Subject: [PATCH 15/17] Pid-NS(V3) Address signals in pid namespaces.
Posted by Sukadev Bhattiprolu on Sat, 16 Jun 2007 23:05:05 GMT
View Forum Message <> Reply to Message

From: sukadev@linux.vnet.ibm.com
Subject: [PATCH 15/17] Pid-NS(V3) Address signals in pid namespaces.

With support for multiple pid namespaces, each pid namespace has a
separate child reaper and this process needs some special handling
of signals.

- The child reaper should appear like a normal process to other
  processes in its ancestor namespaces and so should be killable
  (or not) in the usual way.

    - The child reaper should receive, from processes in it's active
      and decendent namespaces, only those signals for which it has
      installed a signal handler.

- System-wide signals (eg: kill signum -1) from within a child namespace
  should only affect processes within that namespace and descendant
  namespaces. They should not be posted to processes in ancestor or
  sibling namespaces.

- If the sender of a signal does not have a pid_t in the receiver's
  namespace (eg: a process in init_pid_ns sends a signal to a process
  in a descendant namespace), the sender's pid should appear as 0
  in the signal's siginfo structure.

- Existing rules for SIGIO delivery still apply and a process can
  choose any other process in its namespace and descendant namespaces
  to receive the SIGIO signal.

  The following appears to be incorrect in the fcntl() man page for
  F_SETOWN.

> Sending a signal to the owner process (group) specified by
> F_SETOWN is subject to the same permissions checks as are
> described for kill(2), where the sending process is the one that
> employs F_SETOWN (but see BUGS below).

Current behavior is that the SIGIO signal is delivered on behalf of
the process that caused the event (eg: made data available on the
file) and not the process that called fcntl().

To implement the above requirements, we:

- Add a check in check_kill_permission() for a process within a
  namespace sending the fast-pathed, SIGKILL signal.

- We use a flag, SIGQUEUE_CINIT, to tell the container-init if
  a signal posted to its queue is from a process within its own
  namespace. The flag is set in send_signal() if a process attempts
  to send a signal to its container-init.

  The SIGQUEUE_CINIT flag is checked in get_signal_to_deliver() - if
  the flag is set, the sender is from within the namespace and the
  container-init can choose to ignore the signal.

  If the SIGQUEUE_CINIT flag is clear in get_signal_to_deliver(), the
  signal originated from an ancestor namespace and so the container-
  init honors the signal.

Signed-off-by: Sukadev Bhattiprolu <sukadev@us.ibm.com>
---
 include/linux/pid.h | 1
 include/linux/sched.h | 11 +++++
 include/linux/signal.h | 1
 kernel/pid.c | 22 ++++++++++
 kernel/signal.c | 101 +++-----
 5 files changed, 125 insertions(+), 11 deletions(-)

Index: lx26-22-rc4-mm2/kernel/pid.c
==========
--- lx26-22-rc4-mm2.orig/kernel/pid.c 2007-06-15 19:21:35.000000000 -0700
+++ lx26-22-rc4-mm2/kernel/pid.c 2007-06-15 19:22:42.000000000 -0700
@@ -554,6 +554,28 @@ struct task_struct * fastcall pid_task(s
 }

 /*
+ * Return TRUE if the task @p is visible in the pid namespace @ns
+ */
+int is_task_in_pid_ns(struct task_struct *p, struct pid_namespace *ns)
+{
+ int i;
+ struct upid *upid;
+ struct pid *pid = task_pid(p);
+
+ if (!pid)
+ return 0;
+
+ for (i = 0; i < pid->num_upids; i++) {
+ upid = &pid->upid_list[i];
+ if (upid->pid_ns == ns)
+   return 1;

```
+ }
+
+ return 0;
+}
+EXPORT_SYMBOL(is_task_in_pid_ns);
+
+/*
  * Must be called under rcu_read_lock() or with tasklist_lock read-held.
  */
 struct task_struct *find_task_by_pid_type(int type, int nr)
Index: lx26-22-rc4-mm2/kernel/signal.c
===================================================================
--- lx26-22-rc4-mm2.orig/kernel/signal.c 2007-06-15 19:04:53.000000000 -0700
+++ lx26-22-rc4-mm2/kernel/signal.c 2007-06-15 19:26:48.000000000 -0700
@@ -284,7 +284,8 @@ unblock_all_signals(void)
  spin_unlock_irqrestore(&current->sighand->siglock, flags);
 }

-static int collect_signal(int sig, struct sigpending *list, siginfo_t *info)
+static int collect_signal(int sig, struct sigpending *list, siginfo_t *info,
+  int *sig_cinit)
 {
  struct sigqueue *q, *first = NULL;
  int still_pending = 0;
@@ -308,6 +309,8 @@ static int collect_signal(int sig, struc
  if (first) {
   list_del_init(&first->list);
   copy_siginfo(info, &first->info);
+  if (first->flags & SIGQUEUE_CINIT)
+   *sig_cinit = 1;
   __sigqueue_free(first);
   if (!still_pending)
    sigdelset(&list->signal, sig);
@@ -328,10 +331,11 @@ static int collect_signal(int sig, struc
 }

 static int __dequeue_signal(struct sigpending *pending, sigset_t *mask,
-   siginfo_t *info)
+   siginfo_t *info, int *sig_cinit)
 {
  int sig = next_signal(pending, mask);

+ *sig_cinit = 0;
  if (sig) {
   if (current->notifier) {
    if (sigismember(current->notifier_mask, sig)) {
@@ -342,7 +346,7 @@ static int __dequeue_signal(struct sigpe
    }
```

```
   }

-  if (!collect_signal(sig, pending, info))
+  if (!collect_signal(sig, pending, info, sig_cinit))
    sig = 0;
  }

@@ -350,17 +354,28 @@ static int __dequeue_signal(struct sigpe
 }

 /*
- * Dequeue a signal and return the element to the caller, which is
- * expected to free it.
+ * Dequeue a signal and update the @sig_cinit flag to say if the signal
+ * was posted to the container-init from within its pid namespace.
  *
- * All callers have to hold the siglock.
+ * TODO: Rather than use the  pointer-parameter, @sig_cinit, can we define
+ *   a wrapper around 'struct siginfo' so we can add fields to that
+ *   structure, without affecting user space ?
+ *
+ *       struct kern_siginfo {
+ *        struct signinfo *user_siginfo;
+ *        int flags;
+ *       }
+ *
+ *       Functions like dequeue_signal() could take kern_siginfo as a
+ *       parameter but still update user_siginfo when necessary ?
  */
-int dequeue_signal(struct task_struct *tsk, sigset_t *mask, siginfo_t *info)
+static int dequeue_signal_check_sig_cinit(struct task_struct *tsk,
+  sigset_t *mask, siginfo_t *info, int *sig_cinit)
 {
- int signr = __dequeue_signal(&tsk->pending, mask, info);
+ int signr = __dequeue_signal(&tsk->pending, mask, info, sig_cinit);
  if (!signr) {
   signr = __dequeue_signal(&tsk->signal->shared_pending,
-      mask, info);
+      mask, info, sig_cinit);
   /*
    * itimer signal ?
    *
@@ -419,6 +434,19 @@ int dequeue_signal(struct task_struct *t
 }

 /*
+ * Dequeue a signal and return the element to the caller, which is
+ * expected to free it.
```

```
+ *
+ * All callers have to hold the siglock.
+ */
+int dequeue_signal(struct task_struct *tsk, sigset_t *mask, siginfo_t *info)
+{
+ int sig_cinit;
+
+ return dequeue_signal_check_sig_cinit(tsk, mask, info, &sig_cinit);
+}
+
+/*
  * Tell a process that it has a new active signal..
  *
  * NOTE! we rely on the previous spin_lock to
@@ -500,6 +528,21 @@ static int rm_from_queue(unsigned long m
  return 1;
 }

+static int deny_signal_to_container_init(struct task_struct *t, int sig)
+{
+ /*
+  * If receiver is the container-init of sender and signal is SIGKILL
+  * reject it right-away. If signal is any other one, let the container
+  * init decide (in get_signal_to_deliver()) whether to handle it or
+  * ignore it.
+  */
+ if (is_container_init(t) && (sig == SIGKILL) &&
+    (task_active_pid_ns(current) == task_active_pid_ns(t)))
+  return -EPERM;
+
+ return 0;
+}
+
 /*
  * Bad permissions for sending the signal
  */
@@ -523,6 +566,10 @@ static int check_kill_permission(int sig
      && !capable(CAP_KILL))
   return error;

+ error = deny_signal_to_container_init(t, sig);
+ if (error)
+  return error;
+
  return security_task_kill(t, info, sig, 0);
 }

@@ -688,6 +735,25 @@ static int send_signal(int sig, struct s
```

```
    copy_siginfo(&q->info, info);
    break;
   }
+
+  /*
+   * If sender and receiver are from the same namespace and the
+   * receiver is the container-init, set the SIGQUEUE_CINIT so
+   * the container-init can choose to ignore the signal.
+   *
+   * If the container-init receives a signal from its ancestor
+   * namespace, it must honor the signal (so SIGQUEUE_CINIT
+   * flag should be clear). But since the sender does not have
+   * a pid in the receiver's namespace, set si_pid to 0.
+   */
+  if (is_same_active_pid_ns(t)) {
+   if (is_container_init(t)) {
+    q->flags |= SIGQUEUE_CINIT;
+   }
+  } else {
+   q->info.si_pid = 0;
+  }
+
  } else if (!is_si_special(info)) {
   if (sig >= SIGRTMIN && info->si_code != SI_USER)
   /*
@@ -1127,6 +1193,8 @@ EXPORT_SYMBOL_GPL(kill_pid_info_as_uid);
 static int kill_something_info(int sig, struct siginfo *info, int pid)
 {
  int ret;
+ struct pid_namespace *my_ns = task_active_pid_ns(current);
+
  rcu_read_lock();
  if (!pid) {
   ret = kill_pgrp_info(sig, info, task_pgrp(current));
@@ -1136,6 +1204,15 @@ static int kill_something_info(int sig,

   read_lock(&tasklist_lock);
   for_each_process(p) {
+   int in_ns = is_task_in_pid_ns(p, my_ns);
+
+   /*
+    * System-wide signals apply only to the sender's
+    * pid namespace, unless issued from init_pid_ns.
+    */
+   if (!in_ns)
+    continue;
+
    if (p->pid > 1 && p->tgid != current->tgid) {
```

```
     int err = group_send_sig_info(sig, info, p);
     ++count;
@@ -1756,6 +1833,7 @@ int get_signal_to_deliver(siginfo_t *inf
 {
  sigset_t *mask = &current->blocked;
  int signr = 0;
+ int sig_cinit;

  try_to_freeze();

@@ -1768,7 +1846,8 @@ relock:
       handle_group_stop())
     goto relock;

-  signr = dequeue_signal(current, mask, info);
+  signr = dequeue_signal_check_sig_cinit(current, mask, info,
+    &sig_cinit);

   if (!signr)
     break; /* will return 0 */
@@ -1829,7 +1908,7 @@ relock:
     * within that pid space. It can of course get signals from
     * its parent pid space.
     */
-  if (current == task_child_reaper(current))
+  if (sig_cinit)
     continue;

   if (sig_kernel_stop(signr)) {
Index: lx26-22-rc4-mm2/include/linux/pid.h
===================================================================
--- lx26-22-rc4-mm2.orig/include/linux/pid.h 2007-06-15 19:22:42.000000000 -0700
+++ lx26-22-rc4-mm2/include/linux/pid.h 2007-06-15 19:22:42.000000000 -0700
@@ -93,6 +93,7 @@ extern struct task_struct *FASTCALL(pid_
 extern struct task_struct *FASTCALL(get_pid_task(struct pid *pid,
       enum pid_type));

+extern int is_task_in_pid_ns(struct task_struct *tsk, struct pid_namespace *ns);
 extern struct pid *get_task_pid(struct task_struct *task, enum pid_type type);

 /*
Index: lx26-22-rc4-mm2/include/linux/signal.h
===================================================================
--- lx26-22-rc4-mm2.orig/include/linux/signal.h 2007-06-15 19:04:53.000000000 -0700
+++ lx26-22-rc4-mm2/include/linux/signal.h 2007-06-15 19:22:42.000000000 -0700
@@ -20,6 +20,7 @@ struct sigqueue {

 /* flags values. */
```

```
 #define SIGQUEUE_PREALLOC 1
+#define SIGQUEUE_CINIT  2

 struct sigpending {
  struct list_head list;
Index: lx26-22-rc4-mm2/include/linux/sched.h
===================================================================
--- lx26-22-rc4-mm2.orig/include/linux/sched.h 2007-06-15 19:22:42.000000000 -0700
+++ lx26-22-rc4-mm2/include/linux/sched.h 2007-06-15 19:22:42.000000000 -0700
@@ -1285,6 +1285,17 @@ static inline int is_global_init(struct
  return tsk == init_pid_ns.child_reaper;
 }

+/**
+ * Return TRUE if the active pid namespace of @tsk is same as active
+ * pid namespace of 'current'
+ *
+ * Note the difference between this function and is_task_in_pid_ns().
+ **/
+static inline int is_same_active_pid_ns(struct task_struct *t)
+{
+ return task_active_pid_ns(t) == task_active_pid_ns(current);
+}
+
 /*
  * is_container_init:
  * check whether in the task is init in it's own pid namespace.

--
```