Identité multiple avec le namespace user

Rachid Koucha
[Ingénieur développement logiciel]

Après deux articles assez fastidieux mais néanmoins instructifs sur les internes des namespaces au sein du noyau de Linux, nous revenons en espace utilisateur pour une revue détaillée de chaque namespace. Cet opus se consacre au namespace user.

Table des matières

Avant-propos

Le code source des exemples utilisés dans cet article sont disponibles sur Github : https://github.com/Rachid-Koucha/linux_ns.

Cet article a été publié dans GNU Linux Magazine France n°246 du mois de mars 2021 :

Introduction

Cet article se consacre au namespace user (user_ns). Il a donné du fil à retordre aux développeurs du noyau Linux car il prend à sa charge une grande partie de la sécurité du système. Ce namespace isole les identifiants d'utilisateur et de groupe ainsi que d'autres attributs relatifs à la sécurité comme le répertoire racine, les capacités (« capabilities » en anglais). Il est à la base des conteneurs LXC non privilégiés.

Une entrée est dédiée à ce namespace dans la documentation : **man 7 user_namespaces**.

1 User_ns propriétaire

On a pu voir lors de l'étude des appels système, que l'opération **NS_GET_USERNS** du service **ioctl()** retourne un descripteur sur le fichier associé au user_ns auquel appartient un namespace. L'appartenance d'un namespace à un user_ns est déterminée au moment de sa création lors de l'appel à **unshare()** ou **clone()** :

- Si le drapeau **CLONE_NEWUSER** n'est pas passé, les namespaces créés appartiennent au user_ns du processus courant (l'appelant) ;

- Si le drapeau **CLONE_NEWUSER** est passé en même temps que les autres alors **le noyau garantit la création du nouveau user_ns en premier** afin que les namespaces créés dans la foulée appartiennent à ce dernier et non pas au user_ns du processus appelant.

Un user_ns nouvellement créé appartient quant à lui, toujours au user_ns du processus courant lors de l'appel à **unshare()** ou **clone()**. C'est ce qui fait que l'opération **NS_GET_PARENT** est synonyme de **NS_GET_USERNS** pour un user_ns car ils retournent le même résultat.

A titre d'illustration du propos, utilisons notre programme **ownerns** pour déterminer le user_ns propriétaire des user_ns, uts_ns et ipc_ns initiaux :

```
# ./ownerns $$ uts ipc
/proc/10659/ns/uts belongs to [Device,Inode]: [4,4026531837]
/proc/10659/ns/ipc belongs to [Device,Inode]: [4,4026531837]
```

Lançons notre programme **shns** dans un nouvel **user_ns**, **uts_ns** et **ipc_ns** :

```
# ./shns user uts ipc
New namespace 'ipc'
New namespace 'user'
New namespace 'uts'
$
```

Puis relançons **ownerns** avec les mêmes paramètres :

```
$ ./ownerns $$ uts ipc
/proc/13118/ns/uts belongs to [Device,Inode]: [4,4026532707]
/proc/13118/ns/ipc belongs to [Device,Inode]: [4,4026532707]
```

Nous vérifions bien que les deux namespaces créés ont un user_ns propriétaire différent qu'on peut voir dans un autre terminal avec l'identifiant du sous-shell :

```
# ls -l /proc/13118/ns/user
lrwxrwxrwx 1 rachid rachid 0 avril  7 15:42 user -> 'user:[4026532707]'
```

L'étude des sources du noyau a montré que les structures décrivant les namespaces ont un champ nommé **user_ns** référençant le descripteur du user_ns propriétaire.

L'un des objectifs majeurs est de permettre à des processus non privilégiés d'avoir des capacités à effectuer des actions uniquement dans le cadre du user_ns auquel ils sont attachés. Par exemple, si un processus fait appel à **sethostname()** afin de changer le nom de la machine, l'information étant sous le contrôle de l'uts_ns auquel il est lié, c'est le user_ns auquel est associé l'uts_ns qui définit les prérogatives pour effectuer l'opération.

En résumé, le user_ns parent sert au contrôle des privilèges pour l'accès et la modification des informations sous la responsabilité des namespaces qui lui appartiennent.

2 Les capacités

Le sujet est vaste. Une description se trouve dans le manuel de Linux : **man 7 capabilities**. Nous nous contenterons d'un tour d'horizon très rapide pour faciliter la compréhension de la suite.

A l'origine, Linux distinguait le super utilisateur privilégié avec tous les droits, des utilisateurs non privilégiés soumis à divers contrôles de permissions pour effectuer des actions. Depuis la version 2.2 du noyau, cette vision binaire a été nuancée afin de décomposer les privilèges en unités distinctes configurables unitairement. Ce sont les capacités (« capabilities » en anglais) ou prérogatives [1]. Elles sont attachées aux threads. Leur liste est longue. Le manuel les énumère toutes. Citons ici celles qui ont un rapport de près ou de loin avec les namespaces et LXC :

- **CAP_SETUID** et **CAP_SETGID** : autorisent la manipulation des identifiants d'utilisateurs et groupes, le renseignement des champs **uid** et **gid** des données auxiliaires de type **SCM_CREDENTIALS**, l'écriture dans les fichiers **/proc/<pid>/uid_map** et **/proc/<pid>/gid_map** ;

- **CAP_SYS_ADMIN** : autorise de nombreuses fonctions d'administration ainsi que d'autres actions non administratives qui n'ont pu être classées dans les autres capacités. Il y a entre autres l'utilisation des drapeaux **CLONE_*** avec **clone()** et **unshare()**, l'appel de services (**setns()**, **pivot_root()**, **sethostname()**, **setdomainname()**…), renseignement du champ **pid** des données auxiliaires de type **SCM_CREDENTIALS**… ;

- **CAP_SYS_CHROOT** : autorise l'appel à **chroot()** ;

- **CAP_SYS_RESOURCE** : autorise entre autres, l'opération **PR_SET_MM** de l'appel système **prctl()**. C'est utilisé par LXC pour renommer le processus de supervision des conteneurs en « [lxc monitor]... ».

Un thread a cinq ensembles de capacités dont les trois suivants :

- **Permitted** : limites des ensembles **Effective** et **Inheritable** ;

- **Inheritable** : capacités préservées après un appel à **execve()** ;

- **Effective** : capacités utilisées par le noyau pour les contrôles de permissions.

Ils sont visibles pour chaque thread dans **/proc/<pid>/task/<tid>/status** ou **/proc/<pid>/status** dans le cas du thread principal ou d'un programme mono-threadé avec les noms respectifs : **CapPrm**, **CapInh** et **CapEff**. Ils sont hérités par les processus fils et threads secondaires (**fork()**, **clone()**).

Le nombre maximum de capacités est 64 (un bit par capacité dans un entier). Les indices des bits sont définis dans le fichier d'entête **<linux/capability.h>**.

```
[...]
/* Allows setgid(2) manipulation */
/* Allows setgroups(2) */
/* Allows forged gids on socket credentials passing. */

#define CAP_SETGID          6

/* Allows set*uid(2) manipulation (including fsuid). */
/* Allows forged pids on socket credentials passing. */

#define CAP_SETUID          7
[...]
```

Actuellement, le fichier en expose 38. Pour une tâche non privilégiée, aucune capacité n'est activée :

```
$ cat /proc/$$/status | egrep "^Cap"
CapInh:       0000000000000000
CapPrm:       0000000000000000
CapEff:       0000000000000000
[...]
```

Pour une tâche du super utilisateur, elles sont toutes activées :

```
$ sudo su
# cat /proc/$$/status | egrep "^Cap"
CapInh: 0000000000000000
CapPrm:       0000003fffffffff
CapEff:       0000003fffffffff
[...]
```

Un processus résultant d'un appel à **clone()** ou suite à l'appel **unshare()** avec le drapeau **CLONE_NEWUSER**, **bénéficie de toutes les capacités dans le nouveau user_ns** (on l'a vu dans l'étude des sources du noyau). Ce qui semble contradictoire au premier abord est que ces fonctions nécessitent la capacité **CAP_SYS_ADMIN** pour être appelées avec les drapeaux relatifs aux namespaces. Un utilisateur non privilégié ne peut donc pas les appeler. Cependant, il existe une exception : **la création des user_ns est autorisée pour tout utilisateur**. En d'autres termes, à partir du moment où le drapeau **CLONE_NEWUSER** est spécifié, les appels système **clone()** et **unshare()** sont autorisés pour tout utilisateur (privilégié ou non). C'est sur ce « détail » que repose la possibilité pour un utilisateur de LXC de créer des conteneurs dits non privilégiés. Le manuel (**man 5 lxc.container.conf**) met même en exergue le fait d'être le premier gestionnaire de conteneur à avoir utilisé ce principe :

« LXC was the first runtime to support unprivileged containers after user namespaces were merged into the mainline kernel ».

La figure 1 présente la hiérarchisation des namespaces avec deux conteneurs LXC. L'un est privilégié tandis que l'autre est non privilégié. Les user_ns sont hiérarchisés avec le champ **parent** tandis que les autres namespaces sont « connectés » à leur user_ns propriétaire via le champ éponyme.

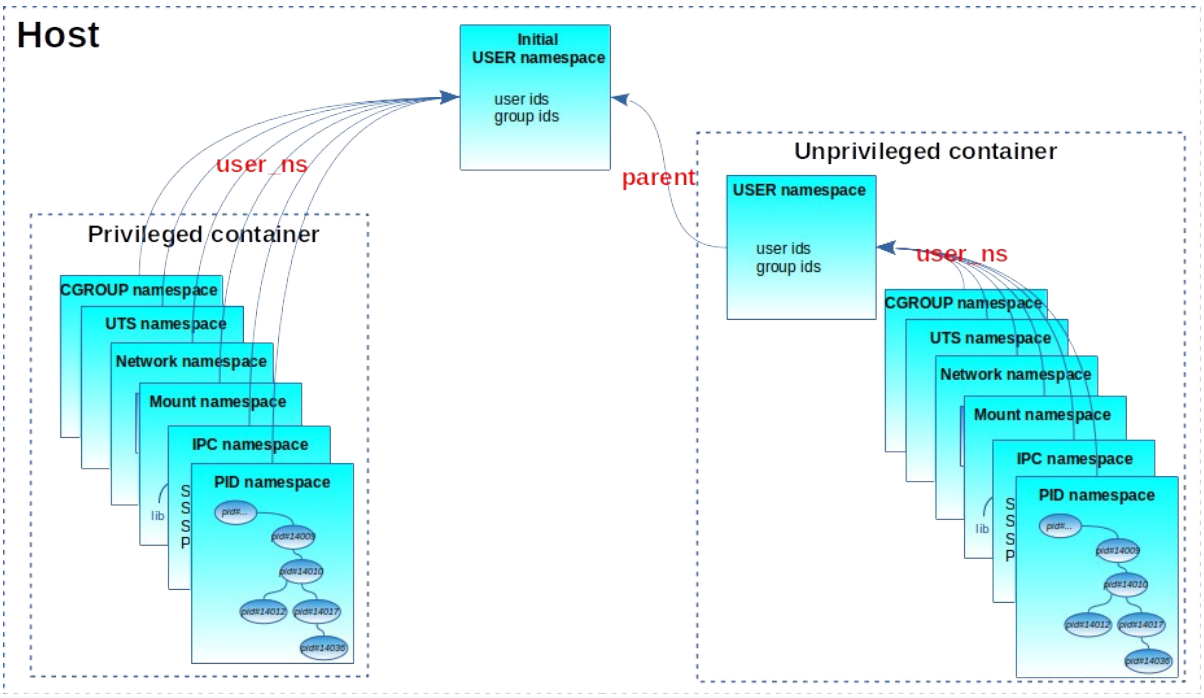

Fig. 1: Conteneur privilégié et non privilégié

Illustrons cela avec notre programme **shns** qui, nous le rappelons, appelle **unshare()** pour les namespaces demandés puis **fork()**/**execv()** pour exécuter un shell. Exécutons-le dans le contexte d'un **utilisateur non privilégié** :

```
$ PS1='SHNS$ ' ./shns ipc uts mnt
New namespace 'ipc'
New namespace 'uts'
New namespace 'mnt'
ERROR@main#110: unshare(0xc020000): 'Operation not permitted' (1)
```

Comme l'utilisateur n'est pas privilégié, l'appel **unshare()** retourne l'erreur **EPERM**. En relançant avec l'ajout du user_ns en paramètre, il n'y a plus d'erreur :

```
$ PS1='SHNS$ ' ./shns ipc uts mnt user
New namespace 'ipc'
New namespace 'user'
```

```
New namespace 'uts'
New namespace 'mnt'
SHNS$
```

Nous rappelons qu'à partir du moment où **unshare()** (tout comme **clone()**) est appelé avec le drapeau **CLONE_NEWUSER**, le user_ns est créé avant tous les autres. Le processus appelant **unshare()** (ou qui en résulte pour **clone()**) bénéficie par conséquent de toutes les capacités et peut donc créér tous les autres namespaces demandés. C'est la raison pour laquelle les namespaces **ipc**, **uts** et **mnt** demandés ont pu être créés dans la foulée du user_ns alors que le processus appelant **shns** n'est pas privilégié.

Nous venons de dire que le processus bénéficie de toutes les capacités lorsqu'il entre dans un nouveau user_ns pourtant si nous regardons les capacités du sous-shell, aucune n'est activée :

```
SHNS$ cat /proc/$$/status | egrep "^Cap"
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
[...]
```

Comme **shns** appelle **fork()** après **unshare()**, le processus fils devient bien privilégié grâce au drapeau **CLONE_NEWUSER** car les capacités sont héritées. Mais ensuite ce dernier appelle **execv()** pour lancer le shell. Cela lui fait perdre toutes ses capacités. Le manuel des user_ns (**man 7 user_namespaces**) le rappelle:
Note that a call to execve(2) will cause a process's capabilities to be recalculated in the usual way (see capabilities(7)). Consequently, unless the process has a user ID of 0 within the namespace, or the executable file has a nonempty inheritable capabilities mask, the process will lose all capabilities.
Pour les garder, il aurait fallu qu'il ait l'identité **0** (c.-à-d. super utilisateur) dans le nouveau user_ns. La correspondance d'identifiants d'utilisateurs et groupes entre user_ns donne la possibilité de changer l'identité d'un processus. C'est le mécanisme de *mapping*.

3 Mapping

L'intérêt premier des user_ns est d'établir une correspondance entre des identifiants d'utilisateurs et groupes dans le user_ns père avec des identifiants dans le user_ns fils. Cela donne par exemple la possibilité de mapper un utilisateur non privilégié (identifiant différent de 0) dans le user_ns père sur le super utilisateur (identifiant égal à **0**) dans le user_ns fils. De cette manière, le super utilisateur dans un conteneur, mappé sur un utilisateur non privilégié côté hôte, peut donc effectuer des tâches d'administration (configuration des interfaces réseau, montage/démontage de systèmes de fichiers...). **Les impératifs de sécurité limitent tout de même certaines opérations au user_ns initial**. Par exemple, il ne sera pas possible de changer l'heure système ou charger/décharger des modules dans le noyau à partir d'un conteneur.

A la création, le user_ns n'a pas de correspondance d'identifiant avec son namespace père. Cela se vérifie dans l'exemple avec **shns** :

```
SHNS$ id
uid=65534(nobody) gid=65534(nogroup) groups=65534(nogroup)
```

Par défaut, les identifiants sont positionnés à la valeur 65534 respectivement tirée des fichiers **/proc/sys/kernel/overflowuid** et **/proc/sys/kernel/overflowgid**.

Le mapping des identifiants d'utilisateurs et groupes est effectué en écrivant respectivement dans les fichiers **/proc/<pid>/uid_map** et **/proc/<pid>/gid_map** pour un processus d'identifiant **pid**. Ces fichiers sont vides à la création du user_ns :

```
SHNS$ cat /proc/$$/uid_map
SHNS$ cat /proc/$$/gid_map
```

Leur format est composé de lignes avec trois valeurs numériques :

- La valeur de départ de l'intervalle d'identifiants dans le user_ns du processus **<pid>** ;

- La valeur de départ de l'intervalle d'identifiants correspondants dans le user_ns père si le processus qui consulte ce fichier a l'identifiant **<pid>** ou appartient au même user_ns que le processus **<pid>**. Sinon c'est le user_ns du processus qui a ouvert le fichier (ce n'est pas facile à appréhender au premier abord) ;

- Le nombre d'identifiants consécutifs dans l'intervalle.

La procédure de mapping consiste à écrire dans les fichiers d'un des processus associés au user_ns nouvellement créé. De nombreuses restrictions sont imposées essentiellement pour des raisons de sécurité (cf. **man 7 user_namespaces**). Par exemples :

- Les fichiers ne peuvent être mis à jour qu'une seule fois ;

- Il y a au maximum 340 lignes par fichier (limite arbitraire qui a évolué d'une version à l'autre du noyau) et le nombre total de caractères ne doit pas dépasser la taille d'une page mémoire ;

- Il est nécessaire de désactiver l'appel système **setgroups()** en écrivant « deny » dans le fichier **/proc/<pid>/setgroups** avant d'écrire dans le fichier **/proc/<pid>/gid_map** (contournement ajouté à postériori pour résoudre des problèmes de sécurité)...

Revenons à notre exemple ci-dessus afin d'ajouter un mapping d'identifiants entre les user_ns père et fils. Le but est d'établir une correspondance entre les identifiants d'utilisateur et groupe du shell dans le user_ns père (dans notre cas c'est 1000 pour les deux) avec les identifiants 0 pour l'utilisateur et le groupe dans le user_ns fils (c.-à-d. le super utilisateur !). Ainsi, le processus fils exécutant le sous-shell sera super utilisateur.

Repérons l'identifiant du sous-shell en affichant la variable **$$** :

```
SHNS$ echo $$
14229
```

Dans un autre terminal, nous sommes dans le user_ns initial d'où on a créé le user_ns du sous_shell. **Notre vision sur le deuxième champ des fichiers uid_map et gid_map correspond donc à la plage d'utilisateurs dans le user_ns du processus courant donc le user_ns père du sous-shell**.

Effectuons les correspondances pour les identifiants d'utilisateur et groupe avec les écritures suivantes dans les fichiers **uid_map** et **gid_map** :

```
$ id
uid=1000(rachid) gid=1000(rachid) groups=1000(rachid),4(adm),...
$ echo "0 1000 1" > /proc/14229/uid_map
$ echo "deny" > /proc/14229/setgroups
$ echo "0 1000 1" > /proc/14229/gid_map
```

Vu du sous-shell, les identifiants d'utilisateur et groupe sont bien changés :

```
SHNS$ id
uid=0(root) gid=0(root) groups=0(root),65534(nogroup)
```

L'utilisateur non privilégié dans le user_ns initial (uid et gid égaux à 1000) correspond donc à un utilisateur privilégié dans le user_ns fils (uid et gid égaux à 0) ! Vérifions les capacités de ce super-utilisateur :

```
SHNS$ cat /proc/$$/status | egrep "^Cap"
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
[...]
```

Les capacités sont toujours désactivées ! On est dans un cas peu commun où un super utilisateur n'a aucun droit. En fait, nous avons certes résolu l'identité de l'utilisateur mais au moment où le changement est opéré, l'appel à **execv()** pour le sous-shell est déjà fait. Et **execv()** fait perdre les capacités avant le mapping d'identité du user_ns père vers le user_ns fils.

Le programme **shns2** est une version modifiée de **shns**. Il a simplement été ajouté l'attente d'une réponse de l'opérateur entre l'appel à **fork()** et l'appel à **execv()** de sorte à donner la possibilité d'effectuer les actions de mapping entre les deux appels système :

```
[...]
  // Fork a child process
  child = fork();
  if (!child) {

    // Child process

    int   c;
    char *av_cmd[] = { DEFAULT_CMD, NULL };

    prompt("Process#%d go forward ([Y]/N)? ", getpid());
     c = getanswer();
     if ('\n' == c || 'y' == c || 'Y' == c) {
       execv(av_cmd[0], av_cmd);
[...]
     }

     return 1;
[...]
```

Relançons notre sous-shell avec **shns2** qui affiche aussi l'identifiant (15706) de processus fils créé :

```
$ PS1='SHNS$ ' ./shns2 ipc uts mnt user
New namespace 'ipc'
New namespace 'user'
New namespace 'uts'
New namespace 'mnt'
Process#15706 go forward ([Y]/N)?
```

Dans l'autre terminal, effectuons les actions de mapping :

```
$ echo "0 1000 1" > /proc/15706/uid_map
$ echo "deny" > /proc/15706/setgroups
$ echo "0 1000 1" > /proc/15706/gid_map
```

Profitons-en au passage, pour vérifier qu'une deuxième tentative d'écriture dans les fichiers génère bien une erreur (**EPERM**) comme le manuel l'affirme :

```
$ echo "0 1000 1" > /proc/15706/uid_map
bash: echo: write error: Operation not permitted
$ echo "0 1000 1" > /proc/15706/gid_map
bash: echo: write error: Operation not permitted
```

Dans le sous-shell, nous entrons la réponse attendue (**Y**) pour débloquer le processus fils de **shns2** afin qu'il exécute le shell. Etant maintenant exécuté avec l'identifiant du super-utilisateur dans le processus fils, l'appel **execv()** ne supprime plus les capacités du processus appelant :

```
Process#15706 go forward ([Y]/N)? Y
SHNS$ id
uid=0(root) gid=0(root) groups=0(root),65534(nogroup)
SHNS$ cat /proc/$$/status | egrep "^Cap"
CapInh: 0000000000000000
CapPrm:       0000003fffffffff
CapEff:       0000003fffffffff   # The 38 capabilities are set
[...]
```

Nous venons de faire pas à pas ce qui est réalisé par les options **-f** et **-r** de la commande **unshare**. Elles provoquent respectivement les opérations de **fork()** puis de mapping de l'utilisateur appelant vers le super utilisateur dans le user_ns fils avant d'exécuter la commande :

```
$ PS1='SHNS$ ' unshare -i -u -m -U -f -r /bin/sh
SHNS# id
uid=0(root) gid=0(root) groups=0(root),65534(nogroup)
SHNS$ cat /proc/$$/status | egrep "^Cap"
CapInh: 0000000000000000
CapPrm:       0000003fffffffff
CapEff:       0000003fffffffff
[...]
```

Le résultat de **strace** pour la commande précédente est le suivant :

```
[...]
geteuid()                               = 1000 # uid of calling user
getegid()                               = 1000 # gid of calling user
[...]
unshare(CLONE_NEWNS|CLONE_NEWUTS|CLONE_NEWIPC|CLONE_NEWUSER) = 0
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD...
openat(AT_FDCWD, "/proc/self/setgroups", O_WRONLY) = 3
write(3, "deny", 4)                     = 4
close(3)                                = 0
openat(AT_FDCWD, "/proc/self/uid_map", O_WRONLY) = 3
write(3, "0 1000 1", 8)                 = 8
close(3)                                = 0
openat(AT_FDCWD, "/proc/self/gid_map", O_WRONLY) = 3
write(3, "0 1000 1", 8)                 = 8
close(3)                                = 0
[...]
execve("/bin/sh", ["/bin/sh"], 0x7fffcd2654c0 /* 59 vars */) = 0
[...]
```

Le cheminement du mapping jusqu'à maintenant a montré qu'on ne peut faire correspondre que l'identité de l'utilisateur créateur du user_ns (1000 dans notre exemple) avec l'identité d'un utilisateur (0 ou autre) dans le user_ns fils. Mais il nous est interdit de mapper un autre utilisateur du user_ns père à cause des impératifs de sécurité. Par exemple, avec un nouveau lancement de **shns2** qui créé un fils d'identifiant égal à **13319**. Les tentatives de mapper plusieurs plages d'identifiants (**[0,1]** avec **[1000,1001]** et **[100000,100099]** avec **[3,102]**) ou une plage plus grande que un (**[0,1]** avec **[1000,1001]**) se soldent par l'erreur **EPERM** :

```
$ cat /proc/13319/uid_map
$ echo -e "0 1000 2\n100000 3 100" > /proc/13319/uid_map
bash: echo: write error: Operation not permitted
$ echo -e "0 1000 2" > /proc/13319/uid_map
bash: echo: write error: Operation not permitted
$ cat /proc/13319/uid_map
```

Comme l'indique le manuel, pour cela il faudrait que le processus qui écrit, ait les capacités **CAP_SETUID** et **CAP_SETGID**. C'est à priori contradictoire avec le principe qui consiste à pouvoir créer des conteneurs par un utilisateur non privilégié. Serait-on limité à la création de conteneurs avec un unique utilisateur ?

Pour mapper plusieurs utilisateurs, il faut en fait passer par un « **setuid helper** » !

4 Identifiants subordonnés

On a pu le voir jusqu'à maintenant que le *mapping* d'utilisateur entre user_ns est une fonction indispensable mais vite dissuasive lorsqu'il s'agit d'en comprendre toutes les conditions d'utilisation. Pour faciliter la tâche des opérateurs, les outils **newuidmap** et **newgidmap** ont été introduits. Sous une distribution comme **Ubuntu**, ces outils s'installent à partir du package **uidmap**. Ils permettent de configurer les intervalles de *mapping* des identifiants d'utilisateurs et groupes entre un user_ns père et son fils. Ils sont installés sous la propriété de **root** et avec le bit **setuid** :

```
$ ls -la /usr/bin/new[ug]idmap
-rwsr-xr-x 1 root root 41392 août  29 15:00 /usr/bin/newgidmap
-rwsr-xr-x 1 root root 41392 août  29 15:00 /usr/bin/newuidmap
```

Le bit **setuid** est très important car il permet de conférer à l'utilisateur appelant, les privilèges de l'utilisateur propriétaire du programme (root ici). Ils aident les applications à configurer les mappings des user_ns lorsque l'opérateur n'est pas privilégié. Ils sont par exemple utilisés en interne par LXC pour la configuration des conteneurs non privilégiés. Les synopsis sont les suivants :

```
newuidmap pid uid loweruid count [uid loweruid count [ ... ]]
newgidmap pid gid lowergid count [gid lowergid count [ ... ]]
```

Les deux programmes sont appelés suivant le même schéma. Ils reçoivent en premier paramètre l'identifiant de processus cible dans le user_ns fils (typiquement le premier processus) et les plages d'identifiants à mapper du user_ns fils vers le user_ns père. Ces plages sont des triplets avec les identifiants de départ de la plage dans les user_ns fils et père, suivis de la longueur de la plage. En interne, les programmes consultent respectivement les fichiers **/etc/subuid** et **/etc/subgid** dans lesquels se trouvent pour les utilisateurs, les identifiants utilisables comme subordonnés dans les user_ns fils. Les lignes de ces fichiers consistent en trois champs séparés par des « : » :

- Le nom d'utilisateur ;

- Le début de la plage d'identifiants qu'il peut utiliser en subordonnés ;

- La longueur de la plage d'identifiants qu'il peut utiliser en subordonnés.

A titre d'exemple, ici l'utilisateur **rachid** peut mapper les identifiants **100000** à **165535** tandis que l'utilisateur **rachid1** peut utiliser les identifiants **165536** à **231071** :

```
$ cat /etc/subuid
rachid:100000:65536
rachid1:165536:65536
```

De même, pour les identifiants de groupe :

```
$ cat /etc/subgid
rachid:100000:65536
rachid1:165536:65536
```

Relançons notre programme **shns2** qui bloque le processus fils juste avant d'exécuter le shell :

```
$ PS1='SHNS$ ' ./shns2 ipc uts mnt user
New namespace 'ipc'
New namespace 'user'
New namespace 'uts'
New namespace 'mnt'
Process#15303 go forward ([Y]/N)?
```

A ce moment, le processus a toutes les capacités en tant que premier processus du user_ns et il les gardera tant qu'il ne fera pas appel à **execv()**. Mais comme on a besoin qu'il exécute le shell en gardant ses privilèges, il faut qu'ils ait l'identité du super-utilisateur (uid/gid égaux à **0**). Ce qu'on a réussi à faire jusqu'à maintenant. Dans un autre terminal, on va en plus définir les plages d'utilisateurs supplémentaires désirées et autorisées en passant l'identifiant de processus puis les plages en paramètres à **newuidmap** et **newgidmap** :

```
$ newuidmap 15303 0 1000 1 1 100000 100
$ cat /proc/15303/uid_map
         0       1000          1
         1     100000        100
$ newgidmap 15303 0 1000 1 1 100000 100
$ cat /proc/15303/gid_map
         0       1000          1
         1     100000        100
```

Nous avons ainsi déclaré dans le user_ns fils que :

- L'utilisateur/groupe d'identifiant 0 (super-utilisateur) est mappé sur l'utilisateur/groupe d'identifiant 1000 dans le user_ns père. **Ce mapping s'impose car l'utilisateur qui a lancé shns2 et donc son fils est l'utilisateur d'identifiant 1000 dans le user_ns père, il faut lui créer une correspondance dans le user_ns fils pour qu'il ait une identité**. Les manuels de **newuidmap**/**newgidmap** prêtent à confusion car ils indiquent que les outils vérifient les intervalles passés en paramètre par rapport au contenu des fichiers **subuid**/**subgid** dans **/etc**. Or l'uid/gid 1000 du processus appelant n'est pas dans ce fichier ! **En fait l'outil autorise aussi en paramètre le mapping de l'uid/gid du processus appelant avec un intervalle de taille 1** (cf. encadré intitulé « Les intervalles passés à new[ug]idmap ») ;

- Les utilisateurs/groupes d'identifiants 1 à 99 sont mappés sur les utilisateurs/groupes d'identifiants 100000 à 100099.

De retour dans le sous-shell, on saisit **<Y>** et on constate bien que l'identité a été mise à jour comme précédemment :

```
Process#15677 go forward ([Y]/N)? Y
SHNS$ id
uid=0(root) gid=0(root) groups=0(root),65534(nogroup)
SHNS$
```

On note au passage, que ces **newuidmap**/**newgidmap** rendent transparentes des spécificités inhérentes à la manipulation des mappings (c.-à-d. l'écriture préalable dans **/proc/<pid>/setgroups**) afin de simplifier la vie de l'opérateur.

Les intervalles de new[ug]idmap

Le manuel indique que **new[ug]idmap** met à jour les plages de correspondance dans **/proc/<pid>/[ug]id_map**. Mais il vérifie au préalable l'appartenance du processus **<pid>** (premier paramètre) à l'utilisateur appelant puis s'assure que les triplets qui suivent sur la ligne de commande sont cohérents avec les intervalles définis pour l'utilisateur dans le fichier **/etc/sub[ug]id**.

Mais l'explication n'est pas complète car le code source disponible dans le fichier **new[ug]idmap.c** du répertoire **https://github.com/shadow-maint/shadow/blob/master/src** autorise aussi l'appelant à mapper son propre **[ug]id** avec un identifiant quelconque dans le user_ns du processus cible. L'intervalle de correspondance doit avoir un seul élément dans ce cas. Voici par exemple, la fonction de validation des intervalles passés en argument dans **newgidmap** :

```
static bool verify_range(struct passwd *pw, struct map_range *range, bool *allow_setgroups)
{
[...]
  /* Test /etc/subgid. If the mapping is valid then we allow setgroups. */
  if (have_sub_gids(pw->pw_name, range->lower, range->count)) {
    *allow_setgroups = true;
    return true;
  }

  /* Allow a process to map its own gid. */
  if ((range->count == 1) && (pw->pw_gid == range->lower)) {
    /* noop -- if setgroups is enabled already we won't disable it. */
    return true;
  }

  return false;
}
```

Le programme **shns3** est une synthèse de tout ce qui a été vu jusqu'à maintenant. Il crée un shell privilégié à partir d'un utilisateur non privilégié. L'analyse des paramètres est plus évoluée car elle s'appuie sur le service **getopt_long()**. Le synopsis est le suivant :

```
$ ./shns3 -h
Usage: shns3 [-h] [-d level] [-n nsname] [-u uidmap] [-g gidmap] [-s path]

  -h|--help            : This help
  -h|--debug level     : Debug level
  -n|--nspc nsname     : Create namespace
                         'nsname' is: cgroup|ipc|net|mnt|pid|user|uts
  -u|--umap uidmap     : User id mapping
                         'uidmap' is 'uid loweruid count'
  -g|--gmap gidmap     : Group id mapping
                         'gidmap' is 'gid lowergid count'
  -s|--shell path      : Execute shell
                         'path' is '/bin/sh' by default
```

Les namespaces à créer sont passés avec l'option **--nspc**. Les plages (triplets) de correspondance des identifiants d'utilisateurs et groupes sont passés respectivement via les options **--umap** et **--gmap** afin d'exécuter **new[ug]idmap** lorsque la liste des namespaces passée en paramètre via l'option **--nspc** mentionne un user_ns. Trois processus sont mis en œuvre en interne :

- Le processus principal (le père) analyse la ligne de commande et prépare les drapeaux à passer à **unshare()** pour la création des namespaces en fonction du paramètre **--nspc** ;

- Si un user_ns est demandé, un premier processus fils est créé dans les namespaces courant (donc avant l'appel à **unshare()** !). La synchronisation consiste à faire une lecture bloquante sur un tube afin de recevoir l'identifiant du second processus fils afin de le passer en premier paramètre de la commande **new[ug]idmap**. L'intérêt de ce premier fils est de s'exécuter dans les namespaces de départ donc sans perdre l'identité de l'utilisateur appelant (sinon une entrée dans un nouvel user_ns ferait perdre le mapping de l'utilisateur courant et rendrait donc impossible l'exécution de la commande) ;

- Le processus père crée les namespaces demandés en appelant **unshare()** puis crée un second processus fils ;

- Le second processus fils étant créé après **unshare()**, il s'exécute dans les nouveaux namespaces demandés. Comme on l'a déjà vu (et on le reverra lors de l'étude plus détaillée des pid_ns), l'intérêt de ce

second fils est que seuls les processus fils d'un processus créant un nouveau pid_ns s'exécutent effectivement dans le nouveau pid_ns ;

- Le second processus fils effectue une lecture bloquante sur un second tube ;

- Une fois le second processus fils créé, le processus père débloque le premier processus fils en écrivant l'identifiant du second processus dans le premier tube puis se met en attente de la fin de ce dernier avec un appel à **waitpid()** ;

- Le premier processus fils sort de sa lecture bloquante sur le premier tube, exécute les commandes **new[ug]idmap** avec pour paramètres le pid du second fils tout juste reçu et les plages de correspondance passées via les options **--umap** et **--gmap**. Puis il se termine ;

- Le père sort de son attente de la fin du premier fils, débloque la lecture du second fils en écrivant une donnée quelconque dans le second tube puis se met en attente de la fin de ce dernier avec un appel à **waitpid()** ;

- Le second processus fils sort de sa léthargie, exécute le shell demandé via l'option **--shell** (ou celui par défaut). Comme le premier fils a effectué les mappings d'identités, le second fils obtient l'identité de l'utilisateur appelant mappée sur une identité conforme aux mappings demandés via les options **--umap** et **--gmap** ;

- Le processus père ne rendra la main qu'à partir du moment où l'opérateur sortira du shell exécuté par le second fils.

Le code source de **shns3** est le suivant :

```
int main(int ac, char *av[])
{
[...]
  while ((opt = getopt_long(ac, av, ":hd:s:u:g:n:", shns_longopts, NULL)) != EOF) {
[...]
  // For each namespace of the target process, set the flag
  flags = 0;
  for (i = 0; i < NB_NS_NAME; i ++) {
    if (ns_list[i].selected) {
      DBG(1, "New namespace '%s'\n", ns_list[i].name);
      flags |= ns_list[i].flag;
    } // End if namespace selected
  } // End for
[...]
  // If user namespace, make the maps
  if (ns_list[SHNS_USER_IDX].selected) {

    // Make the synchro pipe
    rc = pipe(sync1);
[...]
    // We need to fork a process before changing the namespaces
    // to run new[ug]idmap otherwise the programs will check the
    // current user id in a user_ns where nothing is mapped (uid/gid = 65534).
    // So, the programs will end in error reporting that the current user
    // has no permission to make the mapping operation
    child1 = fork();

    switch(child1) {

      case 0: { // Child process#1
[...]
        // Synchronize with father
        close(sync1[1]);
        rc = read(sync1[0], &child2, sizeof(child2));
[...]
        // Make the mappings

        cmdline = make_cmdline("newuidmap", child2, umap, numap);
[...]
        rc = run_cmdline(cmdline);
[...]
        cmdline = make_cmdline("newgidmap", child2, gmap, ngmap);
[...]
        rc = run_cmdline(cmdline);
[...]
```

```
        // End of 1st child
        exit(0);

      } break;
[...]
      default: { // Father

        // Let's continue...

      } break;

    } // End switch

  } // End if user namespace

  // Make the synchro pipe
  rc = pipe(sync2);
[...]
  // Create brand new namespaces
  rc = unshare(flags);
[...]
  // Fork a child process to make sure that it will be run
  // in new pid_ns (if any)
  child2 = fork();
  switch(child2) {

    case 0: { // Child process#2

      char *av_cmd[] = { shell, NULL };

      // Synchronize with father
      close(sync2[1]);
      rc = read(sync2[0], &i, sizeof(i));
[...]
      // Execute the shell
       execv(av_cmd[0], av_cmd);
[...]
    } break;
[...]
    default: {

      // Father process, let's continue...
      close(sync2[0]);

    } break;

  } // End switch fork()

  // If user namespaces, synchronize with the first child
  if (ns_list[SHNS_USER_IDX].selected) {
[...]
    // Synchronize with the 1st child
    rc = write(sync1[1], &child2, sizeof(child2));
[...]
    // Wait for the end of the 1st child process
    rc = waitpid(child1, &status, 0);
[...]
  } // End if user namespaces

  // Synchronize with the 2nd child
[...]
  rc = write(sync2[1], &i, sizeof(i));
[...]
  // Wait for the end of the child process
  rc = waitpid(child2, &status, 0);
[...]
  return 0;
} // main
```

Relançons avec **shns3** un test identique au précédent avec **shns2**. Comme aucun namespace n'est passé en paramètre, tous les namespaces sont créés. On ajoute l'option de debug de manière à afficher quelques informations supplémentaires au démarrage :

```
$ id
uid=1000(rachid) gid=1000(rachid) groups=1000(rachid),4(adm),24(cdrom),[...]
```

```
$ ./shns3 -u '0 1000 1' -u '1 100000 100' -g '0 1000 1' -g '1 100000 100' -d 1
DEBUG_1 (main#358): New namespace 'ipc'
DEBUG_1 (main#358): New namespace 'pid'
DEBUG_1 (main#358): New namespace 'net'
DEBUG_1 (main#358): New namespace 'user'
DEBUG_1 (main#358): New namespace 'uts'
DEBUG_1 (main#358): New namespace 'cgroup'
DEBUG_1 (main#358): New namespace 'mnt'
DEBUG_1 (main#409): Running 'newuidmap 7565 0 1000 1 1 100000 100'...
DEBUG_1 (main#424): Running 'newgidmap 7565 0 1000 1 1 100000 100'...
# id
uid=0(root) gid=0(root) groups=0(root),65534(nogroup)
#
```

Nous voilà donc avec un shell privilégié lancé à partir d'un utilisateur non privilégié ! La figure 2 donne une version graphique de l'algorithme de **shns3** dans le cadre de cet exemple.

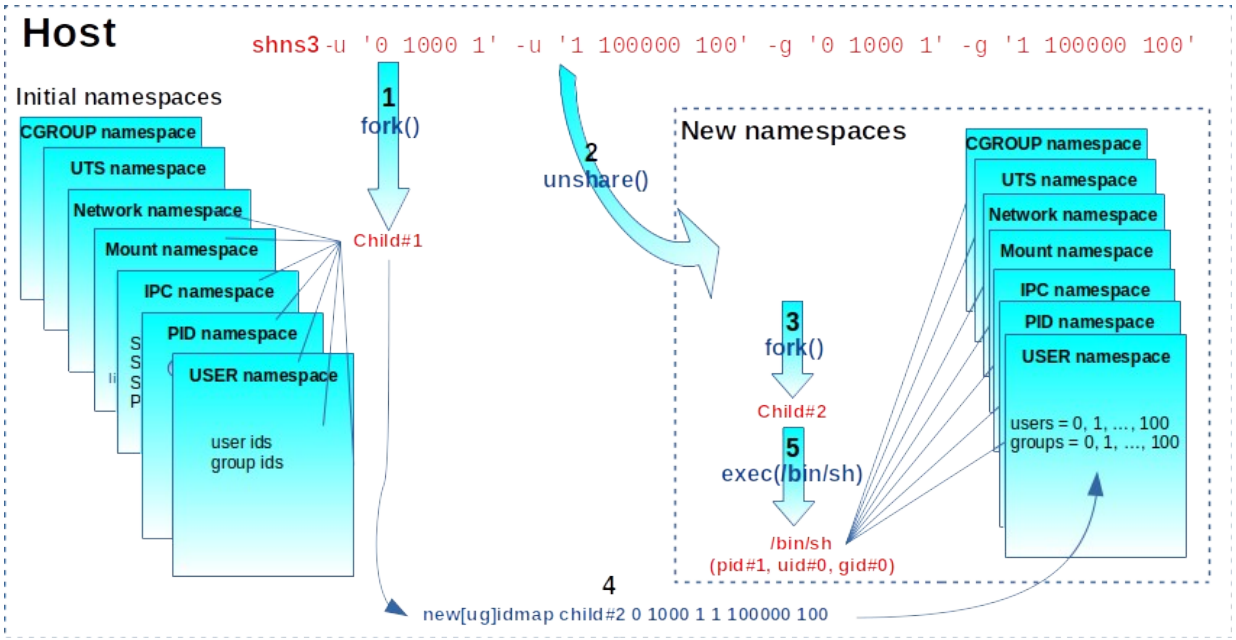

*Fig. 2: **shns3** en action*

Cependant le résultat de la commande **ps** semble incohérent car elle affiche les identifiants de processus du namespace initial alors qu'on s'exécute dans un nouveau pid_ns avec un seul processus (le shell d'identifiant **1**) :

```
# echo $$
1
# ps
  PID TTY          TIME CMD
 7730 pts/2    00:00:00 shns3
 7732 pts/2    00:00:00 sh
 7740 pts/2    00:00:00 ps
30805 pts/2    00:00:00 bash
```

On déborde sur l'étude des mount_ns à venir pour préciser que **ps** va chercher ses informations dans **/proc**. Mais **PROCFS** exporte les informations correspondant au pid_ns du processus qui l'a monté. Ici c'est un processus d'initialisation du système et par conséquent le pid_ns initial ! Il faut donc remonter **PROCFS** avec les commandes « magiques » suivantes pour avoir un système de fichiers **/proc** en accord avec les nouveaux namespaces :

```
# mount --make-rslave /
# mount -t proc proc /proc
# ps
  PID TTY          TIME CMD
    1 pts/2    00:00:00 sh
    5 pts/2    00:00:00 ps
```

Cela deviendra plus limpide dans un article suivant...

5 PROCFS

Dans chaque user_ns, les fichiers suivants sous **/proc/sys/user** donnent les limites sur le nombre de namespaces qui peuvent être créés dans chaque catégorie (cf. **man namespaces**) :

- **max_cgroup_namespaces** : nombre maximum de cgroup_ns qu'il peut être créé dans le user_ns courant ;

- **max_ipc_namespaces** : nombre maximum d'ipc_ns qu'il peut être créé dans le user_ns courant ;

- et ainsi de suite pour tous les autres types...

Dans les namespaces initiaux, les limites sont la moitié du nombre maximum de threads qu'il peut être créé et dont la valeur se trouve dans **/proc/sys/kernel/threads-max** :

```
$ cat /proc/sys/kernel/threads-max
127318
$ cat /proc/sys/user/max_cgroup_namespaces
63659     # = 127318 / 2
$ cat /proc/sys/user/max_net_namespaces
63659     # = 127318 / 2
```

Dans les user_ns fils (c.-à-d. descendants du user_ns initial), les valeurs sont par défaut positionnées à **MAXINT**. Par exemple, dans notre shell **shns3** lancé ci-dessus :

```
# cat /proc/sys/user/max_cgroup_namespaces
2147483647   # = MAXINT
# cat /proc/sys/user/max_net_namespaces
2147483647   # = MAXINT
```

Comme c'est spécifique à chaque user_ns, si on modifie les valeurs dans **shns3** :

```
# echo 1024 > /proc/sys/user/max_net_namespaces
# cat /proc/sys/user/max_net_namespaces
1024
```

Cela n'impacte pas les autres user_ns. Le user_ns initial reste par exemple inchangé :

```
$ cat /proc/sys/user/max_net_namespaces
63659
```

Lorsque ces limites sont atteintes, les appels système comme **clone()** ou **unshare()** retournent l'erreur **ENOSPC**.

6 Conversion d'identifiants

Les user_ns étant hiérarchiques, un processus a un identifiant d'utilisateur et de groupe dans le user_ns courant et dans tous les user_ns parents jusqu'au user_ns initial. Une fonctionnalité de Linux laconiquement présentée dans la rubrique « Miscellaneous » du manuel des user_ns, indique :

« When a process's user and group IDs are passed over a UNIX domain socket to a process in a different user namespace (see the description of SCM_CREDENTIALS in unix(7)), they are translated into the corresponding values as per the receiving process's user and group ID mappings ».

En d'autres termes, à partir d'un user_ns, l'uid et le gid dans les données auxiliaires d'un message envoyé via une socket Unix d'un user_ns à un autre, sont implicitement convertis en accord avec les *mappings* définis. Nous reviendrons sur cet aspect dans l'article dédié aux pid_ns car ils présentent une fonctionnalité similaires avec les identifiants de processus...

Conclusion

Le user_ns est d'une telle importance qu'il méritait bien un article dédié. Il est à la base de la sécurité des conteneurs notamment en donnant la possibilité de créer des conteneurs non privilégiés au sein desquels le super utilisateur est mappé sur un utilisateur normal côté hôte.

Références

[1] Linux capabilities support for user namespaces : **https://lwn.net/Articles/420624/**