Subject: Re: checkpointing and restoring processes
Posted by Cedric Le Goater on Wed, 06 Jun 2007 12:48:07 GMT
View Forum Message <> Reply to Message

Mark Pflueger wrote:
> hi everyone!
>
> i'm not subscribed to the list, so if you care to flame because of my noob
> question, just do it to the list, otherwise please cc me.

you should subscribe to containers@lists.osdl.org and send your ideas on that list. There's a BOF on that topic at OLS if you can attend.

cheers,

C.

> i'm trying to write a checkpoint/restore module for processes and so have
> a basic version going already - problem is, when i restore the process,
> one of three things happens at random. first is, the process restored
> segfaults. second is, i get a kernel null pointer dereference and third
> is, i get a virtual address lookup error and a kernel crash. the trace
> back and the address always change.
>
> the user space process is as simple as i could make it: (error checking
> and debugging messages are left out)
>
>
> void take_chkpt(void) {
> pid_t pid;
> char call_pid[10];
> char call_num[10];
>
> chkptpid = getpid();
> snprintf(call_pid, 9, "%d", chkptpid);
> snprintf(call_num, 9, "%d", checkpointnum);
>
> switch(pid = fork()) {
> case -1:
> fprintf(stderr, "Fork failed.\n");
> return;
> break;
> case 0: /* child process */
> if(!execl("child_take", call_pid, call_num, (char *)0))
> perror("execl: ");
> break;
> default: /* parent process */
> waitpid(pid, NULL, 0);

```
>                 break;
>  }
>
>         return;
> }
>
>
> void restore_chkpts(void) {
>         pid_t pid;
>         char call_pid[10];
>         char call_num[10];
>
>  ENTERFUN();
>
>         if(restore_retry) // do nothing on second call to restore
>                 return;
>
>         chkptpid = getpid();
>         snprintf(call_pid, 9, "%d", chkptpid);
>         snprintf(call_num, 9, "%d", checkpointnum);
>
>  switch(pid = fork()) {
>  case -1:
>                 fprintf(stderr, "MP: Fork failed.\n");
>                 return;
>                 break;
>  case  0:   /* child process */
>                 if(!execl("child_restore", call_pid, call_num, (char *)0))
>                         perror("execl: ");
>                 break;
>  default:   /* parent process */
>                 INF(("Parent Process"));
>                 restore_retry=1;
>                 INF(("Wait for Child..."));
>                 waitpid(pid, NULL, 0);
>                 break;
>  }
>
>  LEAVEFUN();
>
>         return;
> }
>
> int main(int argc, char* argv[]) {
>  take_chkpt();
>  printf("Hello cruel world!\n");
>  restore_chkpts();
>  return 0;
```

```
> }
>
> where child_take and child_restore do the following:
>
>
> void child_take_chkpt(int chkptpid, int checkpointnum) {
>         struct chkpt_ioctl chkptio;
>         int dev_fd; // ioctl device file
>         char chkptname[30];
>
>         if ((dev_fd = open(CHKPT_DEVICE, O_RDWR)) < 0) {
>                 perror("MP: Open device file");
>                 exit(EXIT_FAILURE);
>         }
>         chkptio.pid = chkptpid;
>         snprintf(chkptname, 29, "/tmp/chkpt_%d_%d", chkptio.pid, checkpointnum);
>         chkptio.file = creat(chkptname, 00755);
>         sleep(1); // to go sure the parent process is in waitpid -- ugly,
> but works
>         kill(chkptio.pid, SIGSTOP);
>         sleep(1);
>         ioctl(dev_fd, CHKPT_IOCTL_SAVE, (unsigned long)&chkptio);
>         close(dev_fd);
>         close(chkptio.file);
>         kill(chkptio.pid, SIGCONT);
>         exit(0);
> }
>
> void child_restore_chkpts(int chkptpid, int checkpointnum) {
>         struct chkpt_ioctl chkptio;
>         int dev_fd; // ioctl device file
>         char chkptname[30];
>
>         snprintf(chkptname, 29, "/tmp/chkpt_%d_%d", chkptpid, checkpointnum-1);
>         chkptio.file = open(chkptname, O_RDONLY);
>         chkptio.pid = chkptpid;
>         dev_fd = open(CHKPT_DEVICE, O_RDWR);
>         sleep(1);
>         kill(chkptpid, SIGSTOP);
>         sleep(1);
>         ioctl(dev_fd, CHKPT_IOCTL_RESTORE, (unsigned long)&chkptio);
>         close(chkptio.file);
>         close(dev_fd);
>         kill(chkptpid, SIGCONT);
>         exit(0);
> }
>
> the header for the files is this:
```

```
>
>
> enum {
>         CHKPT_IOCTL_SAVE,
>         CHKPT_IOCTL_RESTORE
> };
>
> struct chkpt_ioctl {
>         pid_t pid; // for fork tests
>         int file;
> };
>
> struct chkpt {
>         pid_t pid; // for fork tests
>         struct pt_regs regs;
>         unsigned int datasize;
>         unsigned int brksize;
>         unsigned int stacksize;
> };
>
>
> and finally the kernel module:
>
> int chkpt_ioctl_handler(struct inode *i, struct file *f,
>                         unsigned int cmd, unsigned long arg) {
>         struct chkpt_ioctl pmio, *u_pmio;
>         int ret = -1;
>
>         u_pmio = (struct chkpt_ioctl *)arg;
>
>         switch(cmd) {
>         case CHKPT_IOCTL_SAVE:
>                 if (copy_from_user(&pmio, u_pmio, sizeof(struct
> chkpt_ioctl))) {
>                         printk("...failed to copy from user\n");
>                         ret = -1;
>                         break;
>                 }
>                 if(chkpt_save(&pmio) < 0) {
>                         printk("...failed to save chkpt\n");
>                         ret = -1;
>                         break;
>                 }
>                 ret = 0;
>                 break;
>         case CHKPT_IOCTL_RESTORE:
>                 INFO(("CHKPT_IOCTL_RESTORE"));
>                 if (copy_from_user(&pmio, u_pmio, sizeof(struct
```

```
> chkpt_ioctl))) {
>                     printk("...failed to copy from user\n");
>                     ret = -1;
>                     break;
>             }
>             if (chkpt_restore(&pmio) < 0) {
>                     printk("...failed to restore chkpt\n");
>                     ret = -1;
>                     break;
>             }
>             ret = 0;
>             break;
>     default:
>             printk("...illegal ioctl cmd\n");
>             ret = -1;
>             break;
>     }
>     return ret;
> }
>
> static int chkpt_save(struct chkpt_ioctl *chkptio) {
>     struct task_struct *tsk;
>     struct chkpt chkpt;
>     unsigned int datasz, brksz, stacksz;
>     struct file *f;
>
>     if (!(tsk = find_task_by_pid(chkptio->pid))) {
>             printk("...task %d not found\n", chkptio->pid);
>             return -1;
>     }
>
>     f = current->files->fd[chkptio->file];
>
>     datasz = tsk->mm->end_data - tsk->mm->start_data; // data
>     brksz = tsk->mm->brk - tsk->mm->start_brk; // brk
>     stacksz = tsk->thread.esp0 - tsk->thread.esp; // stack
>
>     /* saving most important information belonging to tsk */
>     /* NO FILES, SOCKETS, PIPES, SHARED MEMORY AND SEMAPHORES */
>     chkpt.pid = chkptio->pid;
>     /* REGISTERS */
>     memcpy(&chkpt.regs, REGS, sizeof(struct pt_regs));
>     if (in_syscall(tsk))
>             intr_syscall(&chkpt.regs);
>     chkpt.datasize = datasz;
>     chkpt.brksize = brksz;
>     chkpt.stacksize = stacksz;
>     pack_write(f, (void *)&chkpt, sizeof(struct chkpt), 0);
```

```
>         /* TASK */
>         pack_write(f, (void*)tsk, THREAD_SIZE, 0);
>         /* MEMORY */
>         pack_write(f, (void *)tsk->mm->start_data, datasz, 0);
>         pack_write(f, (void *)tsk->mm->start_brk, brksz, 0);
>         pack_write(f, (void *)tsk->thread.esp, stacksz, 0);
>
>         pack_write(f, NULL, 0, 1); /* last packet */
>
>         return 0;
> }
>
> static int pack_write (struct file *f, char *buf, int len,
>           int last_pkt) {
>     static char *pack = NULL;
>     static long pos = 0;
>     int ret, to_copy, wrtn = 0;
>
>     if (pack==NULL)
>     {
>  pack=(char*)kmalloc(PACKET_SIZE, GFP_KERNEL);
>  if (!pack)
>  {
>      printk("pack_write: no mem!\n");
>      return -1;
>  }
>     }
>
>     while (len>0)
>     {
>  to_copy = (len>(PACKET_SIZE-pos))?(PACKET_SIZE-pos):(len);
>
>         memcpy(&(pack[pos]), buf+wrtn, to_copy);
>
>  pos += to_copy;
>  len -= to_copy;
>  wrtn +=to_copy;
>
>  if ( (pos==PACKET_SIZE) || (last_pkt) )
>  {
>      mm_segment_t fs = get_fs();
>
>      set_fs(KERNEL_DS);
>      ret = f->f_op->write(f, pack, pos, &(f->f_pos));
>      set_fs(fs);
>      if (ret!=pos)
>   return ret;
>
```

```
>      pos = 0;
>      if (last_pkt)
>      {
>   kfree(pack);
>   pack = NULL;
>      }
> }
>     }
>
>     if ( (last_pkt) && (pack!=NULL) )
>     {
>  if (pos!=0)
>  {
>      mm_segment_t fs = get_fs();
>
>      set_fs(KERNEL_DS);
>      wrtn = f->f_op->write(f, pack, pos, &f->f_pos);
>      set_fs(fs);
>  }
>  kfree(pack);
>  pack = NULL;
>  pos = 0;
>     }
>
>     return wrtn;
> }
>
> static int in_syscall(struct task_struct *tsk) {
>         unsigned char ins, opc;
>         long ret;
>         unsigned long flags;
>         struct pt_regs *regs;
>
>         spin_lock_irqsave(&runqueue_lock, flags);
>
>         regs = (((struct pt_regs *)(THREAD_SIZE + (unsigned long)tsk)) -
> 1);
>
>         get_user(ins, (unsigned char *)(regs->eip) - 2);
>         get_user(opc, (unsigned char *)(regs->eip) - 1);
>
>         if ((ins == 0xCD) && (opc == 0x80)) {
>                 ret = regs->orig_eax;
>         } else {
>                 ret = 0;
>         }
>
>         if (ret && ((regs->orig_eax < 0) || (regs->orig_eax >
```

```
> NR_syscalls))) {
>               INFO(("syscall number out of bounds %ld\n",
> regs->orig_eax));
>               ret = 0;
>       }
>
>       spin_unlock_irqrestore(&runqueue_lock, flags);
>
>       return ret;
> }
>
> static int intr_syscall(struct pt_regs *regs) {
>
>       /* handle in_syscall depending on syscall number */
>       switch(regs->orig_eax) {
>       case 4: /* write */
>               /* report interuption */
>               regs->eax = -EINTR;
>               break;
>       default:
>               /* restart */
>               regs->eax = regs->orig_eax;
>               regs->eip -= 2;
>               break;
>       }
>
>       return regs->orig_eax;
> }
>
> static int chkpt_restore(struct chkpt_ioctl *chkptio) {
>       struct task_struct *tsk, *saved_tsk;
>       struct chkpt chkpt;
>       struct file *f;
>
>       if ((tsk = find_task_by_pid(chkptio->pid)) == NULL) {
>               printk("...failed task not found %u\n", chkptio->pid);
>               return -1;
>       }
>
>
>       if ((saved_tsk = kmalloc(sizeof(struct task_struct), GFP_KERNEL))
> == NULL) {
>               printk("kmalloc failed\n");
>               return -1;
>       }
>
>       f = current->files->fd[chkptio->file];
>       do_read(f, &f->f_pos, (void *)&chkpt, sizeof(struct chkpt));
```

```
>         do_read(f, &f->f_pos, (void *)saved_tsk, THREAD_SIZE);
>
>         /* TASK */
>         if (pm_overwrite(tsk, saved_tsk) == NULL) {
>                 printk("...failed task overwrite\n");
>                 return -1;
>         }
>         /* REGISTERS */
>         memcpy(REGS, &chkpt.regs, sizeof(struct pt_regs));
>         /* MEMORY */
>         do_read(f, &f->f_pos, (void *)tsk->mm->start_data,
> chkpt.datasize);
>         do_read(f, &f->f_pos, (void *)chkpt.regs.esp, chkpt.stacksize);
>         do_read(f, &f->f_pos, (void *)tsk->mm->start_brk, chkpt.brksize);
>         kfree(saved_tsk);
>
>         return 0;
> }
>
>
> static int inline do_read(struct file *file, loff_t *offset,
>      char * addr, unsigned long count) {
>     mm_segment_t old_fs;
>     int ret;
>
>     if (!file->f_op->read)
>  return -ENOSYS;
>     old_fs = get_fs();
>     set_fs(get_ds());
>     ret = file->f_op->read(file, addr, count, offset);
>     set_fs(old_fs);
>     return ret;
> }
>
> /* overwrite tsk with the one in pmio, making a backup first,
>    then restoring all values dependant on the current local
>    OS state (except register values) */
> static struct task_struct *pm_overwrite(struct task_struct *tsk,
>                                         struct task_struct *saved_tsk) {
>  struct task_struct *backup;
>
>         if ((backup = kmalloc(sizeof(*backup), GFP_KERNEL)) == NULL) {
>                 printk("kmalloc failed\n");
>                 return NULL;
>         }
>
>         memcpy(backup, tsk, sizeof(struct task_struct));
>
```

```
>       write_lock_irq(&tasklist_lock);
>
>       memcpy(tsk, saved_tsk, THREAD_SIZE);
>
>       tsk->next_task = backup->next_task;
>       tsk->prev_task = backup->prev_task;
>
>       tsk->exec_domain = backup->exec_domain;
>       tsk->binfmt      = backup->binfmt;
>
>       tsk->pid         = backup->pid;
>       tsk->pgrp        = backup->pgrp;
>       tsk->tty_old_pgrp = backup->tty_old_pgrp;
>       tsk->session     = backup->session;
>       tsk->tgid        = backup->tgid;
>       tsk->leader      = backup->leader;
>
>       tsk->p_opptr = backup->p_opptr;
>       tsk->p_pptr  = backup->p_pptr;
>       tsk->p_cptr  = backup->p_cptr;
>       tsk->p_ysptr = backup->p_ysptr;
>       tsk->p_osptr = backup->p_osptr;
>
>       tsk->thread_group = backup->thread_group;
>
>       tsk->pidhash_next  = backup->pidhash_next;
>       tsk->pidhash_pprev = backup->pidhash_pprev;
>
>       tsk->real_timer    = backup->real_timer;
>
>       tsk->uid   = backup->uid;
>       tsk->euid  = backup->euid;
>       tsk->suid  = backup->suid;
>       tsk->fsuid = backup->fsuid;
>       tsk->gid   = backup->gid;
>       tsk->egid  = backup->egid;
>       tsk->sgid  = backup->sgid;
>       tsk->fsgid = backup->fsgid;
>
>       tsk->ngroups = backup->ngroups;
>       memcpy(tsk->groups, backup->groups, sizeof(tsk->groups));
>
>       tsk->cap_effective     = backup->cap_effective;
>       tsk->cap_inheritable   = backup->cap_inheritable;
>       tsk->cap_permitted     = backup->cap_permitted;
>       tsk->keep_capabilities = backup->keep_capabilities;
>
>       tsk->user = backup->user;
```

```
>
>       tsk->tty = backup->tty;
>
>       tsk->semundo     = backup->semundo;
>       tsk->semsleeping = backup->semsleeping;
>
>       tsk->fs = backup->fs;
>
>       tsk->files = backup->files;
>
>       tsk->sigmask_lock = backup->sigmask_lock;
>       tsk->sig          = backup->sig;
>       tsk->pending      = backup->pending;
>
>       tsk->sas_ss_sp     = backup->sas_ss_sp;
>       tsk->sas_ss_size   = backup->sas_ss_size;
>       tsk->notifier      = backup->notifier;
>       tsk->notifier_data = backup->notifier_data;
>       tsk->notifier_mask = backup->notifier_mask;
>
>       tsk->parent_exec_id = backup->parent_exec_id;
>       tsk->self_exec_id   = backup->self_exec_id;
>
>       tsk->alloc_lock = backup->alloc_lock;
>
>       tsk->mm        = backup->mm;
>       tsk->active_mm = backup->active_mm;
>
>       write_unlock_irq(&tasklist_lock);
>       kfree(backup);
>       return tsk;
> }
>
> /* standard module stuff */
>
> int __init chkpt_init(void)
> {
>  if (register_chrdev(CHKPT_DEV_MAJOR, "chkpt", &file_ops)) {
>   printk("...failed register_chrdev\n");
>   return -1;
>  }
>  return 0;
> }
>
> void __exit chkpt_exit(void)
> {
>  unregister_chrdev(CHKPT_DEV_MAJOR, "chkpt");
>  return;
```

> }
>
> EXPORT_SYMBOL(chkpt_init);
> EXPORT_SYMBOL(chkpt_exit);
> EXPORT_SYMBOL(chkpt_ioctl_handler);
>
> if anyone has any ideas, please let me know. thanx in advance.
>
> greetings
>
> marks
> -
> To unsubscribe from this list: send the line "unsubscribe linux-kernel" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at http://vger.kernel.org/majordomo-info.html
> Please read the FAQ at http://www.tux.org/lkml/
>

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