*Linux Security and Isolation APIs Fundamentals*

# User Namespaces and Capabilities

Michael Kerrisk, man7.org © 2025

January 2025

mtk@man7.org

## Outline

Rev: #8d7fc39ab521

# Outline



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 process's effective set
  - Which user NS the process is a member of
  - The process's effective UID
  - The effective UID of process that created target user NS
  - The parental relationship between process's user NS and target user NS
- See also `namespaces/ns_capable.c`
  - (A program that encapsulates the rules described next)

# Capability rules for user namespaces

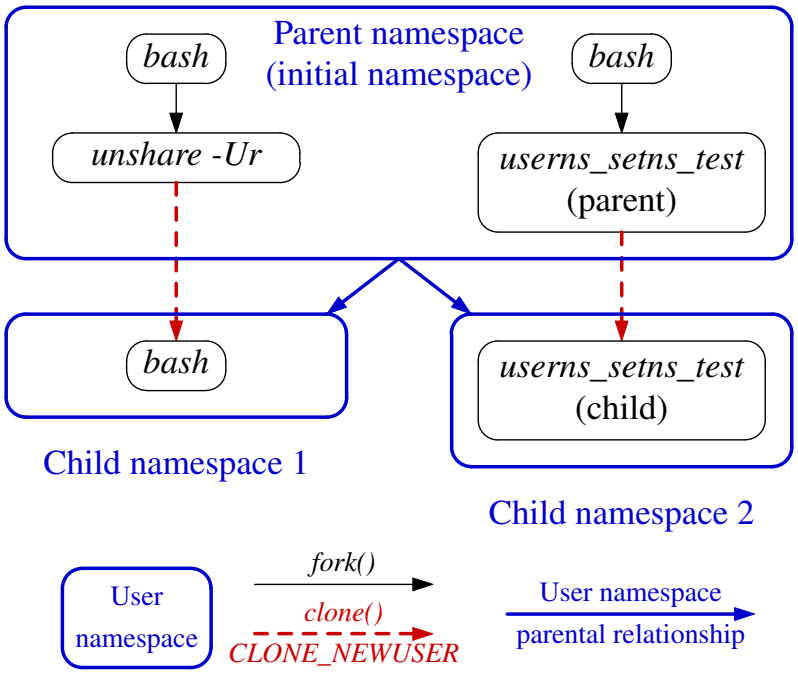
1. A process has a capability in a user NS if:
   - it is a **member of the user NS**, and
   - **capability is present in its effective set**
   - Note: this rule doesn't grant that capability in parent NS
2. A process that has a capability in a user NS **has the capability in all descendant user NSs** 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
   - By virtue of previous rule, process also has capabilities in all descendant user NSs

# Outline



# Demonstration of capability rules

Set up following scenario; then both `userns_setns_test` processes will try to join *Child namespace 1* using *setns()*



## namespaces/userns_setns_test.c

```
./userns_setns_test /proc/PID/ns/user
```

- Creates a child process in a new user NS
- Parent and child then both call *setns()* to attempt to join user NS identified by argument
  - *setns()* requires `CAP_SYS_ADMIN` capability in target NS



## namespaces/userns_setns_test.c

```
int main(int argc, char *argv[]) {
    ...
    long fd = open(argv[1], O_RDONLY);

    pid_t child_pid = clone(childFunc, stack + STACK_SIZE,
                            CLONE_NEWUSER | SIGCHLD, (void *) fd);
    test_setns("parent: ", fd);
    printf("\n");

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

- 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

## namespaces/userns_setns_test.c

```
static int childFunc(void *arg) {
    long fd = (long) arg;

    usleep(100000);
    test_setns("child: ", fd);
    return 0;
}
```

- Child sleeps briefly, to allow parent's output to appear first
- Child attempts to join user NS referred to by *fd*



## namespaces/userns_setns_test.c

```
static void display_symlink(char *pname, char *link) {
    char target[PATH_MAX];
    ssize_t s = readlink(link, target, PATH_MAX);
    printf("%s%s ==> %.*s\n", pname, link, (int) s, target);
}

static void test_setns(char *pname, int fd) {
    display_symlink(pname, "/proc/self/ns/user");
    display_creds_and_caps(pname);
    if (setns(fd, CLONE_NEWUSER) == -1) {
        printf("%s setns() failed: %s\n", pname, strerror(errno));
    } else {
        printf("%s setns() succeeded\n", pname);
        display_symlink(pname, "/proc/self/ns/user");
        display_creds_and_caps(pname);
    }
}
```

- Display caller's user NS symlink, credentials, and capabilities
- Try to *setns()* into user NS referred to by *fd*
- On success, again display user NS symlink, credentials, and capabilities

## namespaces/userns_functions.c

```
 1 static void display_creds_and_caps(char *msg) {
 2     printf("%seUID = %ld;  eGID = %ld;  ", msg,
 3             (long) geteuid(), (long) getegid());
 4
 5     cap_t caps = cap_get_proc();
 6     char *s = cap_to_text(caps, NULL)
 7     printf("capabilities: %s\n", s);
 8
 9     cap_free(caps);
10     cap_free(s);
11 }
```

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



## namespaces/userns_setns_test.c

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

```
$ id -u
1000
$ readlink /proc/$$/ns/user
user:[4026531837]
$ PS1='sh2# ' unshare -Ur bash
sh2# echo $$
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

## namespaces/userns_setns_test.c

```
$ ./userns_setns_test /proc/30623/ns/user
parent: readlink("/proc/self/ns/user") ==> user:[4026531837]
parent: eUID = 1000; eGID = 1000;  capabilities: =
parent: setns() succeeded
parent: eUID = 0;  eGID = 0;  capabilities: =ep

child:  readlink("/proc/self/ns/user") ==> user:[4026532639]
child:  eUID = 65534; eGID = 65534;  capabilities: =ep
child:  setns() failed: Operation not permitted
```

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



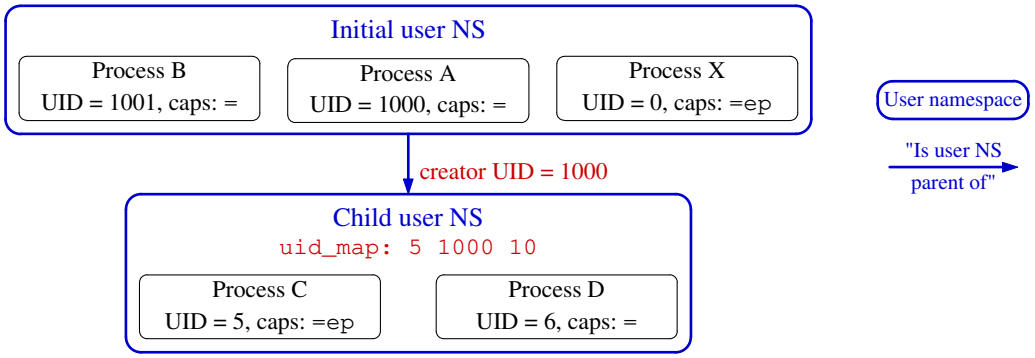
## namespaces/userns_setns_test.c

```
$ ./userns_setns_test /proc/30623/ns/user
parent: readlink("/proc/self/ns/user") ==>
        user:[4026531837]
parent: setns() succeeded
parent: eUID = 0;  eGID = 0;  capabilities: =ep

child:  readlink("/proc/self/ns/user") ==>
        user:[4026532639]
child:  setns() failed: Operation not permitted
```

- *setns()* in child failed:
  - Rule 3: "processes in **parent** user NS that have **same eUID** as creator of user NS have all capabilities in the NS"
  - Parent `userns_setns_test` process was in **parent user NS** of target user NS and so had `CAP_SYS_ADMIN`
  - Child `userns_setns_test` process was in **sibling user NS** and so had no capabilities in target user NS

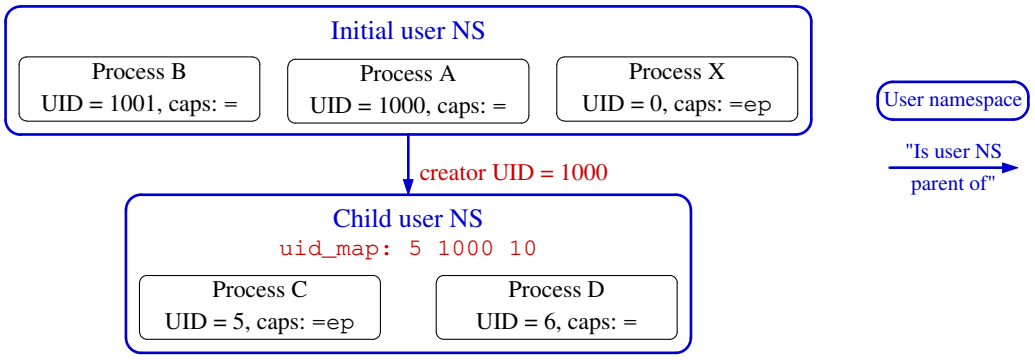
# Quiz (who can signal a process in a child user NS?)




- 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 *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

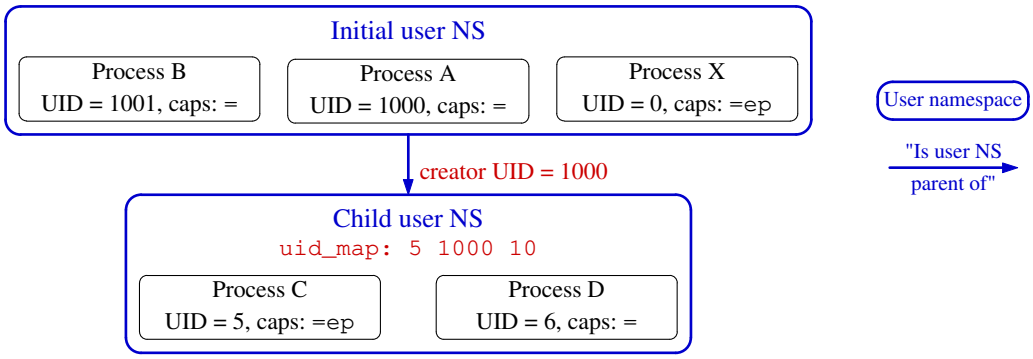


# Quiz (who can signal a process in a child user NS?)




- Sending a signal requires UID match or `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

# Outline



# 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**, in another terminal window use *unshare* to create a user namespace (*–U*) with root mappings (*–r*) and run a shell in that namespace:

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

   - (Root mappings == process's UID and GID in parent NS map to 0 in child NS)
   - 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`" (i.e., UID 0 in the parent namespace maps to UID 0 in the child user namespace):

```
ns2# grep -E 'Cap(Prm|Eff)' /proc/$$/status
ns2# cat /proc/$$/uid_map
```

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

```
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?

# Outline



## User namespaces and capabilities

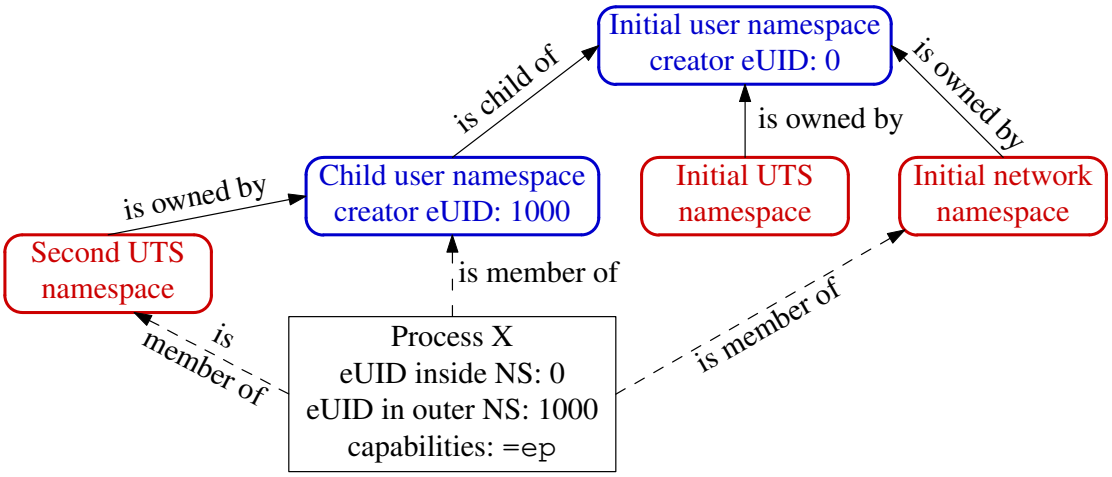
- Kernel grants initial process in new user NS a full set of capabilities
- But, those capabilities are available **only for operations on objects governed by the new user NS**

# User namespaces and capabilities

- **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)

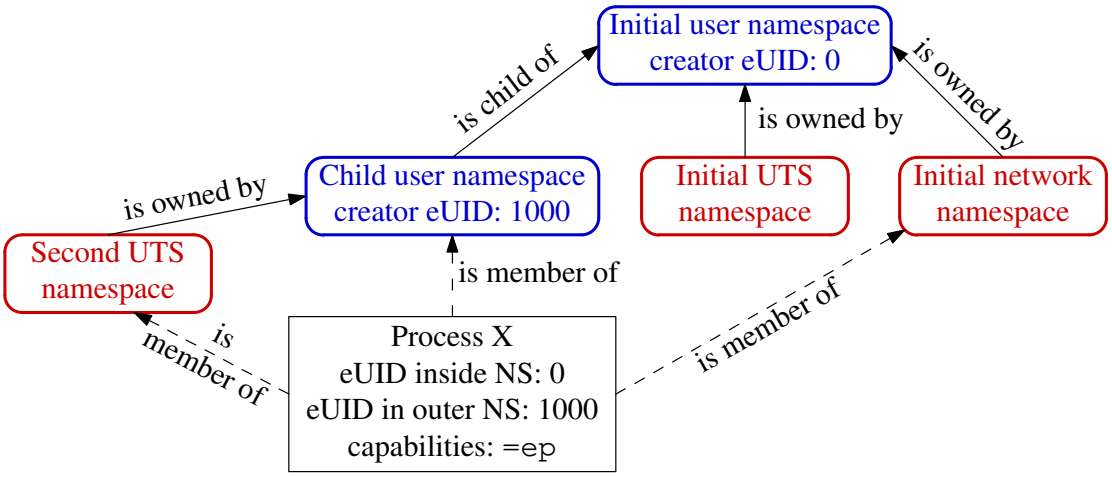


# User namespaces and capabilities–an example




- 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)

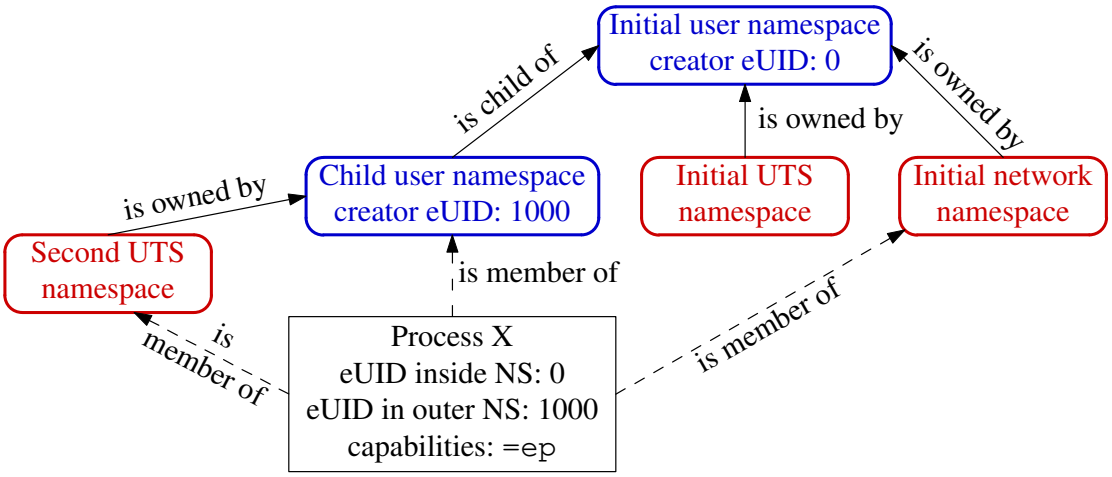
# User namespaces and capabilities–an example




- Suppose X tries to change host name (`CAP_SYS_ADMIN`)
  - E.g., `hostname bienne`
- 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)

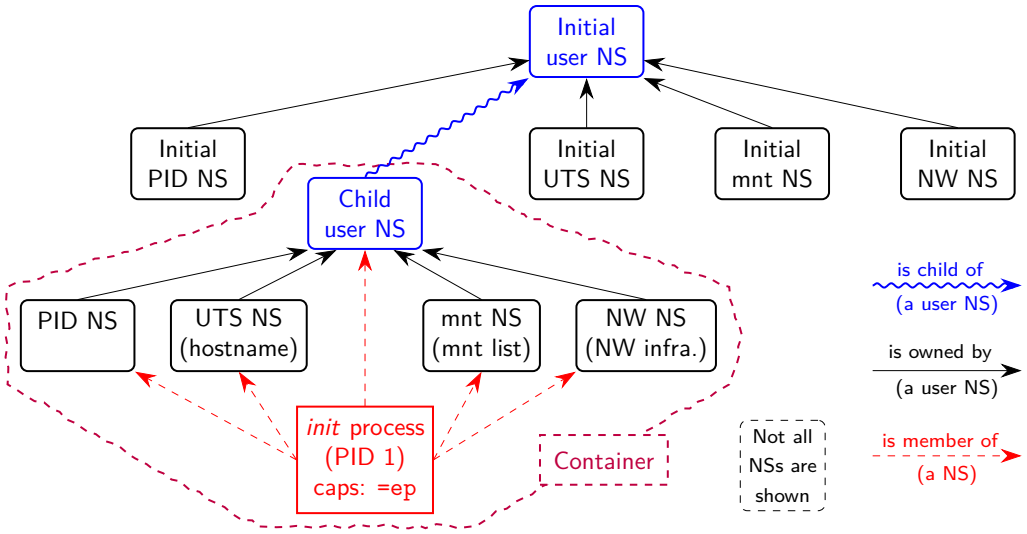


# User namespaces and capabilities–an example




- Suppose X tries to bring network device up/down (`CAP_NET_ADMIN`)
  - E.g., `ip link set dev lo down`
- 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)

# 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)



# Demo: effect of capabilities in a user NS

- Create a shell in new user and UTS NSs:

```
$ unshare -Ur -u bash
# getpcaps $$
929: =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
```

## 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



## Notes