Subject: Re: [PATCH 8/9] signal: Drop signals before sending them to init.
Posted by ebiederm on Tue, 18 Dec 2007 13:36:55 GMT
View Forum Message <> Reply to Message

Oleg Nesterov <oleg@tv-sign.ru> writes:

> On 12/17, Eric W. Biederman wrote:
>>
>> So I would have no problem with a definition said signals
>> will be dropped when sent to init if at the time they are
>> sent the signal is SIG_DFL and unblocked.
>
> Great!

My only issue is that with including the blocked signals in
the definition is that it makes things a little harder to
explain and understand.

>> > But this can happen with
>> > your patch as well. sig_init_drop() returns false if we have a handler,
>> > but this races with sys_rt_sigaction() which can set SIG_DFL, so init
>> > could be killed.
>>
>> I am checking under the sighand lock so we should not race,
>> at least not internally to the kernel.
>
> Yes, but as soon as we drop ->siglock /sbin/init can set SIG_DFL before
> noticing the signal.

I guess I don't see this as a race. The definition in my head is
all about permission to send a signal to init. That permission happens
if init shows interest in the incoming signal. So it is an
instant in time decision.

At another instant the permissions may have changed and we are
allowed to send the signal.

You seem to be implying that something is wrong. If something
is wrong if something can be wrong either we have the definition wrong
or the implementation wrong. If we can not implement things to be
correct with our new definition then it is the wrong definition.
So races are totally NOT ok.

I don't think you have made the switch to the permission check
mentality that I am proposing in my definition.

Can you restate from the perspective of allowing or denying
a signal how looking at the blocked state of the signal

is better then just looking at SIG_DFL?

>> > IOW, I still have a strong feeling that this patch
>> >
>> > http://marc.info/?l=linux-kernel&m=118753610515859
>> >
>> > is better, and more correct. That said, this all is very subjective,
>> > I can't "prove" this of course.
>>
>> My fundamental problem with that patch is that it drops signals
>> after we have started processing them, and it modifies the code
>> of an optimization.
>>
>> To have a clean definition and clean semantics I think we need
>> to drop the signal earlier in the path. Which is what I
>> really object to in your patch.
>
> Hmm. Could you look at this patch again? I'm attaching it at the end.
> (re-diffed against the current code)
>
> It modifies sig_ignored(), so we drop the signal before we started
> processing. And in fact it is more "optimized", because we don't need
> to check sa_handler twice.

Yes. It is more "optimized" but from what I can tell less correct.
It makes it really easy to get the definition wrong.   The big
problem is you allow all signals through in the case of ptrace.
Which is so totally wrong. For an optimization (which sig_ignored is)
that is fine. Since this new behavior is not an optimization we
just can't do that.

The definition needs to be something that guarantees the signal
is not sent to init (not ignored by init) if the conditions are
correct.

Semantically not sent is hugely different from ignored.

Doing it in sig_ignored is just a little to late in the signal sending
pathway, and I believe it will be a maintenance nightmare.

> Btw. I don't think we should change force_sig_info(). Suppose that init
> blocks/ignores SIGSEGV and do_page_fault() does force_sig_info_fault().
> In that case it is better to die explicitely than go into the endless
> loop.

Totally and my patch would be smaller if we did. However I think we
should change which set of signals are dropped in a separate patch,

which is why
definition of where signals are dropped and then
we should explicitly let signals through in a separate patch.

```
>
> Oleg.
>
> --- t/kernel/signal.c~INITSIGS 2007-08-19 14:39:35.000000000 +0400
> +++ t/kernel/signal.c 2007-08-19 19:00:27.000000000 +0400
> @@ -39,11 +39,33 @@
>
>  static struct kmem_cache *sigqueue_cachep;
>
> +static int sig_init_ignore(struct task_struct *tsk)
> +{
> + if (likely(!is_container_init(tsk->group_leader)))
> +  return 0;
> +
> + // ---------------- Multiple pid namespaces ----------------
> + // if (current is from tsk's parent pid_ns && !in_interrupt())
> + // return 0;
```

We need to test siginfo to see if the signal is a signal from
the kernel not in_interrupt(). So (before handling namespaces
this would be)

```
static int sig_init_ignrore(struct task_struct *tsk, siginfo_t *info,
       struct pid *sender)
{
        /* Grumble we should look at the TGID and not need to
         * pass in group_leader.
         */
 if (likely(!is_container_init(tsk->group_leader)))
         return 0;


 /* Ignore signals from the kernel */
 if ((!is_si_special(info) && SI_FROM_KERNEL(info)) ||
     (info != SEND_SIG_NOINFO))
  return 1;

 /* If the kernel didn't send the signal figure out who did */
   if (!sender)
  sender = task_tgid(current);

 /* Don't drop user mode signals from an outer pid
  * namespace.
  */
 if (!pid_in_ns(sender, task_active_pid_ns(task)))
  return 0;

 return 1;
}
```

```
> + return 1;
> +}
> +
> +static int sig_task_ignore(struct task_struct *tsk, int sig)
> +{
> + void __user * handler = tsk->sighand->action[sig-1].sa.sa_handler;
> +
> + if (handler == SIG_IGN)
> +  return 1;
> +
> + if (handler != SIG_DFL)
> +  return 0;
> +
> + return sig_kernel_ignore(sig) || sig_init_ignore(tsk);
> +}
>
>  static int sig_ignored(struct task_struct *t, int sig)
>  {
> - void __user * handler;
> -
>   /*
>    * Tracers always want to know about signals..
>    */
```

Since we are dropping the signal before it is sent, tracers
should never see the signal.

```
> @@ -58,10 +82,7 @@ static int sig_ignored(struct task_struc
>   if (sigismember(&t->blocked, sig) || sigismember(&t->real_blocked, sig))
>    return 0;
>
> - /* Is it explicitly or implicitly ignored? */
> - handler = t->sighand->action[sig-1].sa.sa_handler;
> - return   handler == SIG_IGN ||
> -  (handler == SIG_DFL && sig_kernel_ignore(sig));
> + return sig_task_ignore(t, sig);
>  }
>
>  /*
> @@ -566,6 +587,9 @@ static void handle_stop_signal(int sig,
>     */
>    return;
>
> + if (sig_init_ignore(p))
> +  return;
> +
>   if (sig_kernel_stop(sig)) {
>    /*
>     * This is a stop signal.  Remove SIGCONT from all queues.
> @@ -1786,12 +1810,6 @@ 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
> @@ -2303,8 +2316,7 @@ int do_sigaction(int sig, struct k_sigac
>     *   (for example, SIGCHLD), shall cause the pending signal to
>     *   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))) {
> +  if (sig_task_ignore(current, sig)) {
>     struct task_struct *t = current;
>     sigemptyset(&mask);
>     sigaddset(&mask, sig);
```

Any time you start ignoring the signal here you are not thinking
in terms of never sending the signal or not. Once a signal is sent
we must treat it like normal (to have a clean definition).

In the namespace case we can not look at a pending signal and decide
if we should drop it or not. So changing sigaction is impossible.

Eric