview of user namespaces 7-3
7.2 Creating and joining a user NS 7-9
7.3 User namespaces: UID and GID mappings 7-17
7.4 User namespaces, execve(), and user ID 0 7-31
7.5 Accessing files; file-related capabilities 7-48
7.6 Security issues 7-55
7.7 Use cases 7-62
7.8 Combining user namespaces with other namespaces 7-68

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
- There is also a *user_namespaces*(7) man 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 may have nonzero UID outside NS, and UID of 0 inside NS
  - ⇒ Process has *root* privileges for operations inside user NS
  - We revisit this point in a moment...

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
  - Parent of a user NS == user NS of process that created this user NS using *clone()* or *unshare()*
- Parental relationship determines some rules about operations that can be performed on a (child) user NS (later...)
- *ioctl(fd, NS_GET_PARENT)* can be used to discover parental relationship
  - Since Linux 4.9; see *ioctl_ns(2)* and http://blog.man7.org/2016/12/introspecting-namespace-relationships.html
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 mount points
  - 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...

Understanding this picture is our ultimate goal...
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!

Example: `namespaces/demo_usersns.c`

```
./demo_usersns
```

- (Very) simple user NS demonstration program
- Uses `clone()` to create child in new user NS
- Child displays its UID, GID, and capabilities
Example: namespaces/demo_userns.c

```c
#define STACK_SIZE (1024 * 1024)

int main(int argc, char *argv[]) {
    pid_t pid;
    char *stack = mmap(NULL, STACK_SIZE, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS | MAP_STACK, -1, 0);
    pid = clone(childFunc, stack + STACK_SIZE, CLONE_NEWUSER | SIGCHLD, argv[1]);
    printf("PID of child: %ld\n", (long) pid);
    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

Example: namespaces/demo_userns.c

```c
static int childFunc(void *arg) {
    cap_t caps;
    char *str;
    for (;;) {
        printf("eUID = %ld; eGID = %ld; ",
               (long) geteuid(), (long) getegid());
        caps = cap_get_proc();
        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
Upon running the program, we’ll see something like the above:

- Program was run from unprivileged user account
- \texttt{=ep} means child process has a full set of permitted and effective capabilities
  - If \texttt{libcap} is not aware of all capability numbers supported by kernel, displayed capability sets may be more verbose

Displayed UID and GID are “strange”:

- System calls such as \texttt{geteuid()} and \texttt{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
  - \textbf{⇒} when a UID is unmapped, system calls return value in \texttt{/proc/sys/kernel/overflowuid} (default value: 65534)
    - Unmapped GIDs \textbf{⇒} \texttt{/proc/sys/kernel/overflowgid}
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
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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td>10</td>
</tr>
</tbody>
</table>

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

---

"What does ID X in namespace Y map to in namespace Z?" means "what is the equivalent ID (if any) in namespace Z?"

- What do IDs 0 and 5 in NS 1 map to in each of the other NSs?
- What does ID 15 in NS 3 map to in each of the other NSs?
- What does ID 64 in NS 2 map to in NS 3?
Interpretation of **ID-outside-ns**

- **⚠️** Interpretation of **ID-outside-ns** depends on whether process opening `uid_map/gid_map` is in same NS as **PID**
  - NB: contents of `uid_map/gid_map` are generated on the fly by the kernel, and can be different in different processes
- 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()`
- (Above rules make sense, when we consider how these two cases could be rationally conceived)

---

**Quiz: reading /proc/PID/uid_map**

![Diagram](image)

- 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
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
```

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 permissions than they would otherwise have **outside the namespace**
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
  - Linux 4.14 and earlier: between 1 and 5 lines
    - An arbitrarily chosen limit that was expected to suffice
    - 5 * 12-byte records: small enough to fit in a 64B cache line
  - Since Linux 4.15: between 1 and 340 lines
    - The limit of 5 was in a few cases becoming a hindrance
    - 340 * 12-byte records: can fit in 4KiB
- Each line contains 3 valid numbers, with `length > 0`, and a newline terminator
- The ID ranges specified by the lines may not overlap

Permission rules for updating mapping files

Violation of any of these “permission” rules when updating `uid_map` and `gid_map` files results in `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`
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:
  - \( \Rightarrow \) 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
    - (eUID 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

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_usersns`, we see:

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