TI DSP, MCU, Xilinx Zynq FPGA Based Programming Expert Program

Instructor – Innova Lee (Sanghoon Lee)
gcccompil3r@gmail.com
Student – Howard Kim (Hyungjoo Kim)
mihaelkel@naver.com

Kernel Analyze – execve()

```
1710 SYSCALL_DEFINE3(execve,
1711         const char __user *, filename,
1712         const char __user *const __user *, argv,
1713         const char __user *const __user *, envp)
1714 {
1715     return do_execve(getname(filename), argv, envp);
1716 }
```

execve() is defined like above.
Let's go with do_execve();

```
1628 int do_execve(struct filename *filename,
1629     const char __user *const __user *__argv,
1630     const char __user *const __user *__envp)
1631 {
1632     struct user_arg_ptr argv = { .ptr.native = __argv };
1633     struct user_arg_ptr envp = { .ptr.native = __envp };
1634     return do_execveat_common(AT_FDCWD, filename, argv, envp, 0);
1635 }
```

Before entering do_execveat_common(), find out struct user_arg_ptr and AT_FDCWD

```
 393 struct user_arg_ptr {
 394 #ifdef CONFIG_COMPAT
 395     bool is_compat;
 396 #endif
 397     union {
 398         const char __user *const __user *native;
 399 #ifdef CONFIG_COMPAT
 400         const compat_uptr_t __user *compat;
 401 #endif
 402     } ptr;
 403 };
```

I don't exactly know why user_arg_ptr has to exist, I mean "Can We use argv and envp as it is?", CONFIG_COMPAT impies it is not. It may be associated with compatibility.
OK, argv and envp has been set to argv and evnp (has a type of struct user_arg_ptr).

```
Cscope tag: AT_FDCWD                                    1634,30-33    94%
   #   line  filename / context / line
   1     56  include/uapi/linux/fcntl.h <<AT_FDCWD>>
             #define AT_FDCWD -100
   2     56  /usr/include/linux/fcntl.h <<AT_FDCWD>>
             #define AT_FDCWD -100
```

AT_FDCWD is defined -100, Go with do_execveat_common

```
1484 /*-100, filename, argv, envp, 0*/
1485 static int do_execveat_common(int fd, struct filename *filename,
1486                   struct user_arg_ptr argv,
1487                   struct user_arg_ptr envp,
1488                   int flags)
1489 {
1490     /*
1491     *   fd = -100
1492     *   filename = filename
1493     *   user_arg_ptr argv = argv
1494     *   user_arg_ptr envp = envp
1495     *   flags = 0
1496     */
1497     char *pathbuf = NULL;
1498     struct linux_binprm *bprm;
1499     struct file *file;
1500     struct files_struct *displaced;
1501     int retval;
1502
1503     if (IS_ERR(filename))
1504         return PTR_ERR(filename);
```

Memorize the parameters, keep going.

exec.c (~/kernel/linux-4.4/fs) - VIM

```
1512     if ((current->flags & PF_NPROC_EXCEEDED) &&
1513         atomic_read(&current_user()->processes) > rlimit(RLIMIT_NPROC)) {
1514         retval = -EAGAIN;
1515         goto out_ret;
1516     }
```

From now on, I'll capture all of shots with the path as not to be lost.
The condition impies execve doesn't act(goto out_ret), if the resource has exceeded.

```
1518     /* We're below the limit (still or again), so we don't want to make
1519      * further execve() calls fail. */
1520     current->flags &= ~PF_NPROC_EXCEEDED;
```

PF_NPROC_EXCEEDED bit in current→flags get clear.

```
1522     retval = unshare_files(&displaced);
1523     if (retval)
1524         goto out_ret;
```

We can predict unshare_files() returns 0, and set some values to displaced (because send address value as parameter). Go with unshare_files();

fork.c (~/kernel/linux-4.4/kernel) - VIM

```
2132 int unshare_files(struct files_struct **displaced)
2133 {
2134     struct task_struct *task = current;
2135     struct files_struct *copy = NULL;
2136     int error;
2137
2138     error = unshare_fd(CLONE_FILES, &copy);
2139     if (error || !copy) {
2140         *displaced = NULL;
2141         return error;
2142     }
2143     *displaced = task->files;
2144     task_lock(task);
2145     task->files = copy;
2146     task_unlock(task);
2147     return 0;
2148 }
```

keep going with unshated_fd()

fork.c (~/kernel/linux-4.4/kernel) - VIM

```
1989 static int unshare_fd(unsigned long unshare_flags, struct files_struct **new_fdp)
1990 {
1991     struct files_struct *fd = current->files;
1992     int error = 0;
1993
1994     if ((unshare_flags & CLONE_FILES) &&
1995         (fd && atomic_read(&fd->count) > 1)) {
1996         *new_fdp = dup_fd(fd, &error);
1997         if (!*new_fdp)
1998             return error;
1999     }
2000
2001     return 0;
2002 }
```

new_fdp is displaced, this will get current task's file descriptor.
That is, the new task created by "execve" share origin task's file descriptor.
I did't know it is true or not, so tested it.
(Sorry, I apply my old code to this, so the code is not neat)

newpgm.c (~/HomeworkBackup/23th) - VIM

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <fcntl.h>
int main(void){
    int status;
    pid_t pid;
    char* argv[] = {"./newpgm","newpgm","one","two",NULL};
    char* env[] = {"name = OS_Hacker","age = 20",NULL};
    int fd;
    fd = open("test.txt",O_RDONLY,0644);
    printf("origin fd = %d\n",fd);
    execve("./newpgm",argv,env);
    printf("not be displayed\n");
    return 0;

}
```

test.c 1,1 All

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
int main(int argc,char** argv,char** envp){
    int i;
    for(i=0; argv[i];i++)
        printf("argv[%d] = [%s]\n",i,argv[i]);
    for(i=0;  envp[i] ;i++)
        printf("envp[%d] = [%s]\n",i,envp[i]);
    char buf[1024] = {'\0',};
    read(3,buf,sizeof(buf));
    printf("test.txt :\n%s",buf);

    return 0;

}
```

newpgm.c 1,1 All

If they did't share file descriptor, buf in newpgm.c would be NULL, or buf would be some text in test.txt (Sorry to do hard code fd as 3, not using IPC for sending fd(file descriptor)

```
howi@ubuntu:~/HomeworkBackup/23th$ ./test
origin fd = 3
argv[0] = [./newpgm]
argv[1] = [newpgm]
argv[2] = [one]
argv[3] = [two]
envp[0] = [name = OS_Hacker]
envp[1] = [age = 20]
test.txt :
aaa
bb
cc
```

Well, They clearly share the file descripor

fork.c (~/kernel/linux-4.4/kernel) - VIM

```
1989 static int unshare_fd(unsigned long unshare_flags, struct files_struct **new_fdp)
1990 {
1991     struct files_struct *fd = current->files;
1992     int error = 0;
1993
1994     if ((unshare_flags & CLONE_FILES) &&
1995         (fd && atomic_read(&fd->count) > 1)) {
1996         *new_fdp = dup_fd(fd, &error);
1997         if (!*new_fdp)
1998             return error;
1999     }
2000
2001     return 0;
2002 }
```

Back to the unshare_fd().
unshare_flags was CLONE_FILES, new_fdp was &copy.
This function will duplicate current→files to *(&copy) when calling dup_fd(fd,&error).
Go with dup_fp!

file.c (~/kernel/linux-4.4/fs) - VIM

```
290 struct files_struct *dup_fd(struct files_struct *oldf, int *errorp)
291 {
292     struct files_struct *newf;
293     struct file **old_fds, **new_fds;
294     int open_files, i;
295     struct fdtable *old_fdt, *new_fdt;
296
```

This may copy current→file, return that(as struct files_struct *newf)
keep going!

file.c (~/kernel/linux-4.4/fs) - VIM

```
298     newf = kmem_cache_alloc(files_cachep, GFP_KERNEL);
299     if (!newf)
300         goto out;
301
302     atomic_set(&newf->count, 1);
```

Allocate memory to newf, if the allocation fails, goto out:
atomic_set() would set &newf→count = 1

file.c (~/kernel/linux-4.4/fs) - VIM

```
304     spin_lock_init(&newf->file_lock);
305     newf->resize_in_progress = false;
306     init_waitqueue_head(&newf->resize_wait);
307     newf->next_fd = 0;
308     new_fdt = &newf->fdtab;
309     new_fdt->max_fds = NR_OPEN_DEFAULT;
310     new_fdt->close_on_exec = newf->close_on_exec_init;
311     new_fdt->open_fds = newf->open_fds_init;
312     new_fdt->full_fds_bits = newf->full_fds_bits_init;
313     new_fdt->fd = &newf->fd_array[0];
314
315     spin_lock(&oldf->file_lock);
316     old_fdt = files_fdtable(oldf);
317     open_files = count_open_files(old_fdt);
318
```

new_fdt was struct fdtable *new_fdt, these codes are for setting new_fdt

file.c (~/kernel/linux-4.4/fs) - VIM

```
319     /*
320      * Check whether we need to allocate a larger fd array and fd set.
321      */
322     while (unlikely(open_files > new_fdt->max_fds)) {
323         spin_unlock(&oldf->file_lock);
324
325         if (new_fdt != &newf->fdtab)
326             __free_fdtable(new_fdt);
327
328         new_fdt = alloc_fdtable(open_files - 1);
329         if (!new_fdt) {
330             *errorp = -ENOMEM;
331             goto out_release;
332         }
333
334         /* beyond sysctl_nr_open; nothing to do */
335         if (unlikely(new_fdt->max_fds < open_files)) {
336             __free_fdtable(new_fdt);
337             *errorp = -EMFILE;
338             goto out_release;
339         }
340
341         /*
342          * Reacquire the oldf lock and a pointer to its fd table
343          * who knows it may have a new bigger fd table. We need
344          * the latest pointer.
345          */
346         spin_lock(&oldf->file_lock);
347         old_fdt = files_fdtable(oldf);
348         open_files = count_open_files(old_fdt);
349     }
```

the while loop will execute when opend file number exceed the max value.
If so, fdtable would be newly allocated.

file.c (~/kernel/linux-4.4/fs) - VIM

```
351     copy_fd_bitmaps(new_fdt, old_fdt, open_files);
352
353     old_fds = old_fdt->fd;
354     new_fds = new_fdt->fd;
```

Copy fd_bitmaps, and save fd to fds.

file.c + (~/kernel/linux-4.4/fs) - VIM

```
67 /*                          new_fdt, old_fdt, open_files        */
68 static void copy_fd_bitmaps(struct fdtable *nfdt, struct fdtable *ofdt,
69                 unsigned int count)
70 {
71     unsigned int cpy, set;
72
73     cpy = count / BITS_PER_BYTE;
74     set = (nfdt->max_fds - count) / BITS_PER_BYTE;
75     memcpy(nfdt->open_fds, ofdt->open_fds, cpy);
76     memset((char *)nfdt->open_fds + cpy, 0, set);
77     memcpy(nfdt->close_on_exec, ofdt->close_on_exec, cpy);
78     memset((char *)nfdt->close_on_exec + cpy, 0, set);
79
80     cpy = BITBIT_SIZE(count);
81     set = BITBIT_SIZE(nfdt->max_fds) - cpy;
82     memcpy(nfdt->full_fds_bits, ofdt->full_fds_bits, cpy);
83     memset((char *)nfdt->full_fds_bits + cpy, 0, set);
84 }
```

In copy_fd_bitmaps(), there are some memcpy() and memset().

file.c (~/kernel/linux-4.4/fs) - VIM

```
357     for (i = open_files; i != 0; i--) {
358         struct file *f = *old_fds++;
359         if (f) {
360             get_file(f);
361         } else {
362             /*
363              * The fd may be claimed in the fd bitmap but not yet
364              * instantiated in the files array if a sibling thread
365              * is partway through open().  So make sure that this
366              * fd is available to the new process.
367              */
368             __clear_open_fd(open_files - i, new_fdt);
369         }
370         rcu_assign_pointer(*new_fds++, f);
371     }
372     spin_unlock(&oldf->file_lock);
```

I can guess this is searching all of the opened file, but don't know what they do.
But the note refers to this is for making sure some problems by sibling thread.

file.c (~/kernel/linux-4.4/fs) - VIM

```
374     /* clear the remainder */
375     memset(new_fds, 0, (new_fdt->max_fds - open_files) * sizeof(struct file *));
376
377     Files      assign_pointer(newf->fdt, new_fdt);
378
379     return newf;
380
381 out_release:
382     kmem_cache_free(files_cachep, newf);
383 out:
384     return NULL;
385 }
```

And returns newf.

fork.c (~/kernel/linux-4.4/kernel) - VIM

```
1989 static int unshare_fd(unsigned long unshare_flags, struct files_struct **new_fdp)
1990 {
1991     struct files_struct *fd = current->files;
1992     int error = 0;
1993
1994     if ((unshare_flags & CLONE_FILES) &&
1995         (fd && atomic_read(&fd->count) > 1)) {
1996         *new_fdp = dup_fd(fd, &error);
1997         if (!*new_fdp)
1998             return error;
1999     }
2000
2001     return 0;
2002 }
```

Back to the unshare_fd(), If dup_fd() executed successfully, there are some values in *new_fdp (which is *(&copy)). And return 0;

fork.c (~/kernel/linux-4.4/kernel) - VIM

```
2132 int unshare_files(struct files_struct **displaced)
2133 {
2134     struct task_struct *task = current;
2135     struct files_struct *copy = NULL;
2136     int error;
2137
2138     error = unshare_fd(CLONE_FILES, &copy);
2139     if (error || !copy) {
2140         *displaced = NULL;
2141         return error;
2142     }
2143     *displaced = task->files;
2144     task_lock(task);
2145     task->files = copy;
2146     task_unlock(task);
2147     return 0;
2148 }
```

set the copy(fixed by unshare_fd()) to task→files, which is current→files.
And return 0;

exec.c (~/kernel/linux-4.4/fs) - VIM

```
1522     retval = unshare_files(&displaced);
1523     if (retval)
1524         goto out_ret;
1525
1526     retval = -ENOMEM;
1527     bprm = kzalloc(sizeof(*bprm), GFP_KERNEL);
1528     if (!bprm)
1529         goto out_files;
```

retval set from unshare_files() would be 0, bprm allocated.

exec.c (~/kernel/linux-4.4/fs) - VIM

```
1531     retval = prepare_bprm_creds(bprm);
1532     if (retval)
1533         goto out_free;
1534
1535     check_unsafe_exec(bprm);
1536     current->in_execve = 1;
1537     /*-100, filename, 0*/
1538     file = do_open_execat(fd, filename, flags);
1539     retval = PTR_ERR(file);
1540     if (IS_ERR(file))
1541         goto out_unmark;
```

Analyzing prepare_bprm_creds() and do_open_execat() is out of my ability..

exec.c (~/kernel/linux-4.4/fs) - VIM

```
1543     sched_exec();
```

enter it

sched.h (~/kernel/linux-4.4/include/linux) - VIM

```
2319 /* sched_exec is called by processes performing an exec */
2320 #ifdef CONFIG_SMP
2321 extern void sched_exec(void);
2322 #else
2323 #define sched_exec()   {}
2324 #endif
```

sched_exec() defined when single-CPU which is UMA structure, but We have NUMA.

exec.c (~/kernel/linux-4.4/fs) - VIM

```
1545     bprm->file = file;
1546     if (fd == AT_FDCWD || filename->name[0] == '/') {
1547         bprm->filename = filename->name;
1548     } else {
1549         if (filename->name[0] == '\0')
1550             pathbuf = kasprintf(GFP_TEMPORARY, "/dev/fd/%d", fd);
1551         else
1552             pathbuf = kasprintf(GFP_TEMPORARY, "/dev/fd/%d/%s",
1553                                 fd, filename->name);
1554         if (!pathbuf) {
1555             retval = -ENOMEM;
1556             goto out_unmark;
1557         }
1558         /*
1559          * Record that a name derived from an O_CLOEXEC fd will be
1560          * inaccessible after exec. Relies on having exclusive access to
1561          * current->files (due to unshare_files above).
1562          */
1563         if (close_on_exec(fd, rcu_dereference_raw(current->files->fdt)))
1564             bprm->interp_flags |= BINPRM_FLAGS_PATH_INACCESSIBLE;
1565         bprm->filename = pathbuf;
1566     }
```

fd is AT_FDCWD, so let’s go with if().
It’s simple. Just do
bprm→filename = filename→name

exec.c (~/kernel/linux-4.4/fs) - VIM

```
1567     bprm->interp = bprm->filename;
1568
1569     retval = bprm_mm_init(bprm);
1570     if (retval)
1571         goto out_unmark;
1572
1573     bprm->argc = count(argv, MAX_ARG_STRINGS);
1574     if ((retval = bprm->argc) < 0)
1575         goto out;
1576
1577     bprm->envc = count(envp, MAX_ARG_STRINGS);
1578     if ((retval = bprm->envc) < 0)
1579         goto out;
1580
1581     retval = prepare_binprm(bprm);
1582     if (retval < 0)
1583         goto out;
1584
1585     retval = copy_strings_kernel(1, &bprm->filename, bprm);
1586     if (retval < 0)
1587         goto out;
1588
1589     bprm->exec = bprm->p;
1590     retval = copy_strings(bprm->envc, envp, bprm);
1591     if (retval < 0)
1592         goto out;
1593
1594     retval = copy_strings(bprm->argc, argv, bprm);
1595     if (retval < 0)
1596         goto out;
1597
1598     retval = exec_binprm(bprm);
1599     if (retval < 0)
1600         goto out;
```

Set some values to bprm and exec_binprm(bprm);

exec.c (~/kernel/linux-4.4/fs) - VIM

```
1602     /* execve succeeded */
1603     current->fs->in_exec = 0;
1604     current->in_execve = 0;
1605     acct_update_integrals(current);
1606     task_numa_free(current);
1607     free_bprm(bprm);
1608     kfree(pathbuf);
1609     putname(filename);
1610     if (displaced)
1611         put_files_struct(displaced);
1612     return retval;
```

This may free the current.

Remind how execve() function works.

```
int main(int argc, char** argv, char** envp)
{
        :
        execve("filename",argv,envp);
        printf("this would not be displayed!\n");
        :
}
```