Subject: Re: [PATCH] Signal semantics for /sbin/init
Posted by ebiederm on Tue, 30 Oct 2007 22:25:51 GMT
View Forum Message <> Reply to Message

sukadev@us.ibm.com writes:

> | This change by making the presence of a signal handler effectively
> | a permission check allows us to do all of the work before
> | we enqueue the signal. Which means that we can now allow
> | force_sig_info to send signals to init and that panic the kernel
> | instead of going into an infinite busy loop taking an exception
> | sending a signal and then retaking the same exception.
> |
> | This change also makes it possible to easily implement the
> | the desired semantics of /sbin/init for pid namespaces where
> | outer processes can kill init but processes inside the pid
> | namespace can not.
> |
> | Please take a look and tell me what you think.
>
> Overall, it looks good, couple of questions below. Will port my
> patches and test it out.
>
> |
> | diff --git a/kernel/signal.c b/kernel/signal.c
> | index 4537bdd..79856eb 100644
> | --- a/kernel/signal.c
> | +++ b/kernel/signal.c
> | @@ -546,6 +546,22 @@ static int check_kill_permission(int sig, struct siginfo
> *info,
> | return security_task_kill(t, info, sig, 0);
> | }
> |
> | +static int sig_available(struct task_struct *tsk, int sig)
>
> Hmm. IMHO, 'available' is not immediately obvious/clear.

Agreed. It was what I had to work with. I have since renamed it
sig_init_drop. Which is a little better.

> | +{
> | + void __user * handler;
> | + int available = 1;
> | +
> | + if (likely(!is_global_init(tsk)))
> | + goto out;
>
> With multiple pid namespaces, I guess, this would become

>
> if (!is_container_init(tsk))
>   goto out;

Although I need to make that tsk->group_leader.

> /* from parent namespace, don't ignore */
> if (!task_in_descendant_ns(tsk))
>   goto out;
>
> If this is correct, I have a question below re: do_sigaction.

> | +
> | + handler = tsk->sighand->action[sig-1].sa.sa_handler;
> | + available = (handler != SIG_IGN) &&
> | +   (handler != SIG_DFL);
>
> Can we use sig_user_defined() for the above checks ? sig_available()
> looks like an extension of sig_user_defined() for init.

Well the SIG_IGN check is actually wrong here. We should just check
for SIG_DFL if we want to maintain the most possible compatibility.

> How about
> sig_init_user_defined() or reverse the logic and use sig_init_ignore()
> like Oleg did ?

That could work.

> | +out:
> | + return available;
> | +}
> | +
> | +
> | /* forward decl */
> | static void do_notify_parent_cldstop(struct task_struct *tsk, int why);
> |
> | @@ -948,6 +964,9 @@ __group_send_sig_info(int sig, struct siginfo *info,
> struct task_struct *p)
> | int ret = 0;
> |
> | assert_spin_locked(&p->sighand->siglock);
> | + if (!sig_available(p, sig))
> | + return ret;
> | +
>
> | handle_stop_signal(sig, p);

> |
> | /* Short-circuit ignored signals. */
> | @@ -1379,6 +1398,11 @@ send_group_sigqueue(int sig, struct sigqueue *q, struct
> task_struct *p)
> | read_lock(&tasklist_lock);
> | /* Since it_lock is held, p->sighand cannot be NULL. */
> | spin_lock_irqsave(&p->sighand->siglock, flags);
> | + if (!sig_available(p, sig)) {
> | + ret = 1;
> | + goto out;
> | + }
> | +
> | handle_stop_signal(sig, p);
> |
> | /* Short-circuit ignored signals. */
>
> Hmm. I see now that Oleg's approach would result in the sig_init_ignore()
> check being done twice (once in handle_stop_signal() and once in the following
> sig_ignored()).
>
> Is that why you don't fold sig_available() into sig_ignored() ? Or, there
> other more important/correctness issues as well ?

It is a very slight but subtle point.
I have modified the definition so that signals with a handler SIG_DFL never
reach the init process not even to the pending mask. Essentially
this means we should drop them before any attempts at processing them.

So dropping the signals before handle_stop_signal is important.

As for not folding the signals into sig_ignore. We need to drop the signal
even if the signal is masked. So it looks like an ugly special case.

In addition by not dropping the signal to init everywhere (in particular
on the paths where we force a signal) we allow the kernel to kill init
and panic the system if /sbin/init does something nasty instead of looping
forever.

> | @@ -1860,12 +1884,6 @@ relock:
> | if (sig_kernel_ignore(signr)) /* Default is nothing. */
> | continue;
> |
> | - /*
> | -   * Global init gets no signals it doesn't want.
> | -   */
> | - if (is_global_init(current))
> | -   continue;

> | -
> | if (sig_kernel_stop(signr)) {
> | /*
> | * The default action is to stop all threads in
> | @@ -2246,8 +2264,10 @@ static int do_tkill(int tgid, int pid, int sig)
> | */
> | if (!error && sig && p->sighand) {
> | spin_lock_irq(&p->sighand->siglock);
> | -   handle_stop_signal(sig, p);
> | -   error = specific_send_sig_info(sig, &info, p);
> | +   if (sig_available(p, sig)) {
> | +   handle_stop_signal(sig, p);
> | +   error = specific_send_sig_info(sig, &info, p);
> | +   }
> | spin_unlock_irq(&p->sighand->siglock);
> | }
> | }
> | @@ -2336,7 +2356,8 @@ int do_sigaction(int sig, struct k_sigaction *act,
> struct k_sigaction *oact)
> | *   be discarded, whether or not it is blocked"
> | */
> | if (act->sa.sa_handler == SIG_IGN ||
> | -   (act->sa.sa_handler == SIG_DFL && sig_kernel_ignore(sig))) {
> | +   (act->sa.sa_handler == SIG_DFL && sig_kernel_ignore(sig)) ||
> | +   !sig_available(current, sig)) {
> | struct task_struct *t = current;
> | sigemptyset(&mask);
> | sigaddset(&mask, sig);
>
> Not sure about this. Consider that we extend the sig_available() as
> I mentioned above for container_init(). Then suppose following sequence
> occurs:
>
> - container-init receives a fatal signal say SIGUSR1, from parent-ns
>   i.e the signal is pending.
>
> - before processing the pending signal, the container-init sets
>   the handler to SIG_DFL (which is to terminate).
>
>   Will we then discard the pending SIGUSR1 even though it was from
>   from parent ns ?

Good question.

I guess if a signal from a child is already pending we have limited
responsibility for dropping it, because we have already allowed it
through...

My thought had been especially after I saw that part of Oleg's patch
that it was the right thing to do.

Given my attempt at well defined semantics for dropping the signal
from the sender. The rule would be if the signal makes it to the init
process we should not treat it specially.

The historical behavior would have been that if the signal was from a
child the signal would have been dropped just after this point when it
was delivered.

However if someone wants to prevent that case we can use sigwait, to
remove blocked signals. Further different rules for signal handling
for init I think are largely problematic for authors of different
inits because they are hard to remember.

Further sysvinit doesn't ever set a signal handler to SIG_DFL except
after it forks and just before it execs a process so the common case
will not be affected.

Therefore I think you are right. We don't need this case and it
is likely to be more problematic then useful.

Eric

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