User Namespaces and Capabilities

Michael Kerrisk, man7.org © 2023
March 2023
mtk@man7.org

Outline

10 User Namespaces and Capabilities 10-1
10.1 User namespaces and capabilities 10-3
10.2 What does it mean to be superuser in a namespace? 10-22
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:
  - clone(): parent of new user NS is NS of caller of clone()
  - unshare(): parent of new user NS is caller’s previous NS
- Parental relationship is significant because it plays a part in determining capabilities a process has in user NS

User namespaces and capabilities

- Whether a process has an effective capability inside a “target” user NS depends on several factors:
  - Whether the capability is present in the process’s effective set
  - Which user NS the process is a member of
  - The process’s effective UID
  - The effective UID of the process that created the target user NS
  - The parental relationship between the process’s user NS and the target user NS
- See also namespaces/ns_capable.c
  - (A program that encapsulates the rules described next)
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. A process in a parent user NS that has same eUID as eUID of creator of user NS has all capabilities in the NS
   - At creation time, kernel records eUID of creator as “owner” of user NS
     - Can discover via `ioctl(fd, NS_GET_OWNER_UID)`
   - By virtue of previous rule, capabilities also propagate into all descendant user NSs

Demonstration of capability rules

Set up following scenario; then both `userns_setns_test` processes will try to join Child namespace 1 using `setns()`
creates a child in a new user NS

Both processes then call `setns()` to attempt to join user NS identified by argument

- `setns()` requires `CAP_SYS_ADMIN` capability in target NS

Open `/proc/PID/ns/user` file specified on command line

Create child in new user NS

- `childFunc()` receives file descriptor as argument

Try to join user NS referred to by `fd` (`test_setns()`)

Wait for child to terminate
Child sleeps briefly, to allow parent’s output to appear first
Child attempts to join user NS referred to by \( fd \)

Display caller’s user NS symlink, credentials, and capabilities
Try to \texttt{setns()}\ into user NS referred to by \( fd \)
On success, again display user NS symlink, credentials, and capabilities
static void display_creds_and_caps(char *msg) {
    printf("%seUID = %ld; eGID = %ld; ", msg,
        (long) geteuid(), (long) getegid());

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

cap_free(caps);
cap_free(s);
}

- Display caller’s credentials and capabilities
  - (Different source file)

On a terminal in initial user NS, we run the following commands:

$ id -u
1000
$ readlink /proc/$$/ns/user
user:[4026531837]
$ PS1='sh2# .:/usersns_child_exec 
    -U -M '0 1000 1' -G '0 1000 1' bash
sh2# echo $$
30623
sh2# id -u
0
sh2# readlink /proc/$$/ns/user
user:[4026532638]

- Show UID and user NS for initial shell
- Start a new shell in a new user NS
  - Show PID of new shell
  - Show UID and user NS of new shell
In a second terminal window, we run our `setns()` test program:

- Results of `readlink()` calls show:
  - Parent `userns_setns_test` process is in initial user NS
  - Child `userns_setns_test` is in another user NS
  - `setns()` in parent succeeded, and parent gained full capabilities as it moved into the user NS
  - `setns()` in child fails; child has no capabilities in target NS
Quiz (who can signal a process in a child user NS?)

Initial user NS
- Process B
  - UID = 1001, caps: =
- Process A
  - UID = 1000, caps: =
- Process X
  - UID = 0, caps: =ep

Child user NS
- uid_map: 5 1000 10
- Process C
  - UID = 5, caps: =ep
- Process D
  - UID = 6, caps: =

- Child user NS was created by a process with UID 1000
- That process (which presumably was not A) had capabilities that allowed it to create a user NS with UID map with \textit{length} > 1
- Process X has all capabilities in initial user NS
- Assume process A and process B have no capabilities in initial user NS
- Assume C was first process in child NS and has all capabilities in NS
- Process D has no capabilities

Sending a signal requires UID match or \texttt{CAP_KILL} capability

To which of B, C, D can process A send a signal?
- Can B send a signal to D? Can D send a signal to B?
- Can process X send a signal to processes C and D?
- Can process C send a signal to A? To B?
- Can C send a signal to D?
Quiz (who can signal a process in a child user NS?)

- A can’t signal B, but can signal C (matching credentials) and D (because A has capabilities in D’s NS)
- B can signal D (matching credentials); likewise, D can signal B
- X can signal C and D (because it has capabilities in parent user NS)
- C can signal A (credential match), but not B
- C can signal D, because it has capabilities in its NS

Exercises

1. As an unprivileged user, start two `sleep` processes, one as the unprivileged user and the other as UID 0:

   ```
   $ id -u
   1000
   $ sleep 1000 &
   $ sudo sleep 2000
   ```

   **As superuser**, create a user namespace with root mappings and run a shell in that namespace:

   ```
   $ SUDO_PS1="ns2# " sudo unshare -U -r bash --norc
   ```

   Setting the `SUDO_PS1` environment variable causes `sudo(8)` to set the `PS1` environment variable for the command that it executes. (`PS1` defines the prompt displayed by the shell.) The `bash --norc` option prevents the execution of shell start-up scripts that might change `PS1`.

   [Exercises continue on next slide]
Exercises

Verify that the shell has a full set of capabilities and a UID map “0 0 1”:

```bash
ns2# egrep 'Cap(Prm|Eff)' /proc/$$/status
ns2# cat /proc/$$/uid_map
```

From this shell, try to kill each of the `sleep` processes started above:

```bash
ns2# ps -o 'pid uid cmd' -C sleep  # Discover 'sleep' PIDs
... ns2# kill -9 <PID-1>
ns2# kill -9 <PID-2>
```

Which of the `kill` commands succeeds? Why?
Kernel grants initial process in new user NS a full set of capabilities

But, those capabilities are available **only for operations on objects governed by the new user NS**
Kernel associates each non-user NS instance with a specific user NS instance
- Each non-user NS is “owned” by a user NS
- When creating a new non-user NS, user NS of the creating process becomes the owner of the new NS

Suppose a process operates on global resources governed by a (non-user) NS:
- Privilege checks are done according to process’s capabilities in user NS that owns the NS

⇒ User NSs can deliver full capabilities inside a user NS without allowing capabilities in outer user NS(s)
- (Barring kernel bugs)

Example scenario; X was created with: `unshare -Ur -u <prog>`
- X is in a new user NS, created with root mappings
- 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)
Suppose X tries to change host name (**CAP_SYS_ADMIN**)

- X is in second **UTS** NS

- Privileges checked according to X’s capabilities in user NS that owns that UTS NS ⇒ succeeds (X has capabilities in user NS)

Suppose X tries to bind to reserved socket port (**CAP_NET_BIND_SERVICE**)

- X is in initial **network** NS

- Privileges checked according to X’s capabilities in user NS that owns network NS ⇒ attempt fails (no capabilities in initial user NS)
“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)

---

**What about resources not governed by namespaces?**

- Some privileged operations relate to resources/features not (yet) governed by any namespace
  - E.g., load kernel modules, raise process nice values

- 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., load/unload kernel modules, raise process nice values

- IOW: to perform these operations, process must have the relevant capability in the **initial** user NS