Linux Namespaces and Isolation

Team Emertxe

Contents

Topics

1. Introduction to Linux Namespaces
2. Drawbacks and Security Issues of Set-UID Programs
3. Capabilities in C
4. Linux Capabilities Overview
5. Process Capabilities
6. File Capabilities
7. Viewing and Modifying Capabilities
8. Introduction to 8 Namespaces
9. Namespace Hierarchy
10. UTS Namespace Example

Topics

11. Symlink from /proc
12. Namespace APIs and Commands
13. Demo and Lab Examples
14. User Namespaces: UID and GID Mappings
15. User Namespaces and Capabilities
16. Security Issues with User Namespaces
17. Use Cases of Namespaces
18. Summary
19. Q&A
20. References

Introduction to Namespaces

- Linux namespaces provide process isolation by partitioning kernel resources.
- Introduced in Linux kernel 2.4, expanded in later versions.
- Enables containerization and lightweight virtualization.
- **Key Benefits:** Process isolation, resource control, security enhancement.

Traditional Set-UID Programs

Set-UID programs run with elevated privileges, leading to security risks.

- **Drawbacks:**
  - Privilege escalation attacks.
  - Difficulty in fine-grained access control.
  - Exploitation of vulnerabilities.
- **Security Issues:**
  - Over-privileged execution.
  - Complex debugging and auditing.
  - Powerful, but dangerous

Traditional Set-UID Programs

Coarse granularity of traditional privilege model is a problem:

- E.g, say we want to give a program the power to change system time
- Must also give it power to do everything else *root* can do
- **No limit on possible damage** if program is compromised

Capabilities in Linux

Capabilities in Linux

- Capabilities provide fine-grained privilege management in Linux.
- Breaks monolithic root privileges into smaller sets.
- Implemented via libcap library in C.
- Allows for safe delegation of specific privileges to processes or files

Rationale for capabilities

Capabilities divide power of superuser into small pieces

- 41 capabilities, as at Linux 6.13
- Traditional superuser == process that has full set of capabilities

Goal: replace set-UID-*root* programs with programs that have capabilities

- Compromise in set-UID-*root* binary -> very dangerous
- Compromise in binary with file capabilities -> less dangerous

Selection of Linux capabilities

CAP_CHOWN Make arbitrary changes to file UIDs and GIDs

CAP_DAC_OVERRIDE Bypass file RWX permission checks

CAP_DAC_READ_SEARCH Bypass file R and directory X permission checks

CAP_IPC_LOCK Lock memory

CAP_KILL Send signals to arbitrary processes

CAP_NET_ADMIN Various network-related operations

CAP_SETFCAP Set file capabilities

CAP_SETGID Make arbitrary changes to process's (own) GIDs

CAP_SETPCAP Make changes to process's (own) capabilities

CAP_SETUID Make arbitrary changes to process's (own) UIDs

CAP_SYS_ADMIN Perform a wide range of system admin tasks

More details: *capabilities(7)* manual page

Process Capabilities

- Capabilities assigned to running processes.
- Processes have three sets: Effective, Permitted, Inheritable.
- Example commands:
  - capsh --print
- Modifying process capabilities using prctl() and cap_set_proc().

File Capabilities

- Capabilities assigned to executables to run with specific privileges.
- Useful for granting minimal privileges to non-root processes.
- Example commands:
  - getcap /path/to/binary
  - setcap cap_net_bind_service+ep /path/to/binary

Summary

- Processes can have capabilities (subset of power of root)
- Files can have attached capabilities, which are given to process that executes program
- Privileged binaries/processes using capabilities are less dangerous if compromised

Namespaces

Namespaces

- A namespace (NS) “wraps” some global system resource to provide resource isolation
- Linux supports multiple NS types
  - 8 currently, and counting...

Types of namespaces

- Mount NS (CLONE_NEWNS;2.4.19, 2002)
  - Isolates mount points, allowing different views of the file system.
- UTS NS (CLONE_NEWUTS;2.6.19, 2006)
  - Isolates hostname and domain name settings.
- IPC NS (CLONE_NEWIPC;2.6.19, 2006)
  - Isolates inter-process communication mechanisms.
- PID NS (CLONE_NEWPID;2.6.24, 2008)
  - Isolates process IDs, enabling separate process trees.

Types of namespaces

- Network NS(CLONE_NEWNET;2.6.24, 2008)
  - Isolates network devices and settings.
- User NS (CLONE_NEWUSER;3.8, 2013)
  - Isolates user and group IDs, enabling privilege separation.
- Cgroup NS (CLONE_NEWCGROUP;4.6, 2016)
  - Isolates control groups, allowing per-namespace resource limits.
- Time NS (CLONE_NEWTIME;5.6, 2020)
  - Isolates system time settings within the namespace.

Namespace Hierarchy

- Parent-child relationship between namespaces.
- A user namespace can contain other namespaces.
- PID namespaces are nested within parent namespaces.
- Example:
  - `unshare --fork --pid --mount-proc bash`

NS instances

- Multiple **instances** of NS may exist on a system
- At system boot, there is one instance of each NS type–the **initial namespace**
- A process resides in one NS instance (of each of NS types)
- To processes inside NS instance, it appears that only they can see/modify corresponding global resource
- They are unaware of other instances of resource

NS instances

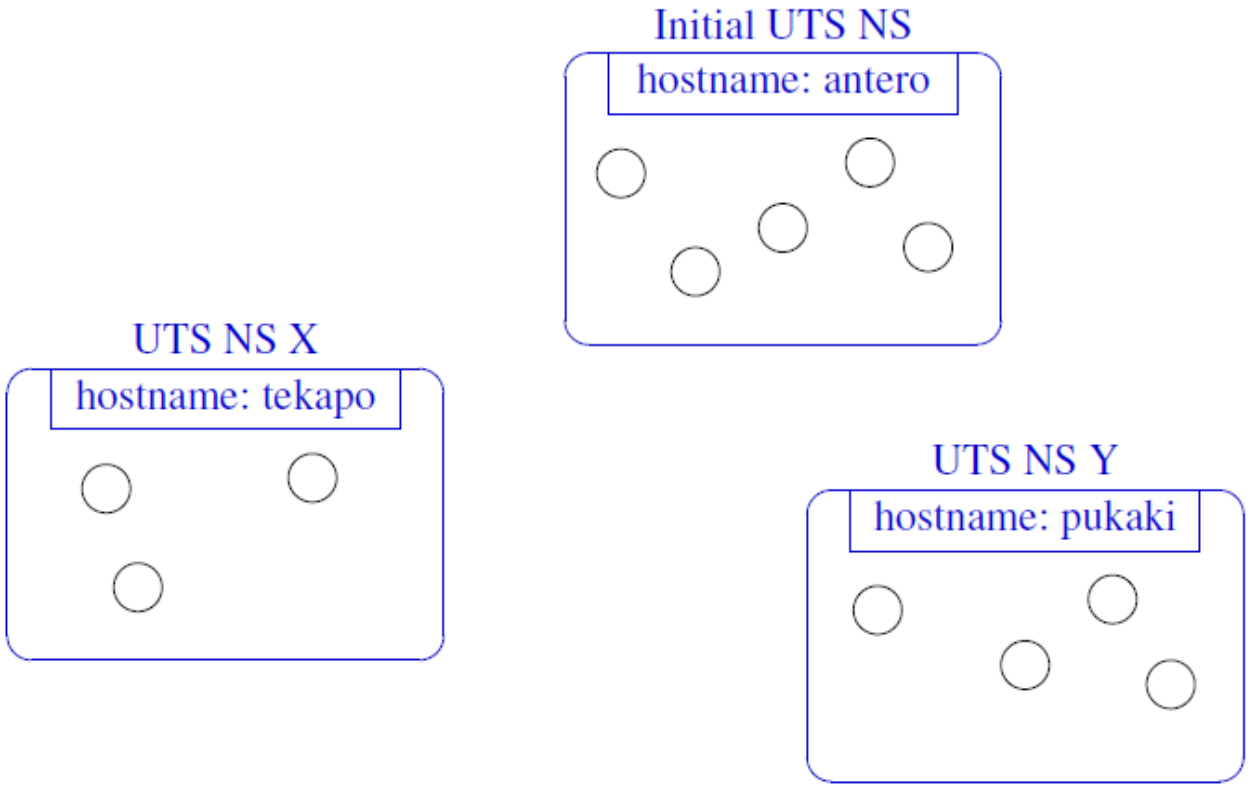
When new (child) process is created (*fork()*), it resides in same set of NSs as parent process

- There are system calls (and commands) for creating new NSs and moving processes into NSs

UTS namespaces

- **Isolate** certain system identifiers, including **hostname**
  - *hostname(1), uname(1)*
- Running system may have multiple UTS NS instances
- Processes in same NS instance access (get/set) same hostname
- Each NS instance has its own hostname
  - Changes to hostname in one NS instance are invisible to other instances

UTS namespace instances



Each UTS NS contains a set of processes (circles) which access (see/modify) same hostname

Symlink from /proc

/proc/[pid]/ns/ contains symbolic links to namespace objects.

Provides access to information about current namespaces.

Example:

- / proc /<PID>/ns/cgroup # Cgroup NS instance
- / proc /<PID>/ns/ipc # IPC NS instance
- / proc /<PID>/ns/mnt # Mount NS instance
- / proc /<PID>/ns/net # Network NS instance
- / proc /<PID>/ns/pid # PID NS instance
- / proc /<PID>/ns/user # User NS instance
- / proc /<PID>/ns/uts # UTS NS instance

Symlink from /proc

- Target of symlink tells us which NS instance process is in:
  - $ readlink / proc /$$/ns/uts
  - uts :[4026531838]
  - Content has form: ns-type :[magic-inode-#]
- Various uses for the /proc/PID/ns symlinks, including:
  - If processes show same symlink target, they are in same NS

APIs and commands

- Key APIs:
  - unshare(): Create new namespaces.
  - setns(): Join existing namespaces.
  - clone(): Create processes with new namespaces.
- Commands:
  - unshare, nsenter, ip netns.

Privilege for creating namespaces

- Creating user NS instances requires no privileges
- Creating instances of other (nonuser) NS types requires privilege
  - CAP_SYS_ADMIN

Demo

EMERTXE

User Namespace

- A user namespace is a special namespace type that isolates user and group IDs.
- It allows a process inside the namespace to have different (mapped) user IDs compared to the host system.
- This enables running processes with root privileges inside the namespace while being non-root outside (improving security).

User Namespace

Key Features of User Namespace:

- Each process inside the namespace can have different UID/GID mappings.
- Enables unprivileged users to create namespaces without root access.
- Useful for containerized applications needing root capabilities without system-wide risks.

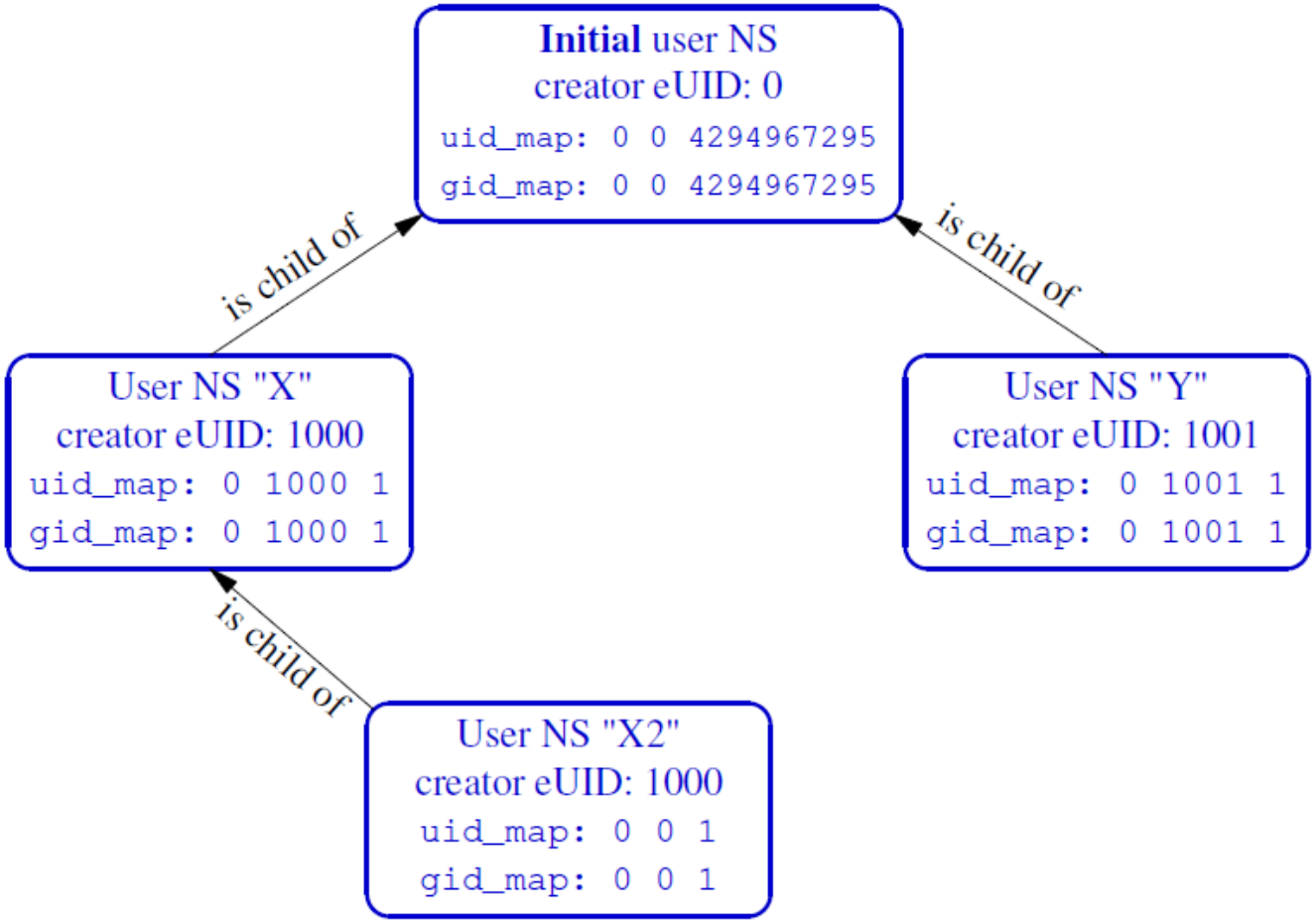
What do user namespaces do?

- 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*

Relationships between user namespaces

- User NSs have a hierarchical relationship:
  - A user NS can have zero or more child user NSs
  - Each user NS has parent NS, going back to initial user NS
- Parent of a user NS == user NS of process that created this user NS
  - Using clone(2), unshare(2), or unshare(1)
- Parental relationship determines some rules about how capabilities work

User namespace hierarchy



Root privileges in a user NS

- When a new user NS is created (unshare(1), clone(2), unshare(2)), first process in NS has all capabilities
- That process has power of superuser!
  - ... but only inside the user NS
- What does “root privileges in a user NS” really mean?

UID and GID mappings

- One of first steps after creating a user NS is to define UID and GID mappings for NS
- Mappings are defined by writing to 2 files:

  /proc/PID/uid_map and /proc/PID/gid_map
- For security reasons, there are many rules + restrictions on:
  - How/when files may be updated
  - Who can update the files
  - Way too many details to cover here...

    See user_namespaces(7)

UID and GID mappings

- Records written to/read from uid_map and gid_map have the 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
- Commonly these files are initialized with a single line containing “root mapping”:

```
0 1000 1
```

  - One ID, 0, inside NS maps to ID 1000 in outer NS

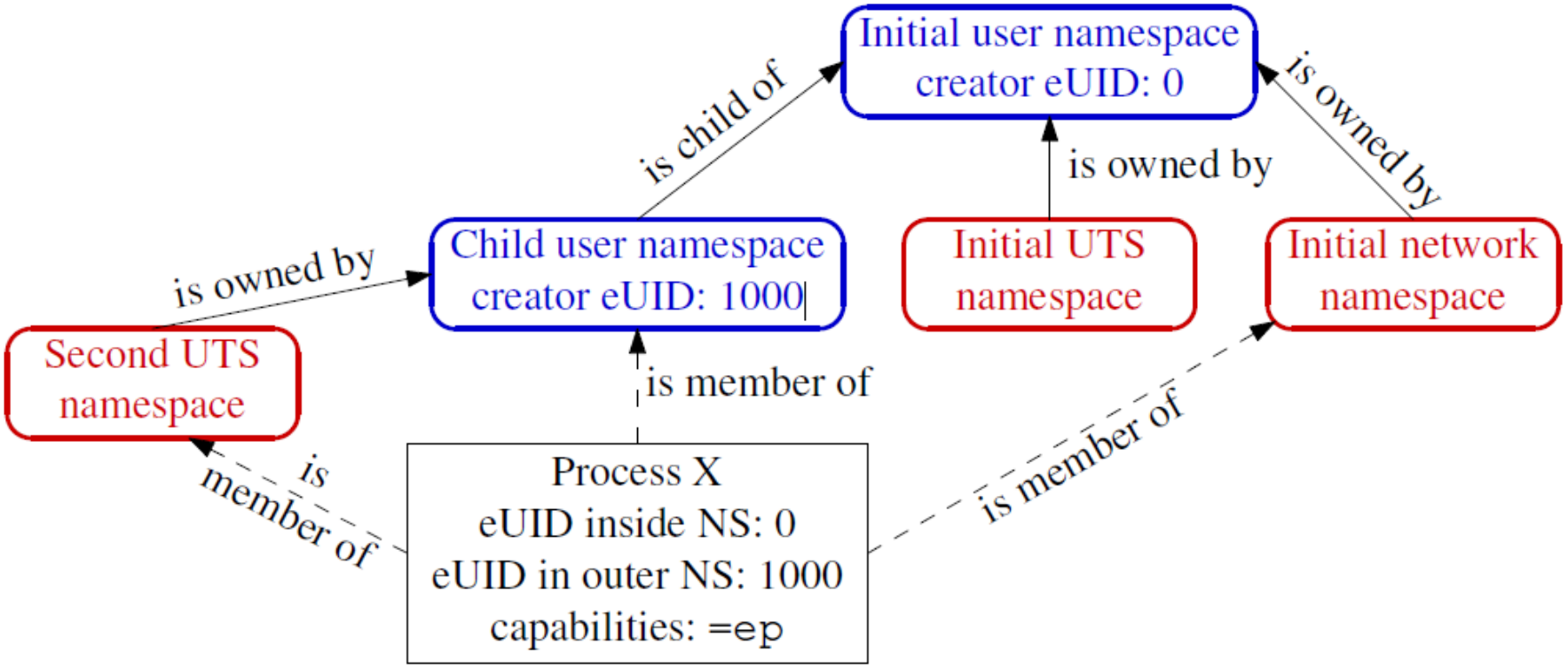
I’m superuser!

- From the shell in new user NS, let’s try to change the hostname
  - Requires CAP_SYS_ADMIN
  - `uns2$ hostname emertxe`
  - `hostname : you must be root to change the host name`
- Shell is UID 0 (superuser) and has CAP_SYS_ADMIN
- The new shell is in new user NS, but still resides in initial UTS NS
  - (Remember: hostname is isolated/governed by UTS NS)
  - Let’s look at this more closely...

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
- Each nonuser NS instance is owned by some user NS instance
  - When creating new nonuser NS, kernel marks that NS as owned by user NS of process creating the new NS
- If a process operates on resources governed by nonuser NS:
  - Permission checks are done according to that process's capabilities in user NS that owns the nonuser NS

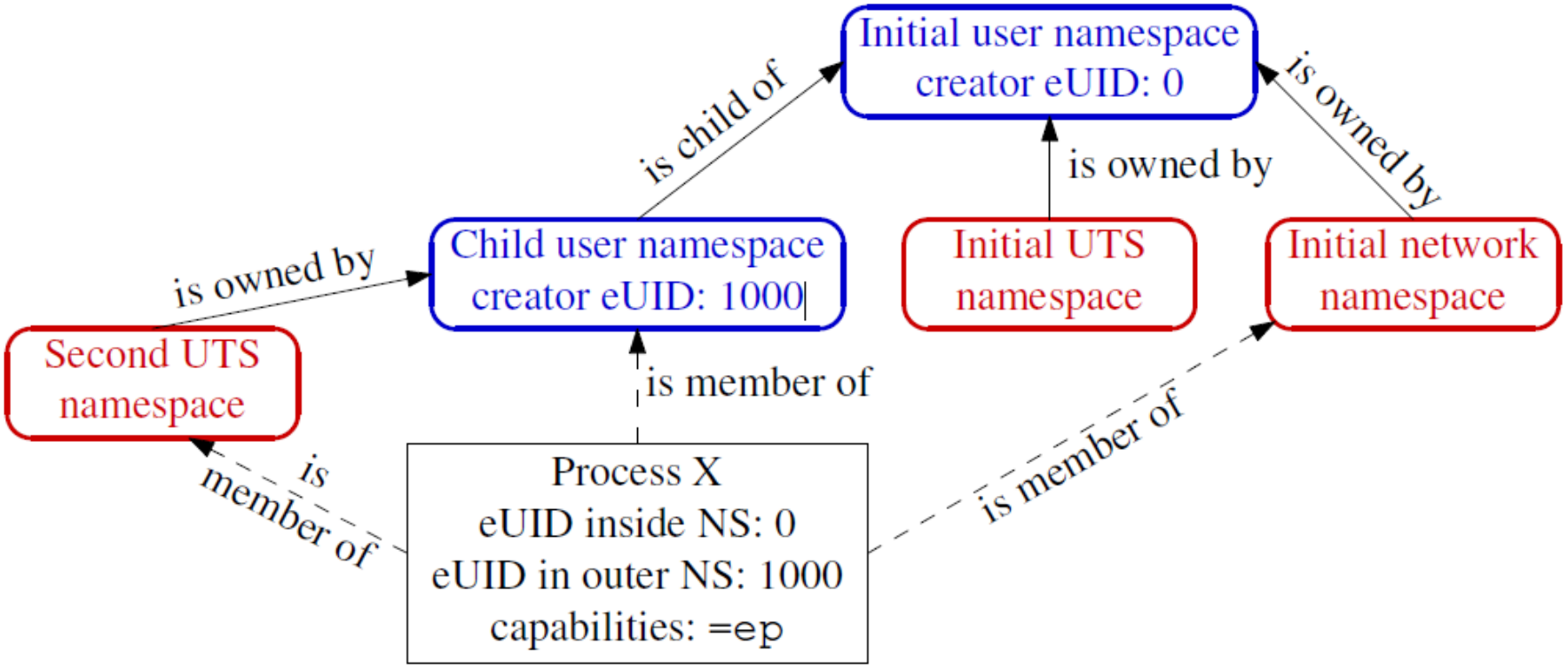
Example :



Example scenario; **X was created with: unshare -Ur -u <prog>**

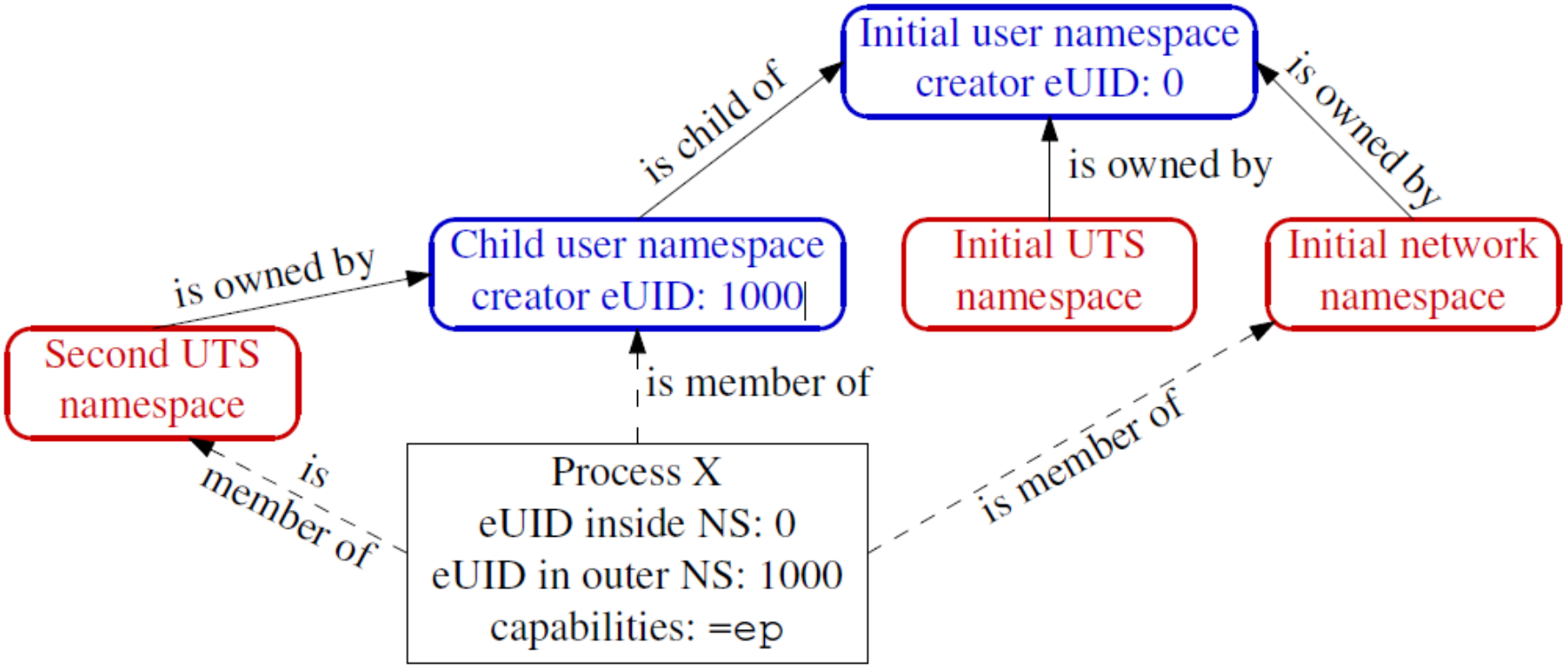
- X is in new user NS, with root mappings, and has all capabilities
- X is in a new UTS NS, which is owned by new user NS
- X is in initial instance of all other NS types (e.g., network NS)

Example :



- Suppose X tries to change hostname (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 that user NS)

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)

Security issues

Developer(s) of user NSs put much effort into ensuring capabilities couldn’t leak from inner user NS to outside NS

- Potential risk: some piece of kernel code might not be refactored to account for distinct user NSs
- unprivileged user who gains all capabilities in child NS might be able to do some privileged operation in **outer** NS

Security issues

User NS implementation touched a **lot** of kernel code

- Perhaps there were/are some unexpected corner case that wasn't correctly handled?
- A number of such cases have occurred (and been fixed)
- Common cause: many kernel code paths that could formerly be exercised only by *root* can now be exercised by any user
- Now, unprivileged users can test for weaknesses in kernel code paths that formerly could be accessed only by *root*

Use cases

- Running Linux Containers Without Root Privileges.
- Chrome-Style Sandboxes Without Set-UID-Root Helpers.
- chroot()-based Applications for Process Isolation.
- fakeroot-type Applications Without LD_PRELOAD/Dynamic Linking
- Firejail: Secure Application Sandboxing

For further information

- LWN.net article series Namespaces in operation
  - https://lwn.net/Articles/531114/
- Man pages:
  - namespaces(7), cgroup_namespaces(7), mount_namespaces(7), pid_namespaces(7), user_namespaces(7)
  - unshare(1), nsenter(1)
  - capabilities(7)
  - clone(2), unshare(2), setns(2), ioctl_ns(2)
- Linux containers in 500 lines of code
  - https://blog.lizzie.io/linux-containers-in-500-loc.html

Thanks