
Subject: [PATCH 5/9] Allow ptrace from non-init user namespaces

Posted by [serge](#) on Thu, 17 Feb 2011 15:03:33 GMT

[View Forum Message](#) <> [Reply to Message](#)

ptrace is allowed to tasks in the same user namespace according to the usual rules (i.e. the same rules as for two tasks in the init user namespace). ptrace is also allowed to a user namespace to which the current task has CAP_SYS_PTRACE capability.

Changelog:

Dec 31: Address feedback by Eric:

- . Correct ptrace uid check
- . Rename may_ptrace_ns to ptrace_capable
- . Also fix the cap_ptrace checks.

Jan 1: Use const cred struct

Jan 11: use task_ns_capable() in place of ptrace_capable().

Signed-off-by: Serge E. Hallyn <serge.hallyn@canonical.com>

```
include/linux/capability.h    |  2 +
include/linux/user_namespace.h |  9 +++++++
kernel/ptrace.c                | 27 ++++++++-----
kernel/user_namespace.c        | 16 ++++++++
security/commoncap.c           | 48 ++++++++-----
5 files changed, 82 insertions(+), 20 deletions(-)
```

diff --git a/include/linux/capability.h b/include/linux/capability.h

index cb3d2d9..bc0f262 100644

--- a/include/linux/capability.h

+++ b/include/linux/capability.h

@@ -546,6 +546,8 @@ extern const kernel_cap_t __cap_init_eff_set;

*/

#define has_capability(t, cap) (security_real_capable((t), &init_user_ns, (cap)) == 0)

+#define has_ns_capability(t, ns, cap) (security_real_capable((t), (ns), (cap)) == 0)

+

/**

* has_capability_noaudit - Determine if a task has a superior capability available (unaudited)

* @t: The task in question

diff --git a/include/linux/user_namespace.h b/include/linux/user_namespace.h

index faf4679..862fc59 100644

--- a/include/linux/user_namespace.h

+++ b/include/linux/user_namespace.h

@@ -39,6 +39,9 @@ static inline void put_user_ns(struct user_namespace *ns)

uid_t user_ns_map_uid(struct user_namespace *to, const struct cred *cred, uid_t uid);

gid_t user_ns_map_gid(struct user_namespace *to, const struct cred *cred, gid_t gid);

+int same_or_ancestor_user_ns(struct task_struct *task,

```

+ struct task_struct *victim);
+
+ #else

static inline struct user_namespace *get_user_ns(struct user_namespace *ns)
@@ -66,6 +69,12 @@ static inline gid_t user_ns_map_gid(struct user_namespace *to,
    return gid;
}

+static inline int same_or_ancestor_user_ns(struct task_struct *task,
+ struct task_struct *victim)
+{
+ return 1;
+}
+
+ #endif

#endif /* _LINUX_USER_H */
diff --git a/kernel/ptrace.c b/kernel/ptrace.c
index 1708b1e..cde4655 100644
--- a/kernel/ptrace.c
+++ b/kernel/ptrace.c
@@ -134,21 +134,24 @@ int __ptrace_may_access(struct task_struct *task, unsigned int mode)
    return 0;
    rcu_read_lock();
    tcred = __task_cred(task);
- if ((cred->uid != tcred->euid ||
-      cred->uid != tcred->suid ||
-      cred->uid != tcred->uid ||
-      cred->gid != tcred->egid ||
-      cred->gid != tcred->sgid ||
-      cred->gid != tcred->gid) &&
-      !capable(CAP_SYS_PTRACE)) {
- rcu_read_unlock();
- return -EPERM;
- }
+ if (cred->user->user_ns == tcred->user->user_ns &&
+      (cred->uid == tcred->euid &&
+       cred->uid == tcred->suid &&
+       cred->uid == tcred->uid &&
+       cred->gid == tcred->egid &&
+       cred->gid == tcred->sgid &&
+       cred->gid == tcred->gid))
+ goto ok;
+ if (ns_capable(tcred->user->user_ns, CAP_SYS_PTRACE))
+ goto ok;
+ rcu_read_unlock();
+ return -EPERM;

```

```

+ok:
    rcu_read_unlock();
    smp_rmb();
    if (task->mm)
        dumpable = get_dumpable(task->mm);
- if (!dumpable && !capable(CAP_SYS_PTRACE))
+ if (!dumpable && !task_ns_capable(task, CAP_SYS_PTRACE))
    return -EPERM;

    return security_ptrace_access_check(task, mode);
@@ -198,7 +201,7 @@ int ptrace_attach(struct task_struct *task)
    goto unlock_tasklist;

    task->ptrace = PT_PTRACED;
- if (capable(CAP_SYS_PTRACE))
+ if (task_ns_capable(task, CAP_SYS_PTRACE))
    task->ptrace |= PT_PTRACE_CAP;

    __ptrace_link(task, current);
diff --git a/kernel/user_namespace.c b/kernel/user_namespace.c
index 9da289c..0ef2258 100644
--- a/kernel/user_namespace.c
+++ b/kernel/user_namespace.c
@@ -129,6 +129,22 @@ gid_t user_ns_map_gid(struct user_namespace *to, const struct cred
 *cred, gid_t
    return overflowgid;
}

+int same_or_ancestor_user_ns(struct task_struct *task,
+ struct task_struct *victim)
+{
+ struct user_namespace *u1 = task_cred_xxx(task, user)->user_ns;
+ struct user_namespace *u2 = task_cred_xxx(victim, user)->user_ns;
+ for (;;) {
+ if (u1 == u2)
+ return 1;
+ if (u1 == &init_user_ns)
+ return 0;
+ u1 = u1->creator->user_ns;
+ }
+ /* We never get here */
+ return 0;
+}
+
+static __init int user_namespaces_init(void)
+{
+ user_ns_cachep = KMEM_CACHE(user_namespace, SLAB_PANIC);
diff --git a/security/commoncap.c b/security/commoncap.c

```

index 51fa9ec..12ff65c 100644

--- a/security/commoncap.c

+++ b/security/commoncap.c

@@ -130,18 +130,34 @@ int cap_settime(struct timespec *ts, struct timezone *tz)

* @child: The process to be accessed

* @mode: The mode of attachment.

*

+ * If we are in the same or an ancestor user_ns and have all the target

+ * task's capabilities, then ptrace access is allowed.

+ * If we have the ptrace capability to the target user_ns, then ptrace

+ * access is allowed.

+ * Else denied.

+ *

* Determine whether a process may access another, returning 0 if permission

* granted, -ve if denied.

*/

int cap_ptrace_access_check(struct task_struct *child, unsigned int mode)

{

int ret = 0;

+ const struct cred *cred, *tcred;

rcu_read_lock();

- if (!cap_issubset(__task_cred(child)->cap_permitted,

- current_cred()->cap_permitted) &&

- !capable(CAP_SYS_PTRACE))

- ret = -EPERM;

+ cred = current_cred();

+ tcred = __task_cred(child);

+ /*

+ * The ancestor user_ns check may be gratuitous, as I think

+ * we've already guaranteed that in kernel/ptrace.c.

+ */

+ if (same_or_ancestor_user_ns(current, child) &&

+ cap_issubset(tcred->cap_permitted, cred->cap_permitted))

+ goto out;

+ if (ns_capable(tcred->user->user_ns, CAP_SYS_PTRACE))

+ goto out;

+ ret = -EPERM;

+out:

rcu_read_unlock();

return ret;

}

@@ -150,18 +166,34 @@ int cap_ptrace_access_check(struct task_struct *child, unsigned int mode)

* cap_ptrace_traceme - Determine whether another process may trace the current

* @parent: The task proposed to be the tracer

*

+ * If parent is in the same or an ancestor user_ns and has all current's

```

+ * capabilities, then ptrace access is allowed.
+ * If parent has the ptrace capability to current's user_ns, then ptrace
+ * access is allowed.
+ * Else denied.
+ *
+ * Determine whether the nominated task is permitted to trace the current
+ * process, returning 0 if permission is granted, -ve if denied.
+ */
int cap_ptrace_traceme(struct task_struct *parent)
{
    int ret = 0;
+ const struct cred *cred, *tcred;

    rcu_read_lock();
- if (!cap_issubset(current_cred()->cap_permitted,
-   __task_cred(parent)->cap_permitted) &&
-   !has_capability(parent, CAP_SYS_PTRACE))
-     ret = -EPERM;
+ cred = __task_cred(parent);
+ tcred = current_cred();
+ /*
+  * The ancestor user_ns check may be gratuitous, as I think
+  * we've already guaranteed that in kernel/ptrace.c.
+  */
+ if (same_or_ancestor_user_ns(parent, current) &&
+     cap_issubset(tcred->cap_permitted, cred->cap_permitted))
+     goto out;
+ if (has_ns_capability(parent, tcred->user->user_ns, CAP_SYS_PTRACE))
+     goto out;
+ ret = -EPERM;
+out:
    rcu_read_unlock();
    return ret;
}
--
1.7.0.4

```

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