Subject: design of user namespaces
Posted by ebiederm on Fri, 20 Jun 2008 02:01:08 GMT
View Forum Message <> Reply to Message

I got to thinking about the user namespace again
today and I wanted to summarize my thoughts.

First and foremost I want a design that solves the suid problem,
and I think I have.

In particular if we can let an unprivileged user run a process
in a different user namespace, and become a user with 0 == uid == gid
and have all capabilities except CAP_SYS_RAWIO and they still can not
do anything security wise they could not before. We have a very useful design.

My thinking is as follows.
- All user visible security credentials uid, gid, selinux filesystem lables,
  etc will be relative to a user namespace.

  This makes security easier, and it the only thing that makes
  sense in the concept of separate administrative domain.

- Each struct user will live in exactly one user namespace.

- Each user namespace will hold the struct user of the user who
  created the user namespace.

- capable() will be extended to take a user namespace parameter. So
  the question capable asks is does the current process have the
  capability with respect to the current user namespace.

  This makes security namespaces loosely hierarchical. And the rule
  would be that if you could do something to the files and directories
  and other objects owned by the creator of the namespace you can do
  it to objects owned by users in the user namespace.

  Going the other direction from child to parent user namespaces,
  users in user namespace no matter what capabilities they have will
  have no permissions except those given to every process.

- Each user namespace will have a bounding set of capabilities that
  suid executables and the initial process get. Basically something
  so fundamentally machine specific capabilities like CAP_SYS_RAWIO
  can never be set inside of a user namespace unless the creator of
  the user namespace had those capabilities.

- The initial user in a user namespace will be uid 0 and have all
  capabilities, non machine specific cpabilities CAP_NET_ADMIN,

CAP_SYS_ADMIN, CAP_SYS_CHROOT etc. Since those capabilities will be restricted to objects in the user namespace they will have no security implications.

- struct vfsmount will include a user namespace pointer. Recording in which mount namespace the mount occurred, and not changing when struct vfsmount is copied or propagated (including through bind mounts). This user namespace will act as the user namespace to map all security credentials on the filesystem into. If you are accessing the vfsmount from another user namespace the best you can get is a the world readable permissions.

  struct vfsmount will also include a filesystem data pointer. Not changing except when struct user changes. Giving filesystems a place to store their per user namespace credential translation state.

  This allows is shared filesystem caches between user namespaces.

- For filesystems to be safely shared between 2 user namespaces we need two things. The owner of the filesystem has to allow it. The user of the filesystem needs to request it.

  For user namespaces we have two cases. Allowing native mounts to a user namespace of a previously mounted filesystem. Allowing native mounts to of an unmounted filesystem.

  Working with previously mounted filesystems is the safest as we don't have to deal with the hard problem of poisoned filesystem data trying to crash our filesystem implementation.

  For previously filesystems the simplest and most comprehensive way I can see to do this is to implement a special case of mount -o remount. In which a parameter is passed telling the filesystem which credential mapping strategy to employ. The credential mapping strategies I have seen to date are
  - identity mapping read-only.
  - identity mapping read-write (this is a problem with quotas)
  - uid/gid offset by a constant (solves the quota problem)
  - security label for namespaces (can solve the quota problem).

  Then the already mounted filesystem can either performs an upcall or examine it's configuration (potentially stored in a normal file in the filesystem like the quotas are) to see if the remount into native mode for the user namespace can be allowed.

  For the specific and very common case of identity mapping read-only filesystems we could even have a completely generic mount flag you

can set a priori to allow any user namespace to remount it native.

Most unix filesystems have common enough properties that we can implement a library they can wire up to implement this functionality without requiring an on disk format change. So despite it being filesystem specific functionality we can extensive share the code.

For network filesystems like nfs I expect the request would go to the server and authenticating a new mount. It is still safer then a totally new nfs mount because an unprivileged user can not specify the server.

- Previously I really wanted to say just do something like idmapd and we would be golden. The problem of quotas in different namespaces would be solved, etc. After thinking about it I realized there is no same place to run such a mapping daemon that could map between arbitrary user namespaces that could do something useful and not also compromise security.

Mapping daemons are good at changing the form of security tokens. Say by looking up user names in /etc/passwd. They are not sufficient to perform a general mapping.

The owner of the filesystem has to configure what you are allowed to see and access. Which uids you can use, which directories you can use etc. While it is the job of the mapping daemon to map the resources it has available to it (username strings typically) to something the users of the filesystem can actually use.

Eric

_____

Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: design of user namespaces
Posted by serue on Fri, 20 Jun 2008 14:05:10 GMT
View Forum Message <> Reply to Message

Quoting Eric W. Biederman (ebiederm@xmission.com):
>
> I got to thinking about the user namespace again
> today and I wanted to summarize my thoughts.
>
> First and foremost I want a design that solves the suid problem,
> and I think I have.

>
> In particular if we can let an unprivileged user run a process
> in a different user namespace, and become a user with 0 == uid == gid
> and have all capabilities except CAP_SYS_RAWIO and they still can not
> do anything security wise they could not before. We have a very useful design.
>
> My thinking is as follows.
> - All user visible security credentials uid, gid, selinux filesystem lables,
> etc will be relative to a user namespace.
>
> This makes security easier, and it the only thing that makes
> sense in the concept of separate administrative domain.
>
> - Each struct user will live in exactly one user namespace.
>
> - Each user namespace will hold the struct user of the user who
> created the user namespace.
>
> - capable() will be extended to take a user namespace parameter. So
> the question capable asks is does the current process have the
> capability with respect to the current user namespace.
>
> This makes security namespaces loosely hierarchical. And the rule
> would be that if you could do something to the files and directories
> and other objects owned by the creator of the namespace you can do
> it to objects owned by users in the user namespace.
>
> Going the other direction from child to parent user namespaces,
> users in user namespace no matter what capabilities they have will
> have no permissions except those given to every process.
>
> - Each user namespace will have a bounding set of capabilities that
> suid executables and the initial process get. Basically something
> so fundamentally machine specific capabilities like CAP_SYS_RAWIO
> can never be set inside of a user namespace unless the creator of
> the user namespace had those capabilities.
>
> - The initial user in a user namespace will be uid 0 and have all
> capabilities, non machine specific cpabilities CAP_NET_ADMIN,
> CAP_SYS_ADMIN, CAP_SYS_CHROOT etc. Since those capabilities will be
> restricted to objects in the user namespace they will have no
> security implications.
>
> - struct vfsmount will include a user namespace pointer. Recording
> in which mount namespace the mount occurred, and not changing
> when struct vfsmount is copied or propagated (including through bind
> mounts). This user namespace will act as the user namespace to map
> all security credentials on the filesystem into. If you are

> accessing the vfsmount from another user namespace the best you can
> get is a the world readable permissions.
>
> struct vfsmount will also include a filesystem data pointer. Not
> changing except when struct user changes. Giving filesystems a
> place to store their per user namespace credential translation
> state.
>
> This allows is shared filesystem caches between user namespaces.
>
> - For filesystems to be safely shared between 2 user namespaces we
> need two things. The owner of the filesystem has to allow it. The
> user of the filesystem needs to request it.
>
> For user namespaces we have two cases. Allowing native mounts
> to a user namespace of a previously mounted filesystem. Allowing
> native mounts to of an unmounted filesystem.
>
> Working with previously mounted filesystems is the safest as we
> don't have to deal with the hard problem of poisoned filesystem data
> trying to crash our filesystem implementation.
>
> For previously filesystems the simplest and most comprehensive way I can
> see to do this is to implement a special case of mount -o remount.
> In which a parameter is passed telling the filesystem which
> credential mapping strategy to employ. The credential mapping
> strategies I have seen to date are
> - identity mapping read-only.
> - identity mapping read-write (this is a problem with quotas)
> - uid/gid offset by a constant (solves the quota problem)
> - security label for namespaces (can solve the quota problem).
>
> Then the already mounted filesystem can either performs an upcall
> or examine it's configuration (potentially stored in a normal file
> in the filesystem like the quotas are) to see if the remount into
> native mode for the user namespace can be allowed.
>
> For the specific and very common case of identity mapping read-only
> filesystems we could even have a completely generic mount flag you
> can set a priori to allow any user namespace to remount it native.
>
> Most unix filesystems have common enough properties that we can
> implement a library they can wire up to implement this functionality
> without requiring an on disk format change. So despite it being
> filesystem specific functionality we can extensive share the code.
>
> For network filesystems like nfs I expect the request would go to
> the server and authenticating a new mount. It is still safer then

> a totally new nfs mount because an unprivileged user can not specify
> the server.
>
> - Previously I really wanted to say just do something like idmapd and
> we would be golden. The problem of quotas in different namespaces
> would be solved, etc. After thinking about it I realized there is
> no same place to run such a mapping daemon that could map between
> arbitrary user namespaces that could do something useful and not
> also compromise security.
>
> Mapping daemons are good at changing the form of security tokens.
> Say by looking up user names in /etc/passwd. They are not
> sufficient to perform a general mapping.
>
> The owner of the filesystem has to configure what you are allowed
> to see and access. Which uids you can use, which directories you
> can use etc. While it is the job of the mapping daemon to map the
> resources it has available to it (username strings typically) to
> something the users of the filesystem can actually use.
>
>
> Eric

Hi Eric,

glad you're giving this some thought. Did you ever read over the approach which I outlined in May (see http://forum.openvz.org/index.php?t=msg&goto=30223&)? We agree on many points. I think we basically solve the suid problem the same way. But I've moved away from a uid-to-uid mapping. Instead, I expand on the relation you also describe: the user who creates a user namespace owns the user namespace and introduce a persistant namespace ID.

I understand that you may reflexively balk at introducing a new global persistant ID when we're focusing on turning everything into a namespace, but I think that would be a misguided reflex (like the ioctl->netlinke one of a few years ago). In particular, in your approach the identifier is the combination of the uid-to-uid mapping and the uids stored on the original filesystem.

I do think the particular form of the ID I suggest will be unsuitable and we'll want something more flexible. Perhaps stick with the unsid for the legacy filesystems with xattr-unsid support, and let advanced filesystems like nfsv4, 9p, and smb use their own domain identifiers.

But since we seem to agree on the first part - introducing a hierarchy between users and the namespaces they create - it sounds like the following patch would suit both of us. (I just started implementing my

approach this past week in my free time). I'm not sending this as any
sort of request for inclusion, just bc it's sitting around...

-serge

>From cfb61975eb8989eee9fcc07a8ddc68c1087874c2 Mon Sep 17 00:00:00 2001
From: Serge Hallyn <serge@us.ibm.com>
Date: Thu, 19 Jun 2008 20:18:17 -0500
Subject: [PATCH 1/1] user namespaces: introduce user_struct->user_namespace relationship

When a task does clone(CLONE_NEWNS), the task's user is the 'creator' of the
new user_namespace, and the user_namespace is tacked onto a list of those
created by this user.

When we create or put a user in a namespace, we also do so for all creator
users up the creator chain.

Signed-off-by: Serge Hallyn <serge@us.ibm.com>
---
 include/linux/sched.h | 2 +
 include/linux/user_namespace.h | 2 +
 kernel/user.c | 68 ++-
 kernel/user_namespace.c | 18 +++++-----
 4 files changed, 79 insertions(+), 11 deletions(-)

diff --git a/include/linux/sched.h b/include/linux/sched.h
index 799bbdd..0837236 100644
--- a/include/linux/sched.h
+++ b/include/linux/sched.h
@@ -604,6 +604,8 @@ struct user_struct {
  /* Hash table maintenance information */
  struct hlist_node uidhash_node;
  uid_t uid;
+ struct user_namespace *user_namespace;
+ struct list_head child_user_ns;

 #ifdef CONFIG_USER_SCHED
  struct task_group *tg;
diff --git a/include/linux/user_namespace.h b/include/linux/user_namespace.h
index b5f41d4..eca4b3a 100644
--- a/include/linux/user_namespace.h
+++ b/include/linux/user_namespace.h
@@ -13,6 +13,8 @@ struct user_namespace {
  struct kref kref;
  struct hlist_head uidhash_table[UIDHASH_SZ];
  struct user_struct *root_user;
+ struct user_struct *creator;
+ struct list_head siblings;

```
 };

 extern struct user_namespace init_user_ns;
diff --git a/kernel/user.c b/kernel/user.c
index 865ecf5..b29d90f 100644
--- a/kernel/user.c
+++ b/kernel/user.c
@@ -21,6 +21,8 @@ struct user_namespace init_user_ns = {
  .kref = {
   .refcount = ATOMIC_INIT(2),
  },
+ .creator = &root_user,
+ .siblings = LIST_HEAD_INIT(root_user.child_user_ns),
  .root_user = &root_user,
 };
 EXPORT_SYMBOL_GPL(init_user_ns);
@@ -53,6 +55,8 @@ struct user_struct root_user = {
  .files  = ATOMIC_INIT(0),
  .sigpending = ATOMIC_INIT(0),
  .locked_shm     = 0,
+ .child_user_ns = LIST_HEAD_INIT(init_user_ns.siblings),
+ .user_namespace = &init_user_ns,
 #ifdef CONFIG_USER_SCHED
  .tg  = &init_task_group,
 #endif
@@ -71,6 +75,18 @@ static void uid_hash_remove(struct user_struct *up)
  hlist_del_init(&up->uidhash_node);
 }

+void inc_user_and_creators(struct user_struct *user)
+{
+ struct user_namespace *ns = user->user_namespace;
+ while (user) {
+  atomic_inc(&user->__count);
+  if (ns == ns->creator->user_namespace)
+   return;
+  user = ns->creator;
+  ns = user->user_namespace;
+ }
+}
+
 static struct user_struct *uid_hash_find(uid_t uid, struct hlist_head *hashent)
 {
  struct user_struct *user;
@@ -78,7 +94,7 @@ static struct user_struct *uid_hash_find(uid_t uid, struct hlist_head
*hashent)

  hlist_for_each_entry(user, h, hashent, uidhash_node) {
```

```
   if (user->uid == uid) {
-   atomic_inc(&user->__count);
+   inc_user_and_creators(user);
    return user;
   }
  }
@@ -315,12 +331,30 @@ done:
  uids_mutex_unlock();
 }

+/*
+ * Decrement use counts for all namespace ancestors of the user
+ * being freed.  The user itself has already been dec'ed, so
+ * we only start at its creator.
+ */
+void dec_creators(struct user_struct *user)
+{
+ struct user_namespace *ns = user->user_namespace;
+ while (ns != ns->creator->user_namespace) {
+  user = ns->creator;
+  atomic_dec(&user->__count);
+  ns = user->user_namespace;
+ }
+}
+
 /* IRQs are disabled and uidhash_lock is held upon function entry.
  * IRQ state (as stored in flags) is restored and uidhash_lock released
  * upon function exit.
  */
 static inline void free_user(struct user_struct *up, unsigned long flags)
 {
+ /* decrement all creator counts */
+ dec_creators(up);
+
  /* restore back the count */
  atomic_inc(&up->__count);
  spin_unlock_irqrestore(&uidhash_lock, flags);
@@ -409,6 +443,8 @@ struct user_struct *alloc_uid(struct user_namespace *ns, uid_t uid)
   if (sched_create_user(new) < 0)
    goto out_free_user;

+  new->user_namespace = ns;
+
   if (uids_user_create(new))
    goto out_destoy_sched;

@@ -429,6 +465,7 @@ struct user_struct *alloc_uid(struct user_namespace *ns, uid_t uid)
    kmem_cache_free(uid_cachep, new);
```

```
   } else {
    uid_hash_insert(new, hashent);
+   inc_user_and_creators(new);
    up = new;
   }
   spin_unlock_irq(&uidhash_lock);
@@ -448,6 +485,35 @@ out_unlock:
  return NULL;
 }

+/*
+ * After doing clone(CLONE_NEWUSER), the new task continues to hold
+ * a refcount on ancestor users, but just switches to the new
+ * root user in the child namespace
+ */
+void switch_uid_for_created_root(struct user_struct *new_user)
+{
+ struct user_struct *old_user;
+
+ old_user = current->user;
+ atomic_inc(&new_user->processes);
+ switch_uid_keyring(new_user);
+ current->user = new_user;
+ sched_switch_user(current);
+
+ /*
+  * We need to synchronize with __sigqueue_alloc()
+  * doing a get_uid(p->user).. If that saw the old
+  * user value, we need to wait until it has exited
+  * its critical region before we can free the old
+  * structure.
+  */
+ smp_mb();
+ spin_unlock_wait(&current->sighand->siglock);
+
+ free_uid(old_user);
+ suid_keys(current);
+}
+
 void switch_uid(struct user_struct *new_user)
 {
  struct user_struct *old_user;
diff --git a/kernel/user_namespace.c b/kernel/user_namespace.c
index a9ab059..775f177 100644
--- a/kernel/user_namespace.c
+++ b/kernel/user_namespace.c
@@ -11,6 +11,7 @@
 #include <linux/slab.h>
```

```
 #include <linux/user_namespace.h>

+extern void switch_uid_for_created_root(struct user_struct *new_user);
 /*
  * Clone a new ns copying an original user ns, setting refcount to 1
  * @old_ns: namespace to clone
@@ -19,7 +20,6 @@
 static struct user_namespace *clone_user_ns(struct user_namespace *old_ns)
 {
  struct user_namespace *ns;
- struct user_struct *new_user;
  int n;

  ns = kmalloc(sizeof(struct user_namespace), GFP_KERNEL);
@@ -31,6 +31,10 @@ static struct user_namespace *clone_user_ns(struct user_namespace
*old_ns)
  for (n = 0; n < UIDHASH_SZ; ++n)
   INIT_HLIST_HEAD(ns->uidhash_table + n);

+ /* set up owner/parent relationship */
+ ns->creator = current->user;
+ list_add_tail(&ns->siblings, &current->user->child_user_ns);
+
  /* Insert new root user.  */
  ns->root_user = alloc_uid(ns, 0);
  if (!ns->root_user) {
@@ -38,15 +42,7 @@ static struct user_namespace *clone_user_ns(struct user_namespace
*old_ns)
   return ERR_PTR(-ENOMEM);
  }

- /* Reset current->user with a new one */
- new_user = alloc_uid(ns, current->uid);
- if (!new_user) {
-  free_uid(ns->root_user);
-  kfree(ns);
-  return ERR_PTR(-ENOMEM);
- }
-
- switch_uid(new_user);
+ switch_uid_for_created_root(ns->root_user);
  return ns;
 }

@@ -72,6 +68,8 @@ void free_user_ns(struct kref *kref)

  ns = container_of(kref, struct user_namespace, kref);
  release_uids(ns);
```

```
+ if (!list_empty(&ns->siblings))
+  list_del(&ns->siblings);
  kfree(ns);
 }
 EXPORT_SYMBOL(free_user_ns);
--
1.5.4.3
```

Subject: Re: design of user namespaces
Posted by ebiederm on Fri, 20 Jun 2008 19:03:21 GMT
View Forum Message <> Reply to Message

"Serge E. Hallyn" <serue@us.ibm.com> writes:
>
> Hi Eric,
>
> glad you're giving this some thought. Did you ever read over the
> approach which I outlined in May (see
> http://forum.openvz.org/index.php?t=msg&goto=30223&)? We agree on many
> points. I think we basically solve the suid problem the same way.

I hadn't read that post although I saw a part of it in your paper.
Solving that problem so unprivileged users can use a user namespace
seems to be the key to making namespaces more widely usable, and
recursive. Plus I really like the idea of a super nobody user.

> But I've moved away from a uid-to-uid mapping. Instead,
> I expand on the relation you also describe: the user who creates a user
> namespace owns the user namespace and introduce a persistant
> namespace ID.

I think there are management issues with a persistent namespace ID.
In particular: Is this user allowed to use this ID?

That is the reason I want to avoid having a generic persistent
namespace ID. Implementing a common library and then modifying the
filesystems to use it on a case by case basis sounds much more
maintainable. Then picking the wrong direction does not bind us for
all time and eternity.

> I understand that you may reflexively balk at introducing a new global
> persistant ID when we're focusing on turning everything into a

> namespace, but I think that would be a misguided reflex (like the
> ioctl->netlinke one of a few years ago). In particular, in your
> approach the identifier is the combination of the uid-to-uid mapping and
> the uids stored on the original filesystem.

Not at all. I thought I had mentioned the xattr thing as one of the possibilities. I forgot the proper term so I probably said acls. The practical problem is that you then have to rev your quota support. To also support the xattr separation. In addition not every filesystem supports xattrs. Although the common ones do.

> I do think the particular form of the ID I suggest will be unsuitable
> and we'll want something more flexible. Perhaps stick with the unsid
> for the legacy filesystems with xattr-unsid support, and let advanced
> filesystems like nfsv4, 9p, and smb use their own domain
> identifiers.

Which is why I said make it filesystem specific with support from a generic library. No prctl just a mount option.

> But since we seem to agree on the first part - introducing a hierarchy
> between users and the namespaces they create - it sounds like the
> following patch would suit both of us. (I just started implementing my
> approach this past week in my free time). I'm not sending this as any
> sort of request for inclusion, just bc it's sitting around...

Yes. At least a loose hierarchy.

It just occurred to me that with unix domain sockets, some signals, /proc we have user namespace crossing (child to parent) where we need to report the uid. In that case the simple solution appears to be to use the overflowuid and overflowgid that were defined as part of the 16bit to 32bit transition. Otherwise it would require mapping all of the child uids into the parent like we do with pids except using an upcall to userspace.

Making capabilities user namespace specific if we can.

Did you have any problems with adding a user_namespace argument to capable?

Just skimming through your patch I don't expect we will need the list of children, and not having should reduct our locking burden.

Eric

_____

Subject: Re: design of user namespaces
Posted by serue on Fri, 20 Jun 2008 20:55:08 GMT
View Forum Message <> Reply to Message

Quoting Eric W. Biederman (ebiederm@xmission.com):
> "Serge E. Hallyn" <serue@us.ibm.com> writes:
> >
> > Hi Eric,
> >
> > glad you're giving this some thought. Did you ever read over the
> > approach which I outlined in May (see
> > http://forum.openvz.org/index.php?t=msg&goto=30223&)? We agree on many
> > points. I think we basically solve the suid problem the same way.
>
> I hadn't read that post although I saw a part of it in your paper.
> Solving that problem so unprivileged users can use a user namespace
> seems to be the key to making namespaces more widely usable, and
> recursive. Plus I really like the idea of a super nobody user.
>
> > But I've moved away from a uid-to-uid mapping. Instead,
> > I expand on the relation you also describe: the user who creates a user
> > namespace owns the user namespace and introduce a persistant
> > namespace ID.
>
> I think there are management issues with a persistent namespace ID.

There are. But one key point is that the namespace ids are not
cryptographic keys. They don't need to be globally unique, or even 100%
unique on one system (though that gets too subtle).

I was wanting to keep them tiny - or at least variably sized - so they
could be stored with each inode.

> In particular: Is this user allowed to use this ID?

One way to address that is actually by having a system-wide tool with
CAP_SET_USERNS=fp which enforces a userid-to-unsid mapping. The host
sysadmin creates a table, say, 500:0 may use unsid 10-15. 500:0 (let's
call him hallyn) doesn't have CAP_SET_USERNS permissions himself, but
can run /bin/set_unsid, which runs with CAP_SET_USERNS and ensures that
hallyn uses only unsids 10-15.

It's not ideal. I'd rather have some sort of fancy collision-proof
global persistant id system, but again I think it's important that if
500:0 creates userns 1 and 400:1 creates userns 2, and 0:2 creates a

file, that the file be persistantly marked as belonging to
(500:0,400:1,0:2), distinct from another file created by
(500:0,400:1,1000:2). Which means these things have to be stored
per-inode, meaning they can't be too large.

> That is the reason I want to avoid having a generic persistent
> namespace ID. Implementing a common library and then modifying the
> filesystems to use it on a case by case basis sounds much more
> maintainable. Then picking the wrong direction does not bind us for
> all time and eternity.
>
> > I understand that you may reflexively balk at introducing a new global
> > persistant ID when we're focusing on turning everything into a
> > namespace, but I think that would be a misguided reflex (like the
> > ioctl->netlinke one of a few years ago). In particular, in your
> > approach the identifier is the combination of the uid-to-uid mapping and
> > the uids stored on the original filesystem.
>
> Not at all. I thought I had mentioned the xattr thing as one of the
> possibilities. I forgot the proper term so I probably said acls. The
> practical problem is that you then have to rev your quota support. To
> also support the xattr separation. In addition not every filesystem
> supports xattrs. Although the common ones do.

Again each fs could do whatever it wanted, and perhaps reiser would not
use xattrs but some funky reiserfs-ish thing :) I'm just talking about
xattrs bc that's what I'd use, on top of ext2/3, in any prototype.

Also, I think we're all agreed that for some filesystems, the semantics
of:

1. filesystem is owned by current->user->user_ns who mounted it
   (complicated by mounts propagation the same way as it
   was with user mounts)
2. access by user in another namespace == access by user nobody

make sense.

For quota support, I see where we'll have to check quota for each user
in the creator chain to which the task belongs - is that what you're
referring to?

> > I do think the particular form of the ID I suggest will be unsuitable
> > and we'll want something more flexible. Perhaps stick with the unsid
> > for the legacy filesystems with xattr-unsid support, and let advanced
> > filesystems like nfsv4, 9p, and smb use their own domain
> > identifiers.
>

> Which is why I said make it filesystem specific with support from a
> generic library. No prctl just a mount option.

Now when you say a generic library, are you referring to a userspace
daemon, queried by the kernel, which does uid translation? Do you have
a particular api in mind? What sort of commands and queries would the
kernel send to that daemon?

> > But since we seem to agree on the first part - introducing a hierarchy
> > between users and the namespaces they create - it sounds like the
> > following patch would suit both of us. (I just started implementing my
> > approach this past week in my free time). I'm not sending this as any
> > sort of request for inclusion, just bc it's sitting around...
>
> Yes. At least a loose hierarchy.
>
> It just occurred to me that with unix domain sockets, some signals, /proc
> we have user namespace crossing (child to parent) where we need to
> report the uid. In that case the simple solution appears to be to use
> the overflowuid and overflowgid that were defined as part of the 16bit
> to 32bit transition. Otherwise it would require mapping all of the
> child uids into the parent like we do with pids except using an upcall
> to userspace.

Again, in my proposal, each child user is also owned by the parent
user, so it should be possible to send the uid at the right user
namespace level the way we do with find_task_by_vpid() for pid
namespaces.

I know, there are still places in the signal code where we
fire-and-forget before we know the target, so those would be a problem.
The whole issue of putting a uid in a siginfo, when we don't know the
target and don't have a lifecycle on a siginfo, is a problem I've been
dreading for a long time now.

> Making capabilities user namespace specific if we can.
>
> Did you have any problems with adding a user_namespace argument to
> capable?

I had started that in a much earlier patchset a year or two ago. Calls
to capable(CAP_KILL) and capable(CAP_DAC_OVERRIDE) needed to be changed
to new functions accepting a target file/task. I don't recall it being
a big deal actually. Far more interesting is what to do with the
other capabilities, like CAP_NET_ADMIN.

As for bounding sets, imo we don't bother the user namespace code with
capabilities at all. We just let the admin specify bounding sets for

containers. For starters we recommend always pulling out things like
CAP_NET_ADMIN, then we can take our time discussing their safety. (i.e.
even if I don't have CAP_NET_ADMIN, if I clone(CLONE_NEWNET), is there
any reason not to give me CAP_NET_ADMIN in the result? After all I need
a capable parent in another namespace to either pass me a device or tie
any veth devices I create into a bridge before I can do any damage...)

> Just skimming through your patch I don't expect we will need the list
> of children, and not having should reduct our locking burden.

Hmm, that's true. I can't see a reason for that. Thanks!

So can you send some more info on the API you envision for the
library/daemon?

thanks,
-serge

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: design of user namespaces
Posted by serue on Fri, 20 Jun 2008 21:47:46 GMT
View Forum Message <> Reply to Message

Quoting Serge E. Hallyn (serue@us.ibm.com):
> > Just skimming through your patch I don't expect we will need the list
> > of children, and not having should reduct our locking burden.
>
> Hmm, that's true. I can't see a reason for that. Thanks!

BTW here is the new, slightly smaller patch:

>From d17fbd87d97f64a0e879a7efbe5e1835fc573eae Mon Sep 17 00:00:00 2001
From: Serge Hallyn <serge@us.ibm.com>
Date: Thu, 19 Jun 2008 20:18:17 -0500
Subject: [PATCH 1/1] user namespaces: introduce user_struct->user_namespace relationship

When a task does clone(CLONE_NEWNS), the task's user is the 'creator' of the
new user_namespace, and the user_namespace is tacked onto a list of those
created by this user.

When we create or put a user in a namespace, we also do so for all creator
users up the creator chain.

Changelog:

```
 Jun 20: Eric Biederman pointed out the sibling/child_user_ns
  list is unnecessary!

Signed-off-by: Serge Hallyn <serge@us.ibm.com>
---
 include/linux/sched.h          |    1 +
 include/linux/user_namespace.h |    1 +
 kernel/user.c                  |   66 ++++++++++++++++++++++++++++++++++++++++++++-
 kernel/user_namespace.c        |   15 ++++++----
 4 files changed, 72 insertions(+), 11 deletions(-)

diff --git a/include/linux/sched.h b/include/linux/sched.h
index 799bbdd..da1bcc6 100644
--- a/include/linux/sched.h
+++ b/include/linux/sched.h
@@ -604,6 +604,7 @@ struct user_struct {
  /* Hash table maintenance information */
  struct hlist_node uidhash_node;
  uid_t uid;
+ struct user_namespace *user_namespace;

 #ifdef CONFIG_USER_SCHED
  struct task_group *tg;
diff --git a/include/linux/user_namespace.h b/include/linux/user_namespace.h
index b5f41d4..f9477c3 100644
--- a/include/linux/user_namespace.h
+++ b/include/linux/user_namespace.h
@@ -13,6 +13,7 @@ struct user_namespace {
  struct kref  kref;
  struct hlist_head uidhash_table[UIDHASH_SZ];
  struct user_struct *root_user;
+ struct user_struct *creator;
 };

 extern struct user_namespace init_user_ns;
diff --git a/kernel/user.c b/kernel/user.c
index 865ecf5..e583be4 100644
--- a/kernel/user.c
+++ b/kernel/user.c
@@ -21,6 +21,7 @@ struct user_namespace init_user_ns = {
  .kref = {
   .refcount = ATOMIC_INIT(2),
  },
+ .creator = &root_user,
  .root_user = &root_user,
 };
 EXPORT_SYMBOL_GPL(init_user_ns);
@@ -53,6 +54,7 @@ struct user_struct root_user = {
```

```
  .files  = ATOMIC_INIT(0),
  .sigpending = ATOMIC_INIT(0),
  .locked_shm     = 0,
+ .user_namespace = &init_user_ns,
 #ifdef CONFIG_USER_SCHED
  .tg  = &init_task_group,
 #endif
@@ -71,6 +73,18 @@ static void uid_hash_remove(struct user_struct *up)
  hlist_del_init(&up->uidhash_node);
 }

+void inc_user_and_creators(struct user_struct *user)
+{
+ struct user_namespace *ns = user->user_namespace;
+ while (user) {
+  atomic_inc(&user->__count);
+  if (ns == ns->creator->user_namespace)
+   return;
+  user = ns->creator;
+  ns = user->user_namespace;
+ }
+}
+
 static struct user_struct *uid_hash_find(uid_t uid, struct hlist_head *hashent)
 {
  struct user_struct *user;
@@ -78,7 +92,7 @@ static struct user_struct *uid_hash_find(uid_t uid, struct hlist_head
*hashent)

  hlist_for_each_entry(user, h, hashent, uidhash_node) {
   if (user->uid == uid) {
-   atomic_inc(&user->__count);
+   inc_user_and_creators(user);
    return user;
   }
  }
@@ -315,12 +329,30 @@ done:
  uids_mutex_unlock();
 }

+/*
+ * Decrement use counts for all namespace ancestors of the user
+ * being freed.  The user itself has already been dec'ed, so
+ * we only start at its creator.
+ */
+void dec_creators(struct user_struct *user)
+{
+ struct user_namespace *ns = user->user_namespace;
```

```
+ while (ns != ns->creator->user_namespace) {
+  user = ns->creator;
+  atomic_dec(&user->__count);
+  ns = user->user_namespace;
+ }
+}
+
 /* IRQs are disabled and uidhash_lock is held upon function entry.
  * IRQ state (as stored in flags) is restored and uidhash_lock released
  * upon function exit.
  */
 static inline void free_user(struct user_struct *up, unsigned long flags)
 {
+ /* decrement all creator counts */
+ dec_creators(up);
+
  /* restore back the count */
  atomic_inc(&up->__count);
  spin_unlock_irqrestore(&uidhash_lock, flags);
@@ -409,6 +441,8 @@ struct user_struct *alloc_uid(struct user_namespace *ns, uid_t uid)
   if (sched_create_user(new) < 0)
    goto out_free_user;

+  new->user_namespace = ns;
+
   if (uids_user_create(new))
    goto out_destoy_sched;

@@ -429,6 +463,7 @@ struct user_struct *alloc_uid(struct user_namespace *ns, uid_t uid)
    kmem_cache_free(uid_cachep, new);
   } else {
    uid_hash_insert(new, hashent);
+   inc_user_and_creators(new);
    up = new;
   }
   spin_unlock_irq(&uidhash_lock);
@@ -448,6 +483,35 @@ out_unlock:
  return NULL;
 }

+/*
+ * After doing clone(CLONE_NEWUSER), the new task continues to hold
+ * a refcount on ancestor users, but just switches to the new
+ * root user in the child namespace
+ */
+void switch_uid_for_created_root(struct user_struct *new_user)
+{
+ struct user_struct *old_user;
```

```
+
+ old_user = current->user;
+ atomic_inc(&new_user->processes);
+ switch_uid_keyring(new_user);
+ current->user = new_user;
+ sched_switch_user(current);
+
+ /*
+  * We need to synchronize with __sigqueue_alloc()
+  * doing a get_uid(p->user).. If that saw the old
+  * user value, we need to wait until it has exited
+  * its critical region before we can free the old
+  * structure.
+  */
+ smp_mb();
+ spin_unlock_wait(&current->sighand->siglock);
+
+ free_uid(old_user);
+ suid_keys(current);
+}
+
 void switch_uid(struct user_struct *new_user)
 {
  struct user_struct *old_user;
diff --git a/kernel/user_namespace.c b/kernel/user_namespace.c
index a9ab059..45ba675 100644
--- a/kernel/user_namespace.c
+++ b/kernel/user_namespace.c
@@ -11,6 +11,7 @@
 #include <linux/slab.h>
 #include <linux/user_namespace.h>

+extern void switch_uid_for_created_root(struct user_struct *new_user);
 /*
  * Clone a new ns copying an original user ns, setting refcount to 1
  * @old_ns: namespace to clone
@@ -19,7 +20,6 @@
 static struct user_namespace *clone_user_ns(struct user_namespace *old_ns)
 {
  struct user_namespace *ns;
- struct user_struct *new_user;
  int n;

  ns = kmalloc(sizeof(struct user_namespace), GFP_KERNEL);
@@ -31,6 +31,9 @@ static struct user_namespace *clone_user_ns(struct user_namespace
*old_ns)
  for (n = 0; n < UIDHASH_SZ; ++n)
   INIT_HLIST_HEAD(ns->uidhash_table + n);
```

```
+ /* set up owner/parent relationship */
+ ns->creator = current->user;
+
  /* Insert new root user.  */
  ns->root_user = alloc_uid(ns, 0);
  if (!ns->root_user) {
@@ -38,15 +41,7 @@ static struct user_namespace *clone_user_ns(struct user_namespace
*old_ns)
   return ERR_PTR(-ENOMEM);
  }

- /* Reset current->user with a new one */
- new_user = alloc_uid(ns, current->uid);
- if (!new_user) {
-  free_uid(ns->root_user);
-  kfree(ns);
-  return ERR_PTR(-ENOMEM);
- }
-
- switch_uid(new_user);
+ switch_uid_for_created_root(ns->root_user);
  return ns;
 }

--
1.5.4.3
```

Subject: Re: design of user namespaces
Posted by ebiederm on Fri, 20 Jun 2008 23:00:12 GMT
View Forum Message <> Reply to Message

"Serge E. Hallyn" <serue@us.ibm.com> writes:

> There are. But one key point is that the namespace ids are not
> cryptographic keys. They don't need to be globally unique, or even 100%
> unique on one system (though that gets too subtle).
>
> I was wanting to keep them tiny - or at least variably sized - so they
> could be stored with each inode.

Sure. I think they make a lot of sense. idmapd uses domain names

for this purpose. At the moment I just don't think the are necessary.
Like veth isn't necessary for network namespaces. Ultimately we
will use these identifiers all of the time but it doesn't mean
the generic code has to deal with them.

>> In particular: Is this user allowed to use this ID?
>
> One way to address that is actually by having a system-wide tool with
> CAP_SET_USERNS=fp which enforces a userid-to-unsid mapping. The host
> sysadmin creates a table, say, 500:0 may use unsid 10-15. 500:0 (let's
> call him hallyn) doesn't have CAP_SET_USERNS permissions himself, but
> can run /bin/set_unsid, which runs with CAP_SET_USERNS and ensures that
> hallyn uses only unsids 10-15.
>
> It's not ideal. I'd rather have some sort of fancy collision-proof
> global persistant id system, but again I think it's important that if
> 500:0 creates userns 1 and 400:1 creates userns 2, and 0:2 creates a
> file, that the file be persistantly marked as belonging to
> (500:0,400:1,0:2), distinct from another file created by
> (500:0,400:1,1000:2). Which means these things have to be stored
> per-inode, meaning they can't be too large.

So my suggestion was something like this:
mount -o nativemount,uidns=1 /

Then the filesystem performs magic to ask if the owner of user namespace
is allowed to use uidns 1. That magic would consult a config file like:

```
[domains]
  local1.mydomain  1
  local2.mydomain  2
  local3.mydomain  3

[users]
  bob   local1.mydomain
  bob   local3.mydomain
  nancy local2.mydomain
```

Or something like that. Reporting which users are allowed to use
which userid namespaces, and the mapping of those userid namespaces
to something compact for storing in the filesystem.

The magic could be an upcall to userspace.
The magic could be loading the configuration file at mount time.
The magic could be storing the config file in the filesystem
and having special commands to access it like quotas.

The very important points are that it is a remount of an existing mount
so that we don't have to worry about corrupted filesystem attacks, and
that authentication is performed at mount time.

I just think that once we get to the point of specifying the parameters at
mount time. There is no need for generic kernel configuration of a
uidns name.

>>
>> > I understand that you may reflexively balk at introducing a new global
>> > persistant ID when we're focusing on turning everything into a
>> > namespace, but I think that would be a misguided reflex (like the
>> > ioctl->netlinke one of a few years ago). In particular, in your
>> > approach the identifier is the combination of the uid-to-uid mapping and
>> > the uids stored on the original filesystem.
>>
>> Not at all. I thought I had mentioned the xattr thing as one of the
>> possibilities. I forgot the proper term so I probably said acls. The
>> practical problem is that you then have to rev your quota support. To
>> also support the xattr separation. In addition not every filesystem
>> supports xattrs. Although the common ones do.
>
> Again each fs could do whatever it wanted, and perhaps reiser would not
> use xattrs but some funky reiserfs-ish thing :) I'm just talking about
> xattrs bc that's what I'd use, on top of ext2/3, in any prototype.

Sounds good to me. Each fs doing whatever it wants so seems the most
sensible approach.

> Also, I think we're all agreed that for some filesystems, the semantics
> of:
>
> 1. filesystem is owned by current->user->user_ns who mounted it
>   (complicated by mounts propagation the same way as it
>   was with user mounts)
Yes.
> 2. access by user in another namespace == access by user nobody
>
> make sense.

yes.

> For quota support, I see where we'll have to check quota for each user
> in the creator chain to which the task belongs - is that what you're
> referring to?

Actually no. Although that may eventually come up. At that level user namespaces
may be completely disjoint. Mostly I was referring to the fact that quotas

as I understand them are per filesystem per uid, and don't take the xattr into
account. So you conflict with multiple uid namespaces when you reuse the uid.

>> > I do think the particular form of the ID I suggest will be unsuitable
>> > and we'll want something more flexible. Perhaps stick with the unsid
>> > for the legacy filesystems with xattr-unsid support, and let advanced
>> > filesystems like nfsv4, 9p, and smb use their own domain
>> > identifiers.
>>
>> Which is why I said make it filesystem specific with support from a
>> generic library. No prctl just a mount option.
>
> Now when you say a generic library, are you referring to a userspace
> daemon, queried by the kernel, which does uid translation? Do you have
> a particular api in mind? What sort of commands and queries would the
> kernel send to that daemon?

I was mostly thinking of something like jbd. That would allow posix
filesystems that all want to implement uid mappings the same way to
use the same code. At which point we can also reuse the same user
space support.

>> > But since we seem to agree on the first part - introducing a hierarchy
>> > between users and the namespaces they create - it sounds like the
>> > following patch would suit both of us. (I just started implementing my
>> > approach this past week in my free time). I'm not sending this as any
>> > sort of request for inclusion, just bc it's sitting around...
>>
>> Yes. At least a loose hierarchy.
>>
>> It just occurred to me that with unix domain sockets, some signals, /proc
>> we have user namespace crossing (child to parent) where we need to
>> report the uid. In that case the simple solution appears to be to use
>> the overflowuid and overflowgid that were defined as part of the 16bit
>> to 32bit transition. Otherwise it would require mapping all of the
>> child uids into the parent like we do with pids except using an upcall
>> to userspace.
>
> Again, in my proposal, each child user is also owned by the parent
> user, so it should be possible to send the uid at the right user
> namespace level the way we do with find_task_by_vpid() for pid
> namespaces.
>
> I know, there are still places in the signal code where we
> fire-and-forget before we know the target, so those would be a problem.
> The whole issue of putting a uid in a siginfo, when we don't know the
> target and don't have a lifecycle on a siginfo, is a problem I've been

> dreading for a long time now.

We actually do know the target at the time the signal is queued up.
We should solve this for the pid namespace first. The patches exist
they just need to get off their sorry buts and get themselves merged.

There are a few issues that cause me to wonder if the user namespace
will not be easy to unshare for similar reasons to the pid namespace.

As for which uid to use. Duh! We use the uid of the creator of the user
namespace. How I missed that one I know. We still have the case of
completely disjoint user namespaces. In which case we do want to use
the mostly reserved overflowuid. Some non 0 uid reserved to indicate
that there is no mapping for this uid in your user namespace. Although
we have just enough mappings we may be able to get away with returning
-EIO for all the other cases.

> I had started that in a much earlier patchset a year or two ago. Calls
> to capable(CAP_KILL) and capable(CAP_DAC_OVERRIDE) needed to be changed
> to new functions accepting a target file/task. I don't recall it being
> a big deal actually. Far more interesting is what to do with the
> other capabilities, like CAP_NET_ADMIN.
>
> As for bounding sets, imo we don't bother the user namespace code with
> capabilities at all. We just let the admin specify bounding sets for
> containers. For starters we recommend always pulling out things like
> CAP_NET_ADMIN, then we can take our time discussing their safety. (i.e.
> even if I don't have CAP_NET_ADMIN, if I clone(CLONE_NEWNET), is there
> any reason not to give me CAP_NET_ADMIN in the result? After all I need
> a capable parent in another namespace to either pass me a device or tie
> any veth devices I create into a bridge before I can do any damage...)

Ok. This part will probably take a little more discussion.
My viewpoint is different.

I think as a fundamental fact that all capabilities and other security
identifiers (selinux lables) should be local to the user namespace.

In the case of capabilities then the result is that a capability
is either potent or impotent in the user_namespace.

So CAP_SYS_RAWIO. Would only be potent in the initial user_namespace.

CAP_NET_ADMIN would only be potent on network namespaces created in
the user namespace (and the children of the user namespace).

Likewise for the other capabilities.

Therefore the first process in a user namespace (that only unshared
the user namespace) will have all capabilities and will simply not be
able to do anything with them.

So capable becomes something like:

```
int capable(struct user_namespace *ns, int cap)
{
 return __capable(current, cap);
}

int __capable(struct task_struct *task, struct user_namespace *ns, int cap)
{
        /* The check below should check the parent user namespaces too */
        if (task->nsproxy->user_ns != ns)
         return -EPERM;
 if (!cap_raised(task->cap_effective, cap))
         return -EPERM;
        return 0;
}
```

Calls to capable initially pass in init_user_ns. And then as we fixup
things like the network namespaces we call
"capable(net->user_ns, CAP_NET_ADMIN);"

Similarly for the other paths. That should be incremental and safe.

>> Just skimming through your patch I don't expect we will need the list
>> of children, and not having should reduct our locking burden.
>
> Hmm, that's true. I can't see a reason for that. Thanks!
>
> So can you send some more info on the API you envision for the
> library/daemon?

Let's talk about the functionality and the API should become obvious
as a logical consequence.

In the case of read only access I don't think we need any user space
configuration at all. Just a filesystem that is ok with it.

mount -o nativemount,ro /

Eric

Subject: Re: design of user namespaces
Posted by ebiederm on Fri, 20 Jun 2008 23:07:45 GMT
View Forum Message <> Reply to Message

"Serge E. Hallyn" <serue@us.ibm.com> writes:

> Quoting Serge E. Hallyn (serue@us.ibm.com):
>> > Just skimming through your patch I don't expect we will need the list
>> > of children, and not having should reduct our locking burden.
>>
>> Hmm, that's true. I can't see a reason for that. Thanks!
>
> BTW here is the new, slightly smaller patch:
>
>>From d17fbd87d97f64a0e879a7efbe5e1835fc573eae Mon Sep 17 00:00:00 2001
> From: Serge Hallyn <serge@us.ibm.com>
> Date: Thu, 19 Jun 2008 20:18:17 -0500
> Subject: [PATCH 1/1] user namespaces: introduce user_struct->user_namespace
> relationship
>
> When a task does clone(CLONE_NEWNS), the task's user is the 'creator' of the
> new user_namespace, and the user_namespace is tacked onto a list of those
> created by this user.
>
> When we create or put a user in a namespace, we also do so for all creator
> users up the creator chain.
>
> Changelog:
> Jun 20: Eric Biederman pointed out the sibling/child_user_ns
>   list is unnecessary!
>
> Signed-off-by: Serge Hallyn <serge@us.ibm.com>
> ---
> include/linux/sched.h | 1 +
> include/linux/user_namespace.h | 1 +
> kernel/user.c | 66 ++-
> kernel/user_namespace.c | 15 +++++------
> 4 files changed, 72 insertions(+), 11 deletions(-)
>
> diff --git a/include/linux/sched.h b/include/linux/sched.h
> index 799bbdd..da1bcc6 100644
> --- a/include/linux/sched.h
> +++ b/include/linux/sched.h
> @@ -604,6 +604,7 @@ struct user_struct {
>   /* Hash table maintenance information */
>   struct hlist_node uidhash_node;
>   uid_t uid;
> + struct user_namespace *user_namespace;
>

> #ifdef CONFIG_USER_SCHED
>   struct task_group *tg;
> diff --git a/include/linux/user_namespace.h b/include/linux/user_namespace.h
> index b5f41d4..f9477c3 100644
> --- a/include/linux/user_namespace.h
> +++ b/include/linux/user_namespace.h
> @@ -13,6 +13,7 @@ struct user_namespace {
>   struct kref kref;
>   struct hlist_head uidhash_table[UIDHASH_SZ];
>   struct user_struct *root_user;
> + struct user_struct *creator;
> };
>
> extern struct user_namespace init_user_ns;
> diff --git a/kernel/user.c b/kernel/user.c
> index 865ecf5..e583be4 100644
> --- a/kernel/user.c
> +++ b/kernel/user.c
> @@ -21,6 +21,7 @@ struct user_namespace init_user_ns = {
>   .kref = {
>   .refcount = ATOMIC_INIT(2),
>   },
> + .creator = &root_user,
>   .root_user = &root_user,
> };
> EXPORT_SYMBOL_GPL(init_user_ns);
> @@ -53,6 +54,7 @@ struct user_struct root_user = {
>   .files = ATOMIC_INIT(0),
>   .sigpending = ATOMIC_INIT(0),
>   .locked_shm   = 0,
> + .user_namespace = &init_user_ns,
> #ifdef CONFIG_USER_SCHED
>   .tg = &init_task_group,
> #endif
> @@ -71,6 +73,18 @@ static void uid_hash_remove(struct user_struct *up)
>   hlist_del_init(&up->uidhash_node);
> }
>
> +void inc_user_and_creators(struct user_struct *user)
> +{
> + struct user_namespace *ns = user->user_namespace;
> + while (user) {
> + atomic_inc(&user->__count);
> + if (ns == ns->creator->user_namespace)
> +   return;
> + user = ns->creator;
> + ns = user->user_namespace;
> + }

> +}
> +

This functionality appears unnecessary. Holding a count on the user
and the user holding a count on it's user_namespace and the user_namespace
holding a count on it's creator should be sufficient.

Or am I missing something?

Eric

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: design of user namespaces
Posted by serue on Sat, 21 Jun 2008 19:05:32 GMT
View Forum Message <> Reply to Message

Quoting Eric W. Biederman (ebiederm@xmission.com):
> "Serge E. Hallyn" <serue@us.ibm.com> writes:
>
> > Quoting Serge E. Hallyn (serue@us.ibm.com):
> >> > Just skimming through your patch I don't expect we will need the list
> >> > of children, and not having should reduct our locking burden.
> >>
> >> Hmm, that's true. I can't see a reason for that. Thanks!
> >
> > BTW here is the new, slightly smaller patch:
> >
> >>From d17fbd87d97f64a0e879a7efbe5e1835fc573eae Mon Sep 17 00:00:00 2001
> > From: Serge Hallyn <serge@us.ibm.com>
> > Date: Thu, 19 Jun 2008 20:18:17 -0500
> > Subject: [PATCH 1/1] user namespaces: introduce user_struct->user_namespace
> > relationship
> >
> > When a task does clone(CLONE_NEWNS), the task's user is the 'creator' of the
> > new user_namespace, and the user_namespace is tacked onto a list of those
> > created by this user.
> >
> > When we create or put a user in a namespace, we also do so for all creator
> > users up the creator chain.
> >
> > Changelog:
> > Jun 20: Eric Biederman pointed out the sibling/child_user_ns
> >   list is unnecessary!
> >

> > Signed-off-by: Serge Hallyn <serge@us.ibm.com>
> > ---
> > include/linux/sched.h | 1 +
> > include/linux/user_namespace.h | 1 +
> > kernel/user.c | 66 ++-
> > kernel/user_namespace.c | 15 +++++------
> > 4 files changed, 72 insertions(+), 11 deletions(-)
> >
> > diff --git a/include/linux/sched.h b/include/linux/sched.h
> > index 799bbdd..da1bcc6 100644
> > --- a/include/linux/sched.h
> > +++ b/include/linux/sched.h
> > @@ -604,6 +604,7 @@ struct user_struct {
> >   /* Hash table maintenance information */
> >   struct hlist_node uidhash_node;
> >   uid_t uid;
> > + struct user_namespace *user_namespace;
> >
> > #ifdef CONFIG_USER_SCHED
> >   struct task_group *tg;
> > diff --git a/include/linux/user_namespace.h b/include/linux/user_namespace.h
> > index b5f41d4..f9477c3 100644
> > --- a/include/linux/user_namespace.h
> > +++ b/include/linux/user_namespace.h
> > @@ -13,6 +13,7 @@ struct user_namespace {
> >   struct kref kref;
> >   struct hlist_head uidhash_table[UIDHASH_SZ];
> >   struct user_struct *root_user;
> > + struct user_struct *creator;
> > };
> >
> > extern struct user_namespace init_user_ns;
> > diff --git a/kernel/user.c b/kernel/user.c
> > index 865ecf5..e583be4 100644
> > --- a/kernel/user.c
> > +++ b/kernel/user.c
> > @@ -21,6 +21,7 @@ struct user_namespace init_user_ns = {
> >   .kref = {
> >   .refcount = ATOMIC_INIT(2),
> >   },
> > + .creator = &root_user,
> >   .root_user = &root_user,
> > };
> > EXPORT_SYMBOL_GPL(init_user_ns);
> > @@ -53,6 +54,7 @@ struct user_struct root_user = {
> >   .files = ATOMIC_INIT(0),
> >   .sigpending = ATOMIC_INIT(0),
> >   .locked_shm   = 0,

> > + .user_namespace = &init_user_ns,
> > #ifdef CONFIG_USER_SCHED
> >   .tg = &init_task_group,
> > #endif
> > @@ -71,6 +73,18 @@ static void uid_hash_remove(struct user_struct *up)
> >   hlist_del_init(&up->uidhash_node);
> > }
> >
> > +void inc_user_and_creators(struct user_struct *user)
> > +{
> > + struct user_namespace *ns = user->user_namespace;
> > + while (user) {
> > + atomic_inc(&user->__count);
> > + if (ns == ns->creator->user_namespace)
> > +   return;
> > + user = ns->creator;
> > + ns = user->user_namespace;
> > + }
> > +}
> > +
>
> This functionality appears unnecessary. Holding a count on the user
> and the user holding a count on it's user_namespace and the user_namespace
> holding a count on it's creator should be sufficient.
>
> Or am I missing something?

Argh. No I don't think you're missing anything. You're absolutely
right.

thanks,
-serge

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: design of user namespaces
Posted by serue on Mon, 30 Jun 2008 21:13:25 GMT
View Forum Message <> Reply to Message

Quoting Eric W. Biederman (ebiederm@xmission.com):
> "Serge E. Hallyn" <serue@us.ibm.com> writes:
>
> > There are. But one key point is that the namespace ids are not
> > cryptographic keys. They don't need to be globally unique, or even 100%
> > unique on one system (though that gets too subtle).

> >
> > I was wanting to keep them tiny - or at least variably sized - so they
> > could be stored with each inode.
>
> Sure. I think they make a lot of sense. idmapd uses domain names
> for this purpose. At the moment I just don't think the are necessary.
> Like veth isn't necessary for network namespaces. Ultimately we
> will use these identifiers all of the time but it doesn't mean
> the generic code has to deal with them.
>
>
> >> In particular: Is this user allowed to use this ID?
> >
> > One way to address that is actually by having a system-wide tool with
> > CAP_SET_USERNS=fp which enforces a userid-to-unsid mapping. The host
> > sysadmin creates a table, say, 500:0 may use unsid 10-15. 500:0 (let's
> > call him hallyn) doesn't have CAP_SET_USERNS permissions himself, but
> > can run /bin/set_unsid, which runs with CAP_SET_USERNS and ensures that
> > hallyn uses only unsids 10-15.
> >
> > It's not ideal. I'd rather have some sort of fancy collision-proof
> > global persistant id system, but again I think it's important that if
> > 500:0 creates userns 1 and 400:1 creates userns 2, and 0:2 creates a
> > file, that the file be persistantly marked as belonging to
> > (500:0,400:1,0:2), distinct from another file created by
> > (500:0,400:1,1000:2). Which means these things have to be stored
> > per-inode, meaning they can't be too large.
>
> So my suggestion was something like this:
> mount -o nativemount,uidns=1 /
>
> Then the filesystem performs magic to ask if the owner of user namespace
> is allowed to use uidns 1. That magic would consult a config file like:
>
> [domains]
> local1.mydomain 1
> local2.mydomain 2
> local3.mydomain 3
>
> [users]
> bob local1.mydomain
> bob local3.mydomain
> nancy local2.mydomain
>
> Or something like that. Reporting which users are allowed to use
> which userid namespaces, and the mapping of those userid namespaces
> to something compact for storing in the filesystem.
>

> The magic could be an upcall to userspace.
> The magic could be loading the configuration file at mount time.
> The magic could be storing the config file in the filesystem
> and having special commands to access it like quotas.
>
> The very important points are that it is a remount of an existing mount
> so that we don't have to worry about corrupted filesystem attacks, and
> that authentication is performed at mount time.

Conceptually that (making corrupted fs attacks a non-issue) is
wonderful. Practically, I may be missing something: When you say
remount, it seems you must either mean a bind mount or a remount. If
remount, then that will want to change superblock flags. If the
child userns(+child mntns) does a real remount, then that will change
the flags for the parent ns as well, right?

If instead we do a bind mount we don't have that problem, but then the
fs can't be the one doing the user namespace work.

I'm probably missing something...

> I just think that once we get to the point of specifying the parameters at
> mount time. There is no need for generic kernel configuration of a
> uidns name.

thanks,
-serge

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: design of user namespaces
Posted by ebiederm on Tue, 01 Jul 2008 07:35:33 GMT
View Forum Message <> Reply to Message

"Serge E. Hallyn" <serue@us.ibm.com> writes:

> Quoting Eric W. Biederman (ebiederm@xmission.com):
>>
>> The very important points are that it is a remount of an existing mount
>> so that we don't have to worry about corrupted filesystem attacks, and
>> that authentication is performed at mount time.
>
> Conceptually that (making corrupted fs attacks a non-issue) is
> wonderful. Practically, I may be missing something: When you say
> remount, it seems you must either mean a bind mount or a remount. If

> remount, then that will want to change superblock flags. If the
> child userns(+child mntns) does a real remount, then that will change
> the flags for the parent ns as well, right?
>
> If instead we do a bind mount we don't have that problem, but then the
> fs can't be the one doing the user namespace work.
>
> I'm probably missing something.

Essentially I am creating a new mount operation that is a
cousin of a remount.

Unlike a real remount you can't change the super flags.
Unlike a bind mount you get the fs involved, and you pass in a string of flags
that the fs can interpret in a standard way.

I expect the flags you pass in would be a subset of what is allowed
in a normal remount.

Which is why I was calling it nativemount. Although usernsmount
may be better.

Eric

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: design of user namespaces
Posted by serue on Mon, 07 Jul 2008 15:24:06 GMT
View Forum Message <> Reply to Message

Quoting Eric W. Biederman (ebiederm@xmission.com):
> "Serge E. Hallyn" <serue@us.ibm.com> writes:
>
> > Quoting Eric W. Biederman (ebiederm@xmission.com):
> >>
> >> The very important points are that it is a remount of an existing mount
> >> so that we don't have to worry about corrupted filesystem attacks, and
> >> that authentication is performed at mount time.
> >
> > Conceptually that (making corrupted fs attacks a non-issue) is
> > wonderful. Practically, I may be missing something: When you say
> > remount, it seems you must either mean a bind mount or a remount. If
> > remount, then that will want to change superblock flags. If the
> > child userns(+child mntns) does a real remount, then that will change
> > the flags for the parent ns as well, right?

> >
> > If instead we do a bind mount we don't have that problem, but then the
> > fs can't be the one doing the user namespace work.
> >
> > I'm probably missing something.
>
> Essentially I am creating a new mount operation that is a
> cousin of a remount.
>
> Unlike a real remount you can't change the super flags.
> Unlike a bind mount you get the fs involved, and you pass in a string of flags
> that the fs can interpret in a standard way.
>
> I expect the flags you pass in would be a subset of what is allowed
> in a normal remount.
>
> Which is why I was calling it nativemount. Although usernsmount
> may be better.
>
> Eric

Ah, ok.

Now you haven't started any sort of coding for this yet, right? I'm
hoping to get some time later this week to think about/play with this.

-serge

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: design of user namespaces
Posted by ebiederm on Mon, 07 Jul 2008 19:25:23 GMT
View Forum Message <> Reply to Message

"Serge E. Hallyn" <serue@us.ibm.com> writes:
> Ah, ok.
>
> Now you haven't started any sort of coding for this yet, right? I'm
> hoping to get some time later this week to think about/play with this.

Not yet. I'm hoping to do something in the near future, as this seems
to be a killer feature though.

Eric

_____