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
```