System Programming for Linux Containers

User Namespaces and Capabilities

Michael Kerrisk, man7.org © 2020

mtk@man7.org

February 2020

Outline

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

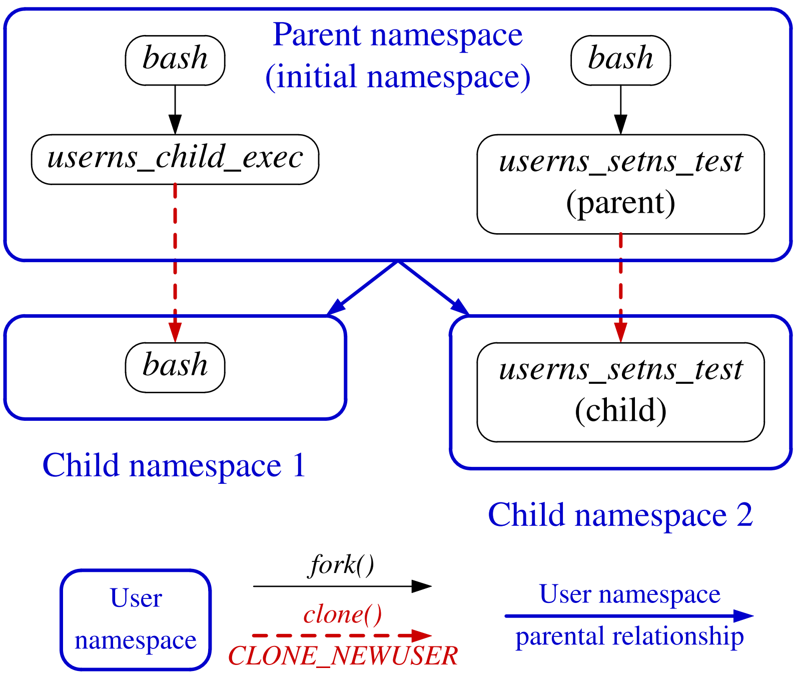
- Whether a process has an effective capability inside a "target" user NS depends on several factors:
  - Whether the capability is present in the process's effective set
  - Which user NS the process is a member of
  - The process's effective UID
  - The effective UID of the process that created the 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. (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" 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()*



namespaces/userns_setns_test.c

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

- Creates a child in a new user NS
- Both processes then call *setns()* to attempt to join user namespace identified by argument
  - *setns()* requires `CAP_SYS_ADMIN` capability in target NS

namespaces/userns_setns_test.c

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

  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");
  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);
  }
}
```

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

namespaces/userns_functions.c

```
static void display_creds_and_caps(char *msg) {
  cap_t caps;
  char *s;

  printf("%seUID = %ld;  eGID = %ld;  ", msg,
          (long) geteuid(), (long) getegid());

  caps = cap_get_proc();
  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)

namespaces/userns_setns_test.c

In one terminal window (in initial user NS), we run the following commands:

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

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
```

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

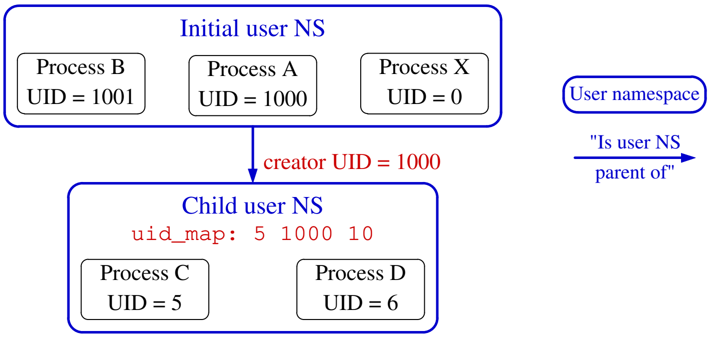
namespaces/userns_setns_test.c

```
$ ./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
```

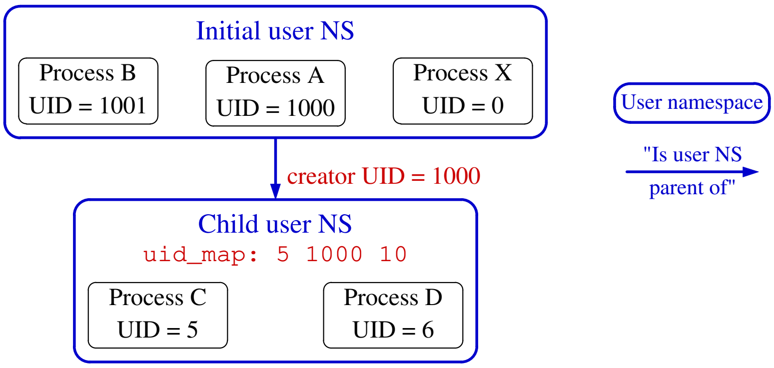
- *setns()* in child failed:
  - Rule 3: "processes in **parent** user NS that have **same eUID** as creator of user NS have all capabilities in the NS"
  - Parent userns_setns_test process was in **parent user NS** of target user NS and so had CAP_SYS_ADMIN
  - Child userns_setns_test process was in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

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



- Sending a signal requires UID match or `CAP_KILL` capability
- Assume A and B have no capabilities in initial user NS
- Assume C was first process in child NS and has all capabilities in NS
- To which of B, C, D can process A send a signal?
- Can process B send a signal to process D?
- Can process X send a signal to processes C and D?
- Can process C send a signal to A? To B? 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 namespace)
- B can signal D (matching credentials)
- X can signal C and D (because it has capabilities in parent user NS)
- C can signal A (credential match), but not B

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:

```
$ PS1="ns2# " sudo unshare -U -r bash --norc
```

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

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

[Exercises continue on next slide]

Exercises

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?

2. Write a program to set up two processes in a child user namespace as in the scenario shown in the previous "Quiz" slide [template: `namespaces/ex.userns_cap_sig_expt.c`]
   - After compiling the program, assign capabilities to the executable as follows:

```
sudo setcap cap_setuid,cap_setgid=pe \
        <program-file>
```

   - While running the program, try sending signals to processes "C" and "D" from a shell in the initial user namespace, in order to verify the answers given for the Quiz.

Outline

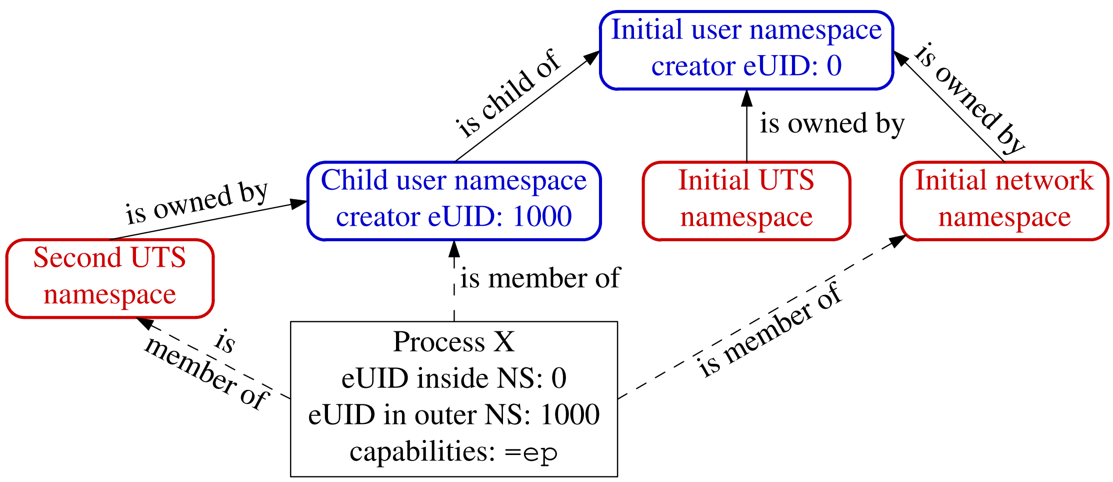
User namespaces and capabilities

- 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

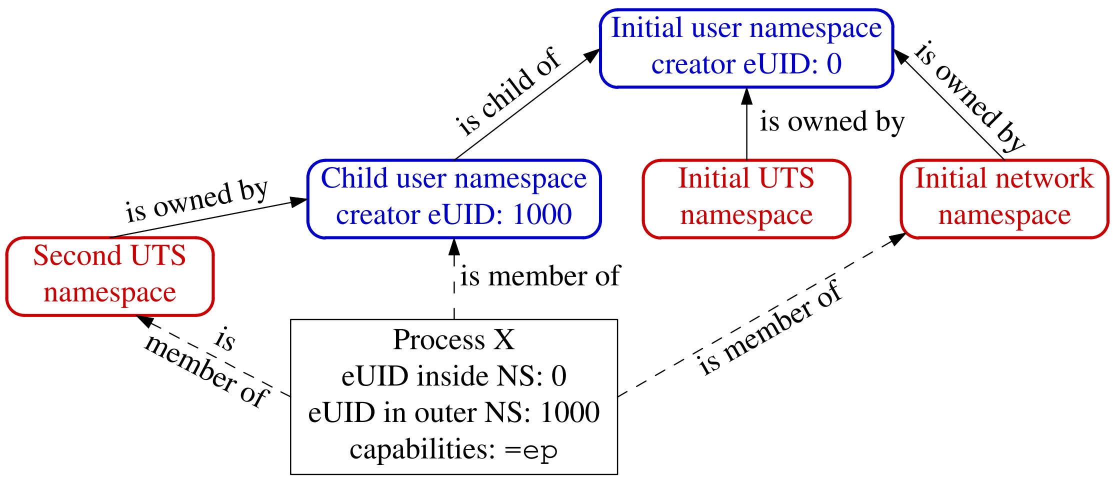
- **Kernel associates each non-user NS instance with a specific user NS instance**
  - When creating new network NS (for example), kernel associates user NS of creating process with new network NS
- Suppose a process operates on global resources governed by new NS:
  - Permission checks are done according to that process's capabilities in user NS that kernel recorded for new NS
- ⇒ User NSs can safely deliver full capabilities inside a NS without allowing users to damage wider system
  - (Barring kernel bugs)

User namespaces and capabilities–an example



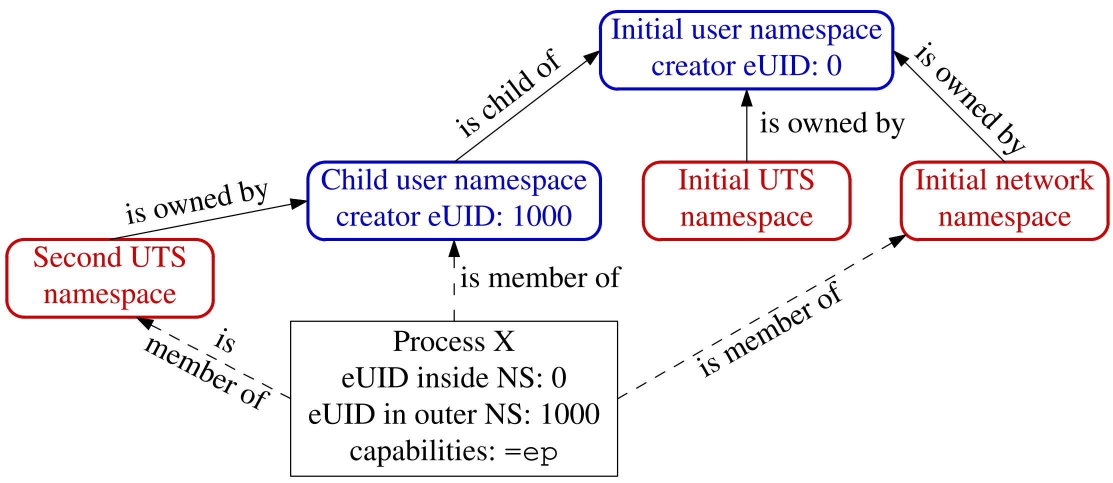
- Example scenario; X was created with: `unshare -Ur -u <prog>`
  - X is in a new user NS, created with root mappings
  - X is in a new UTS NS, which is owned by new user NS
  - X is in initial instance of all other NS types (e.g., network NS)

User namespaces and capabilities–an example



- Suppose X tries to change host name (`CAP_SYS_ADMIN`)
- 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 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

- Recall that there are various *ioctl()* operations that can be used to discover namespace relationships and other info
  - NS_GET_USERNS: get user NS that owns a nonuser NS
  - NS_GET_PARENT: get parent NS (for PID and user NSs)
  - NS_GET_OWNER_UID: get UID of creator of a user NS
  - NS_GET_NSTYPE: get NS type (CLONE_NEW*)
  - Details in *ioctl_ns(2)*
- These operations can be used to build visualization tools for namespaces and their relationships
  - An example: namespaces/namespaces_of.go
    - Scans /proc/PID/ns/* symlinks and uses above *ioctl()* operations to discover namespace relationships

Discovering namespace relationships

- Commands to replicate scenario shown in earlier diagram:

```
$ 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
```

- We can inspect using namespaces/namespaces_of.go
  - Shows namespace memberships of specified processes, in context of user NS hierarchy

Discovering namespace relationships

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

```
$ 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 NSs
  - Second UTS NS is owned by second user NS
  - NS IDs includes device ID (3) from underlying (hidden) NS filesystem
    - As described in *ioctl_ns(2)*, it is the combination of device ID + inode number that uniquely identifies a NS

What about resources not governed by namespaces?

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