Subject: [PATCH v5] Fix rmmod/read/write races in /proc entries
Posted by Alexey Dobriyan on Sun, 11 Mar 2007 17:04:56 GMT
View Forum Message <> Reply to Message

Differences from version 4:
 Updated in-code comments. Largely rewritten changelog.
 Lockdep please. --akpm
 ->read_proc, ->write_proc aren't special, Extend protection to
 most methods for regular /proc files. Mentioned by viro.
Differences from version 3:
 Use completion instead of unlock/schedule/lock
 Move refcount waiting business after removing PDE from lists,
 so that *cough* possible concurrent remove_proc_entry() will
 work.

Fix following races:
==========
1. Write via ->write_proc sleeps in copy_from_user(). Module disappears
   meanwhile. Or, more generically, system call done on /proc file, method
   supplied by module is called, module dissapeares meanwhile.

```
   pde = create_proc_entry()
   if (!pde)
 return -ENOMEM;
   pde->write_proc = ...
    open
    write
    copy_from_user
   pde = create_proc_entry();
   if (!pde) {
 remove_proc_entry();
 return -ENOMEM;
 /* module unloaded */
   }
    *boom*
```

==========
2. bogo-revoke aka proc_kill_inodes()

```
  remove_proc_entry  vfs_read
  proc_kill_inodes  [check ->f_op validness]
    [check ->f_op->read validness]
    [verify_area, security permissions checks]
 ->f_op = NULL;
    if (file->f_op->read)
     /* ->f_op dereference, boom */
```

NOTE, NOTE, NOTE: file_operations are proxied for regular files only. Let's
see how this scheme behaves, then extend if needed for directories.

Directories creators in /proc only set ->owner for them, so proxying for
directories may be unneeded.

NOTE, NOTE, NOTE: methods being proxied are ->llseek, ->read, ->write,
->poll, ->unlocked_ioctl, ->ioctl, ->compat_ioctl, ->open, ->release.
If your in-tree module uses something else, yell on me. Full audit pending.

Signed-off-by: Alexey Dobriyan <adobriyan@sw.ru>

```
 fs/proc/generic.c       |   32 +++++
 fs/proc/inode.c         |  279 ++++++++++++++++++++++++++++++++++++++++++++++++-
 include/linux/proc_fs.h |   13 ++
 3 files changed, 321 insertions(+), 3 deletions(-)

--- a/fs/proc/generic.c
+++ b/fs/proc/generic.c
@@ -20,6 +20,7 @@ #include <linux/idr.h>
 #include <linux/namei.h>
 #include <linux/bitops.h>
 #include <linux/spinlock.h>
+#include <linux/completion.h>
 #include <asm/uaccess.h>

 #include "internal.h"
@@ -613,6 +614,9 @@ static struct proc_dir_entry *proc_creat
  ent->namelen = len;
  ent->mode = mode;
  ent->nlink = nlink;
+ ent->pde_users = 0;
+ spin_lock_init(&ent->pde_unload_lock);
+ ent->pde_unload_completion = NULL;
  out:
  return ent;
 }
@@ -734,9 +738,35 @@ void remove_proc_entry(const char *name,
   de = *p;
   *p = de->next;
   de->next = NULL;
+
+  spin_lock(&de->pde_unload_lock);
+  /*
+   * Stop accepting new callers into module. If you're
+   * dynamically allocating ->proc_fops, save a pointer somewhere.
+   */
+  de->proc_fops = NULL;
+  /* Wait until all existing callers into module are done. */
+  if (de->pde_users > 0) {
```

```
+   DECLARE_COMPLETION_ONSTACK(c);
+
+   if (!de->pde_unload_completion)
+    de->pde_unload_completion = &c;
+
+   spin_unlock(&de->pde_unload_lock);
+   spin_unlock(&proc_subdir_lock);
+
+   wait_for_completion(de->pde_unload_completion);
+
+   spin_lock(&proc_subdir_lock);
+   goto continue_removing;
+  }
+  spin_unlock(&de->pde_unload_lock);
+
+continue_removing:
   if (S_ISDIR(de->mode))
    parent->nlink--;
-  proc_kill_inodes(de);
+  if (!S_ISREG(de->mode))
+   proc_kill_inodes(de);
   de->nlink = 0;
   WARN_ON(de->subdir);
   if (!atomic_read(&de->count))
--- a/fs/proc/inode.c
+++ b/fs/proc/inode.c
@@ -142,6 +142,277 @@ static const struct super_operations pro
  .remount_fs = proc_remount,
 };

+static loff_t proc_reg_llseek(struct file *file, loff_t offset, int whence)
+{
+ struct proc_dir_entry *pde = PDE(file->f_path.dentry->d_inode);
+ loff_t rv = -EINVAL;
+ loff_t (*llseek)(struct file *, loff_t, int);
+
+ spin_lock(&pde->pde_unload_lock);
+ /*
+  * remove_proc_entry() is going to delete PDE (as part of module
+  * cleanup sequence). No new callers into module allowed.
+  */
+ if (!pde->proc_fops)
+  goto out_unlock;
+ /*
+  * Bump refcount so that remove_proc_entry will wail for ->llseek to
+  * complete.
+  */
+ pde->pde_users++;
```

```
+ /*
+  * Save function pointer under lock, to protect against ->proc_fops
+  * NULL'ifying right after ->pde_unload_lock is dropped.
+  */
+ llseek = pde->proc_fops->llseek;
+ spin_unlock(&pde->pde_unload_lock);
+
+ if (llseek)
+  rv = llseek(file, offset, whence);
+
+ spin_lock(&pde->pde_unload_lock);
+ pde->pde_users--;
+ if (pde->pde_unload_completion && pde->pde_users == 0)
+  complete(pde->pde_unload_completion);
+out_unlock:
+ spin_unlock(&pde->pde_unload_lock);
+
+ return rv;
+}
+
+static ssize_t proc_reg_read(struct file *file, char __user *buf, size_t count, loff_t *ppos)
+{
+ struct proc_dir_entry *pde = PDE(file->f_path.dentry->d_inode);
+ ssize_t rv = -EIO;
+ ssize_t (*read)(struct file *, char __user *, size_t, loff_t *);
+
+ spin_lock(&pde->pde_unload_lock);
+ if (!pde->proc_fops)
+  goto out_unlock;
+ pde->pde_users++;
+ read = pde->proc_fops->read;
+ spin_unlock(&pde->pde_unload_lock);
+
+ if (read)
+  rv = read(file, buf, count, ppos);
+
+ spin_lock(&pde->pde_unload_lock);
+ pde->pde_users--;
+ if (pde->pde_unload_completion && pde->pde_users == 0)
+  complete(pde->pde_unload_completion);
+out_unlock:
+ spin_unlock(&pde->pde_unload_lock);
+
+ return rv;
+}
+
+static ssize_t proc_reg_write(struct file *file, const char __user *buf, size_t count, loff_t *ppos)
+{
```

```
+ struct proc_dir_entry *pde = PDE(file->f_path.dentry->d_inode);
+ ssize_t rv = -EIO;
+ ssize_t (*write)(struct file *, const char __user *, size_t, loff_t *);
+
+ spin_lock(&pde->pde_unload_lock);
+ if (!pde->proc_fops)
+  goto out_unlock;
+ pde->pde_users++;
+ write = pde->proc_fops->write;
+ spin_unlock(&pde->pde_unload_lock);
+
+ if (write)
+  rv = write(file, buf, count, ppos);
+
+ spin_lock(&pde->pde_unload_lock);
+ pde->pde_users--;
+ if (pde->pde_unload_completion && pde->pde_users == 0)
+  complete(pde->pde_unload_completion);
+out_unlock:
+ spin_unlock(&pde->pde_unload_lock);
+
+ return rv;
+}
+
+static unsigned int proc_reg_poll(struct file *file, struct poll_table_struct *pts)
+{
+ struct proc_dir_entry *pde = PDE(file->f_path.dentry->d_inode);
+ unsigned int rv = 0;
+ unsigned int (*poll)(struct file *, struct poll_table_struct *);
+
+ spin_lock(&pde->pde_unload_lock);
+ if (!pde->proc_fops)
+  goto out_unlock;
+ pde->pde_users++;
+ poll = pde->proc_fops->poll;
+ spin_unlock(&pde->pde_unload_lock);
+
+ if (poll)
+  rv = poll(file, pts);
+
+ spin_lock(&pde->pde_unload_lock);
+ pde->pde_users--;
+ if (pde->pde_unload_completion && pde->pde_users == 0)
+  complete(pde->pde_unload_completion);
+out_unlock:
+ spin_unlock(&pde->pde_unload_lock);
+
+ return rv;
```

```
+}
+
+static long proc_reg_unlocked_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
+{
+ struct proc_dir_entry *pde = PDE(file->f_path.dentry->d_inode);
+ long rv = -ENOTTY;
+ long (*unlocked_ioctl)(struct file *, unsigned int, unsigned long);
+ int (*ioctl)(struct inode *, struct file *, unsigned int, unsigned long);
+
+ spin_lock(&pde->pde_unload_lock);
+ if (!pde->proc_fops)
+  goto out_unlock;
+ pde->pde_users++;
+ unlocked_ioctl = pde->proc_fops->unlocked_ioctl;
+ ioctl = pde->proc_fops->ioctl;
+ spin_unlock(&pde->pde_unload_lock);
+
+ if (unlocked_ioctl) {
+  rv = unlocked_ioctl(file, cmd, arg);
+  if (rv == -ENOIOCTLCMD)
+   rv = -EINVAL;
+ } else if (ioctl) {
+  lock_kernel();
+  rv = ioctl(file->f_path.dentry->d_inode, file, cmd, arg);
+  unlock_kernel();
+ }
+
+ spin_lock(&pde->pde_unload_lock);
+ pde->pde_users--;
+ if (pde->pde_unload_completion && pde->pde_users == 0)
+  complete(pde->pde_unload_completion);
+out_unlock:
+ spin_unlock(&pde->pde_unload_lock);
+
+ return rv;
+}
+
+#ifdef CONFIG_COMPAT
+static long proc_reg_compat_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
+{
+ struct proc_dir_entry *pde = PDE(file->f_path.dentry->d_inode);
+ long rv = -ENOTTY;
+ long (*compat_ioctl)(struct file *, unsigned int, unsigned long);
+
+ spin_lock(&pde->pde_unload_lock);
+ if (!pde->proc_fops)
+  goto out_unlock;
+ pde->pde_users++;
```

```
+ compat_ioctl = pde->proc_fops->compat_ioctl;
+ spin_unlock(&pde->pde_unload_lock);
+
+ if (compat_ioctl)
+  rv = compat_ioctl(file, cmd, arg);
+
+ spin_lock(&pde->pde_unload_lock);
+ pde->pde_users--;
+ if (pde->pde_unload_completion && pde->pde_users == 0)
+  complete(pde->pde_unload_completion);
+out_unlock:
+ spin_unlock(&pde->pde_unload_lock);
+
+ return rv;
+}
+#endif
+
+static int proc_reg_mmap(struct file *file, struct vm_area_struct *vma)
+{
+ struct proc_dir_entry *pde = PDE(file->f_path.dentry->d_inode);
+ int rv = -EIO;
+ int (*mmap)(struct file *, struct vm_area_struct *);
+
+ spin_lock(&pde->pde_unload_lock);
+ if (!pde->proc_fops)
+  goto out_unlock;
+ pde->pde_users++;
+ mmap = pde->proc_fops->mmap;
+ spin_unlock(&pde->pde_unload_lock);
+
+ if (mmap)
+  rv = mmap(file, vma);
+
+ spin_lock(&pde->pde_unload_lock);
+ pde->pde_users--;
+ if (pde->pde_unload_completion && pde->pde_users == 0)
+  complete(pde->pde_unload_completion);
+out_unlock:
+ spin_unlock(&pde->pde_unload_lock);
+
+ return rv;
+}
+
+static int proc_reg_open(struct inode *inode, struct file *file)
+{
+ struct proc_dir_entry *pde = PDE(inode);
+ int rv = 0;
+ int (*open)(struct inode *, struct file *);
```

```
+
+ spin_lock(&pde->pde_unload_lock);
+ if (!pde->proc_fops)
+  goto out_unlock;
+ pde->pde_users++;
+ open = pde->proc_fops->open;
+ spin_unlock(&pde->pde_unload_lock);
+
+ if (open)
+  rv = open(inode, file);
+
+ spin_lock(&pde->pde_unload_lock);
+ pde->pde_users--;
+ if (pde->pde_unload_completion && pde->pde_users == 0)
+  complete(pde->pde_unload_completion);
+out_unlock:
+ spin_unlock(&pde->pde_unload_lock);
+
+ return rv;
+}
+
+static int proc_reg_release(struct inode *inode, struct file *file)
+{
+ struct proc_dir_entry *pde = PDE(inode);
+ int rv = 0;
+ int (*release)(struct inode *, struct file *);
+
+ spin_lock(&pde->pde_unload_lock);
+ if (!pde->proc_fops)
+  goto out_unlock;
+ pde->pde_users++;
+ release = pde->proc_fops->release;
+ spin_unlock(&pde->pde_unload_lock);
+
+ if (release)
+  rv = release(inode, file);
+
+ spin_lock(&pde->pde_unload_lock);
+ pde->pde_users--;
+ if (pde->pde_unload_completion && pde->pde_users == 0)
+  complete(pde->pde_unload_completion);
+out_unlock:
+ spin_unlock(&pde->pde_unload_lock);
+
+ return rv;
+}
+
+static const struct file_operations proc_reg_file_ops = {
```

```
+ .llseek  = proc_reg_llseek,
+ .read  = proc_reg_read,
+ .write  = proc_reg_write,
+ .poll  = proc_reg_poll,
+ .unlocked_ioctl = proc_reg_unlocked_ioctl,
+#ifdef CONFIG_COMPAT
+ .compat_ioctl = proc_reg_compat_ioctl,
+#endif
+ .mmap  = proc_reg_mmap,
+ .open  = proc_reg_open,
+ .release = proc_reg_release,
+};
+
 struct inode *proc_get_inode(struct super_block *sb, unsigned int ino,
     struct proc_dir_entry *de)
 {
@@ -170,8 +441,12 @@ struct inode *proc_get_inode(struct supe
    inode->i_nlink = de->nlink;
   if (de->proc_iops)
    inode->i_op = de->proc_iops;
-  if (de->proc_fops)
-   inode->i_fop = de->proc_fops;
+  if (de->proc_fops) {
+   if (S_ISREG(inode->i_mode))
+    inode->i_fop = &proc_reg_file_ops;
+   else
+    inode->i_fop = de->proc_fops;
+  }
  }

  return inode;
--- a/include/linux/proc_fs.h
+++ b/include/linux/proc_fs.h
@@ -7,6 +7,8 @@ #include <linux/spinlock.h>
 #include <linux/magic.h>
 #include <asm/atomic.h>

+struct completion;
+
 /*
  * The proc filesystem constants/structures
  */
@@ -56,6 +58,14 @ struct proc_dir_entry {
  gid_t gid;
  loff_t size;
  const struct inode_operations *proc_iops;
+ /*
+  * NULL ->proc_fops means "PDE is going away RSN" or
```

```
+  * "PDE is just created". In either case, e.g. ->read_proc won't be
+  * called because it's too late or too early, respectively.
+  *
+  * If you're allocating ->proc_fops dynamically, save a pointer
+  * somewhere.
+  */
  const struct file_operations *proc_fops;
  get_info_t *get_info;
  struct module *owner;
@@ -66,6 +76,9 @@ struct proc_dir_entry {
  atomic_t count;  /* use count */
  int deleted;  /* delete flag */
  void *set;
+ int pde_users; /* number of callers into module in progress */
+ spinlock_t pde_unload_lock; /* proc_fops checks and pde_users bumps */
+ struct completion *pde_unload_completion;
 };

 struct kcore_list {
```

Subject: Re: [PATCH v5] Fix rmmod/read/write races in /proc entries
Posted by Andrew Morton on Fri, 16 Mar 2007 01:53:04 GMT
View Forum Message <> Reply to Message

On Sun, 11 Mar 2007 20:04:56 +0300 Alexey Dobriyan <adobriyan@sw.ru> wrote:

> Differences from version 4:
> Updated in-code comments. Largely rewritten changelog.
> Lockdep please. --akpm
> ->read_proc, ->write_proc aren't special, Extend protection to
> most methods for regular /proc files. Mentioned by viro.
> Differences from version 3:
> Use completion instead of unlock/schedule/lock
> Move refcount waiting business after removing PDE from lists,
> so that *cough* possible concurrent remove_proc_entry() will
> work.

My, what a lot of code you have here. I note that nobody can be assed even
reviewing it. Now why is that?

> Fix following races:
> ==
> 1. Write via ->write_proc sleeps in copy_from_user(). Module disappears
>   meanwhile. Or, more generically, system call done on /proc file, method
>   supplied by module is called, module dissapeares meanwhile.
>
>   pde = create_proc_entry()

>   if (!pde)
> return -ENOMEM;
>   pde->write_proc = ...
>   open
>   write
>   copy_from_user
>   pde = create_proc_entry();
>   if (!pde) {
> remove_proc_entry();
> return -ENOMEM;
> /* module unloaded */
>   }

We usually fix that race by pinning the module: make whoever registered the
proc entries also register their THIS_MODULE, do a try_module_get() on it
before we start to play with data structures which the module owns.

Can we do that here?

And is the above race fix related to the below one in any fashion?

> ==
> 2. bogo-revoke aka proc_kill_inodes()
>
>   remove_proc_entry vfs_read
>   proc_kill_inodes [check ->f_op validness]
>   [check ->f_op->read validness]
>   [verify_area, security permissions checks]
> ->f_op = NULL;
>   if (file->f_op->read)
>   /* ->f_op dereference, boom */

So you fixed this via sort-of-refcounting on pde->pde_users.

hmm.

> NOTE, NOTE, NOTE: file_operations are proxied for regular files only. Let's
> see how this scheme behaves, then extend if needed for directories.
> Directories creators in /proc only set ->owner for them, so proxying for
> directories may be unneeded.
>
> NOTE, NOTE, NOTE: methods being proxied are ->llseek, ->read, ->write,
> ->poll, ->unlocked_ioctl, ->ioctl, ->compat_ioctl, ->open, ->release.
> If your in-tree module uses something else, yell on me. Full audit pending.
>
> Signed-off-by: Alexey Dobriyan <adobriyan@sw.ru>
> ---
>

```
>  fs/proc/generic.c       |   32 +++++
>  fs/proc/inode.c         |  279 ++++++++++++++++++++++++++++++++++++++++++++++++-
>  include/linux/proc_fs.h |   13 ++
>  3 files changed, 321 insertions(+), 3 deletions(-)
>
> --- a/fs/proc/generic.c
> +++ b/fs/proc/generic.c
> @@ -20,6 +20,7 @@ #include <linux/idr.h>
>  #include <linux/namei.h>
>  #include <linux/bitops.h>
>  #include <linux/spinlock.h>
> +#include <linux/completion.h>
>  #include <asm/uaccess.h>
>
>  #include "internal.h"
> @@ -613,6 +614,9 @@ static struct proc_dir_entry *proc_creat
>   ent->namelen = len;
>   ent->mode = mode;
>   ent->nlink = nlink;
> + ent->pde_users = 0;
> + spin_lock_init(&ent->pde_unload_lock);
> + ent->pde_unload_completion = NULL;
>   out:
>   return ent;
>  }
> @@ -734,9 +738,35 @@ void remove_proc_entry(const char *name,
>    de = *p;
>    *p = de->next;
>    de->next = NULL;
> +
> +  spin_lock(&de->pde_unload_lock);
> +  /*
> +   * Stop accepting new callers into module. If you're
> +   * dynamically allocating ->proc_fops, save a pointer somewhere.
> +   */
> +  de->proc_fops = NULL;
> +  /* Wait until all existing callers into module are done. */
> +  if (de->pde_users > 0) {
> +   DECLARE_COMPLETION_ONSTACK(c);
> +
> +   if (!de->pde_unload_completion)
> +    de->pde_unload_completion = &c;
> +
> +   spin_unlock(&de->pde_unload_lock);
> +   spin_unlock(&proc_subdir_lock);
> +
> +   wait_for_completion(de->pde_unload_completion);
> +
```

```
> +   spin_lock(&proc_subdir_lock);
> +   goto continue_removing;
> +  }
> +  spin_unlock(&de->pde_unload_lock);
> +
> +continue_removing:
>    if (S_ISDIR(de->mode))
>     parent->nlink--;
> -  proc_kill_inodes(de);
> +  if (!S_ISREG(de->mode))
> +   proc_kill_inodes(de);
>    de->nlink = 0;
>    WARN_ON(de->subdir);
>    if (!atomic_read(&de->count))
> --- a/fs/proc/inode.c
> +++ b/fs/proc/inode.c
> @@ -142,6 +142,277 @@ static const struct super_operations pro
>   .remount_fs = proc_remount,
>  };
>
> +static loff_t proc_reg_llseek(struct file *file, loff_t offset, int whence)
> +{
> + struct proc_dir_entry *pde = PDE(file->f_path.dentry->d_inode);
> + loff_t rv = -EINVAL;
> + loff_t (*llseek)(struct file *, loff_t, int);
> +
> + spin_lock(&pde->pde_unload_lock);
> + /*
> +  * remove_proc_entry() is going to delete PDE (as part of module
> +  * cleanup sequence). No new callers into module allowed.
> +  */
> + if (!pde->proc_fops)
> +  goto out_unlock;
> + /*
> +  * Bump refcount so that remove_proc_entry will wail for ->llseek to
> +  * complete.
> +  */
> + pde->pde_users++;
> + /*
> +  * Save function pointer under lock, to protect against ->proc_fops
> +  * NULL'ifying right after ->pde_unload_lock is dropped.
> +  */
> + llseek = pde->proc_fops->llseek;
> + spin_unlock(&pde->pde_unload_lock);
> +
> + if (llseek)
> +  rv = llseek(file, offset, whence);
> +
```

> + spin_lock(&pde->pde_unload_lock);
> + pde->pde_users--;
> + if (pde->pde_unload_completion && pde->pde_users == 0)
> + complete(pde->pde_unload_completion);
> +out_unlock:
> + spin_unlock(&pde->pde_unload_lock);

The above six lines happen rather a lot - perhaps it could be placed in a
helper funtion?

Subject: Re: [PATCH v5] Fix rmmod/read/write races in /proc entries
Posted by Alexey Dobriyan on Fri, 16 Mar 2007 09:09:42 GMT
View Forum Message <> Reply to Message

On Thu, Mar 15, 2007 at 05:53:04PM -0800, Andrew Morton wrote:
> My, what a lot of code you have here. I note that nobody can be assed even
> reviewing it. Now why is that?

I hope, Al could find some time again.

> On Sun, 11 Mar 2007 20:04:56 +0300 Alexey Dobriyan <adobriyan@sw.ru> wrote:
> > Fix following races:
> > ==========
> > 1. Write via ->write_proc sleeps in copy_from_user(). Module disappears
> >   meanwhile. Or, more generically, system call done on /proc file, method
> >   supplied by module is called, module dissapeares meanwhile.
> >
> >   pde = create_proc_entry()
> >   if (!pde)
> > return -ENOMEM;
> >   pde->write_proc = ...
> >   open
> >   write
> >   copy_from_user
> >   pde = create_proc_entry();
> >   if (!pde) {
> > remove_proc_entry();
> > return -ENOMEM;
> > /* module unloaded */
> >   }
>
> We usually fix that race by pinning the module: make whoever registered the
> proc entries also register their THIS_MODULE, do a try_module_get() on it
> before we start to play with data structures which the module owns.
>
> Can we do that here?

We can, but it will be unreliable:

Typical proc entry creation sequence is

```
 pde = create_proc_entry(...);
 if (pde)
  pde->owner = THIS_MODULE;
```

Right after create_proc_entry() ->owner is NULL, so try_module_get()
won't do anything, but proc_delete_inode() could put module which was
never getted.

This should fixable by always setting ->owner before proc entry is
glued to proc entries tree. Something like this:

```
 #define create_proc_entry(...) __create_proc_entry(..., THIS_MODULE)
```

However, I think it's not enough: delete_module(2) first waits for
refcount becoming zero, only then calls modules's exit function which
starts removing proc entries. In between, proc entries are accessible
and fully-functional, so try_module_get() can again get module and
module_put(pde->owner) can happen AFTER module dissapears.
What will it put?

And how can you fix that? The only way I know is to REMOVE ->owner
completely, once we agree on this pde_users/pde_unload_lock stuff.

> And is the above race fix related to the below one in any fashion?
> > ==
> > 2. bogo-revoke aka proc_kill_inodes()
> >
> >   remove_proc_entry vfs_read
> >   proc_kill_inodes [check ->f_op validness]
> >   [check ->f_op->read validness]
> >   [verify_area, security permissions checks]
> > ->f_op = NULL;
> >   if (file->f_op->read)
> >   /* ->f_op dereference, boom */
>
> So you fixed this via sort-of-refcounting on pde->pde_users.
>
> hmm.

Probably, you're right and they are independently fixable. It's all
about following 3 lines after all. My turn to hmm...

> > - proc_kill_inodes(de);
> > + if (!S_ISREG(de->mode))

> > +   proc_kill_inodes(de);

> > + spin_lock(&pde->pde_unload_lock);
> > + pde->pde_users--;
> > + if (pde->pde_unload_completion && pde->pde_users == 0)
> > + complete(pde->pde_unload_completion);
> > +out_unlock:
> > + spin_unlock(&pde->pde_unload_lock);
>
> The above six lines happen rather a lot - perhaps it could be placed in a
> helper funtion?

OK. Should I send incremental updates or full patch again?

Subject: Re: [PATCH v5] Fix rmmod/read/write races in /proc entries
Posted by Andrew Morton on Fri, 16 Mar 2007 11:50:30 GMT
View Forum Message <> Reply to Message

On Fri, 16 Mar 2007 12:16:13 +0300 Alexey Dobriyan <adobriyan@sw.ru> wrote:

> On Thu, Mar 15, 2007 at 05:53:04PM -0800, Andrew Morton wrote:
> > My, what a lot of code you have here. I note that nobody can be assed even
> > reviewing it. Now why is that?
>
> I hope, Al could find some time again.
>
> > On Sun, 11 Mar 2007 20:04:56 +0300 Alexey Dobriyan <adobriyan@sw.ru> wrote:
> > > Fix following races:
> > > ==
> > > 1. Write via ->write_proc sleeps in copy_from_user(). Module disappears
> > >   meanwhile. Or, more generically, system call done on /proc file, method
> > >   supplied by module is called, module dissapeares meanwhile.
> > >
> > >   pde = create_proc_entry()
> > >   if (!pde)
> > > return -ENOMEM;
> > >   pde->write_proc = ...
> > >   open
> > >   write
> > >   copy_from_user
> > >   pde = create_proc_entry();
> > >   if (!pde) {
> > > remove_proc_entry();
> > > return -ENOMEM;
> > > /* module unloaded */

> > > }
> >
> > We usually fix that race by pinning the module: make whoever registered the
> > proc entries also register their THIS_MODULE, do a try_module_get() on it
> > before we start to play with data structures which the module owns.
> >
> > Can we do that here?
>
> We can, but it will be unreliable:
>
> Typical proc entry creation sequence is
>
> pde = create_proc_entry(...);
> if (pde)
>   pde->owner = THIS_MODULE;
>
> Right after create_proc_entry() ->owner is NULL, so try_module_get()
> won't do anything, but proc_delete_inode() could put module which was
> never getted.
>
> This should fixable by always setting ->owner before proc entry is
> glued to proc entries tree. Something like this:
>
> #define create_proc_entry(...) __create_proc_entry(..., THIS_MODULE)

Yes, I was thinking of something like that.

> However, I think it's not enough: delete_module(2) first waits for
> refcount becoming zero, only then calls modules's exit function which
> starts removing proc entries. In between, proc entries are accessible
> and fully-functional, so try_module_get() can again get module and
> module_put(pde->owner) can happen AFTER module dissapears.
> What will it put?
>
> And how can you fix that? The only way I know is to REMOVE ->owner
> completely, once we agree on this pde_users/pde_unload_lock stuff.

I think the rmmod code will take care of that.

Once delete_module() has called try_stop_module(), no following
try_module_get() will succeed. And see that wait_for_zero_refcount() call
in there which waits for any present users of the module to go away.

Subject: Re: [PATCH v5] Fix rmmod/read/write races in /proc entries
Posted by Alexey Dobriyan on Mon, 19 Mar 2007 14:56:15 GMT
View Forum Message <> Reply to Message

On Fri, Mar 16, 2007 at 03:50:30AM -0800, Andrew Morton wrote:
> On Fri, 16 Mar 2007 12:16:13 +0300 Alexey Dobriyan <adobriyan@sw.ru> wrote:
> > On Thu, Mar 15, 2007 at 05:53:04PM -0800, Andrew Morton wrote:
> > > My, what a lot of code you have here. I note that nobody can be assed even
> > > reviewing it. Now why is that?
> >
> > I hope, Al could find some time again.
> >
> > > On Sun, 11 Mar 2007 20:04:56 +0300 Alexey Dobriyan <adobriyan@sw.ru> wrote:
> > > > Fix following races:
> > > > ==
> > > > 1. Write via ->write_proc sleeps in copy_from_user(). Module disappears
> > > >   meanwhile. Or, more generically, system call done on /proc file, method
> > > >   supplied by module is called, module dissapeares meanwhile.
> > > >
> > > >   pde = create_proc_entry()
> > > >   if (!pde)
> > > > return -ENOMEM;
> > > >   pde->write_proc = ...
> > > >   open
> > > >   write
> > > >   copy_from_user
> > > >   pde = create_proc_entry();
> > > >   if (!pde) {
> > > > remove_proc_entry();
> > > > return -ENOMEM;
> > > > /* module unloaded */
> > > >   }
> > >
> > > We usually fix that race by pinning the module: make whoever registered the
> > > proc entries also register their THIS_MODULE, do a try_module_get() on it
> > > before we start to play with data structures which the module owns.
> > >
> > > Can we do that here?
> >
> > We can, but it will be unreliable:
> >
> > Typical proc entry creation sequence is
> >
> > pde = create_proc_entry(...);
> > if (pde)
> >   pde->owner = THIS_MODULE;
> >
> > Right after create_proc_entry() ->owner is NULL, so try_module_get()
> > won't do anything, but proc_delete_inode() could put module which was
> > never getted.
> >
> > This should fixable by always setting ->owner before proc entry is

> > glued to proc entries tree. Something like this:
> >
> > #define create_proc_entry(...) __create_proc_entry(..., THIS_MODULE)
>
> Yes, I was thinking of something like that.
>
> > However, I think it's not enough: delete_module(2) first waits for
> > refcount becoming zero, only then calls modules's exit function which
> > starts removing proc entries. In between, proc entries are accessible
> > and fully-functional, so try_module_get() can again get module and
> > module_put(pde->owner) can happen AFTER module dissapears.
> > What will it put?
> >
> > And how can you fix that? The only way I know is to REMOVE ->owner
> > completely, once we agree on this pde_users/pde_unload_lock stuff.
>
> I think the rmmod code will take care of that.
>
> Once delete_module() has called try_stop_module(), no following
> try_module_get() will succeed. And see that wait_for_zero_refcount() call
> in there which waits for any present users of the module to go away.

See it. So to make all this reliable we need to
a) #define create_proc_entry(...) __create_proc_entry(..., THIS_MODULE)
   and change all users which want to use underscored version.
   Repeat for friends.
b) drag de_get() under proc_subdir_lock
c) drag try_module_get() under proc_subdir_lock

Now there is a problem with ->owner flippers which change ->owner on the
fly, which means try_module_get() and module_put() could be done on
different modules, ick.

One such [easily fixable] user is snd_info_register() which does

```
 p = snd_create_proc_entry(entry->name, entry->mode, root);
 if (!p) {
  mutex_unlock(&info_mutex);
  return -ENOMEM;
 }
 p->owner = entry->module;
```

but snd_create_proc_entry() created proc entry with THIS_MODULE of
sound/core/info.c.

Very little we can do about this except periodically checking every proc
entry. With all those pde_users/pde_unload_lock/pde_unload_completion
patches I've posted ->owner can go away nicely fixing ->owner flippers

problem too. There would be simply nothing to flip.