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

A user namespace hierarchy

![User Namespace Hierarchy Diagram]

- **Initial user NS**
  - creator eUID: 0
  - uid_map: 0 0 4294967295
  - gid_map: 0 0 4294967295

- **User NS "X"**
  - creator eUID: 1000
  - uid_map: 0 1000 1
  - gid_map: 0 1000 1

- **User NS "Y"**
  - creator eUID: 1001
  - uid_map: 0 1001 1
  - gid_map: 0 1001 1

- **User NS "X2"**
  - creator eUID: 1000
  - uid_map: 0 0 1
  - gid_map: 0 0 1
User namespaces and capabilities

Whether a process has an effective capability inside a 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 user NS
- The parental relationship between user NSs

Capability rules for user namespaces

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

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

(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()`

![Diagram showing process and namespace interactions]

**User namespace**
- Adds parental relationship
- `fork()` and `clone()` with `CLONE_NEWUSER`

**namespaces/userns_setns_test.c**

`. /users_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
int main(int argc, char *argv[]) {
...

    fd = open(argv[1], O_RDONLY);

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

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

static int childFunc(void *arg) {

    long fd = (long) arg;

    usleep(100000);
    test_setns("child: ", fd);
    return 0;
}
static void test_setns(char *pname, int fd) {
    char path[PATH_MAX];

    readlink("/proc/self/ns/user", path, PATH_MAX);
    printf("%s readlink("/proc/self/ns/user") => %s\n", pname, path);

    if (setns(fd, CLONE_NEWUSER) == -1)
        printf("%s setns() failed: %s\n", pname, strerror(errno));
    else {
        printf("%s setns() succeeded\n", pname);
        display_creds_and_caps(pname);
    }
}

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

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

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

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

- Display caller’s credentials and capabilities
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

In a second terminal window, we run our `setns()` test program:

```
$ ./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

---

Linux Security and Isolation APIs  ©2017, Michael Kerrisk  User Namespaces 12-59 §12.5

Linux Security and Isolation APIs  ©2017, Michael Kerrisk  User Namespaces 12-60 §12.5
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?)**

![Diagram of user namespaces](image)

- Sending a signal requires UID match or `CAP_KILL` capability
- To which of B, C, D can process A send a signal?
  - Assume process A has no capabilities in initial user NS
- Can process B send a signal to process D?
- Can process X send a signal to processes C and D?
Quiz (who can signal a process in a child user NS?)

- A 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)

Exercise

1. Write a program to set up two processes in a child user namespace as in the scenario shown in the previous 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,cap_dac_override=pe \<program-file>
     ```
   - While running the program, try sending signals to processes “C” and “D” to verify the answers given in the previous exercise.
Combining user namespaces with other namespaces

- Creating other (non-user) types of NS requires `CAP_SYS_ADMIN`
- Creating user NSs requires no capabilities
  - And first process in user NS gets full capabilities
- We can create a user NS, and then create other NS types inside that user NS
  - I.e., two `clone()` calls (creating child + grandchild)
- Actually, we can achieve desired result in one call:
  ```c
  clone(child_func, stackp,
        CLONE_NEWUSER | CLONE_NEWUTS, arg);
  ```
- Kernel always creates user NS first, then other NS types
- Child created by `clone()` is member of all of the new NSs
clone(child_func, stackp, 
CLONE_NEWUSER | CLONE_NEWUTS, arg);

- Can do equivalent of this call using `userns_child_exec`:

```bash
$ uname -n
antero
$ PS1='sh2$ ' ./userns_child_exec -u -U 
   -M '0 1000 1' -G '0 1000 1' bash
sh2$ cat /proc/$$/status | egrep 'Cap(Inh|Prm|Eff)'
CapInh: 0000000000000000
CapPrm: 0000003fffffffff
CapEff: 0000003fffffffff
```

- `-u` for UTS namespace
- Shell has all capabilities

Example

- Let’s modify hostname (resource governed by UTS NS):

  ```bash
  sh2$ hostname bizarro
  sh2$ uname -n
  bizarro
  ```

- Then switch to another terminal window in initial UTS NS:

  ```bash
  $ uname -n
  antero
  ```

  Hostname was not changed in initial UTS NS
Limiting creation of namespaces

- Files in `/proc/sys/user` define limits on number of NSs of each type that can be created by each user inside a user NS

```
$ ls /proc/sys/user
max_cgroup_namespaces  max_net_namespaces  max_uts_namespaces
max_ipc_namespaces     max_pid_namespaces
max_mnt_namespaces     max_user_namespaces
```

- Since Linux 4.9
- Each user namespace has own limits
  - Limits apply hierarchically to nested user NSs
- Privileged processes can modify the limits
  - Default limits are high
    - So as not to get in way of normally operating applications
- Details can be found in `namespaces(7)`
More on 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
More on 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
  - \( \Rightarrow \) User NSs can safely deliver full capabilities inside a NS without allowing users to damage wider system
    - (Barring kernel bugs)
  - \( ioctl(fd, \textit{NS\_GET\_USERNS}) \) can be used to discover which user NS a non-user NS is associated with
    - Since Linux 4.9; see \textit{ioctl\_ns(2)}

More on capabilities–an example

- Suppose X tries to change host name (\textit{CAP\_SYS\_ADMIN})
- X is in second UTS NS
- Permissions checked according to X’s capabilities in user NS that owns that UTS NS \( \Rightarrow \) succeeds (X has capabilities in user NS)
More on 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)

More on capabilities–another example

- Suppose we create a new child in new user NS with clone(CLONE_NEWUSER)
- Child will have all capabilities in new user NS
  - E.g., it will be able to create other NS types within user NS
- But, child could not (say) change the system hostname
  - Child is still in initial UTS NS
  - It would need capabilities in user NS associated with that UTS NS
    - And does not have them
  - Same principles apply for other namespace types
- Child can create new UTS NS, and change hostname in that NS
  - But that does not affect parent UTS NS
What about resources not governed by namespaces?

- Some privileged operations relate to resources/features not (yet) governed by any namespace
  - E.g., system time, kernel modules

- 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 or load/unload kernel modules