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 Analysis – fork

The fork function can be found by grepping “fork” and “SYSCALL”, fork is one of the SYSTEM CALL.

```
howi@ubuntu:~/kernel/linux-4.4$ grep -rn "fork" ./ | grep SYSCALL
./kernel/fork.c:1842:SYSCALL_DEFINE0(fork)
```

Then, let’s find out what is in **./kernel/fork.c line:1842**

```
1842 SYSCALL_DEFINE0(fork)
1843 {
1844 #ifdef CONFIG_MMU
1845     return _do_fork(SIGCHLD, 0, 0, NULL, NULL, 0);
1846 #else
1847     /* can not support in nommu mode */
1848     return -EINVAL;
1849 #endif
1850 }
1851 #endif
```

the fork system call returns **_do_fork(SIGCHLD, 0, 0, NULL, NULL, 0)**;
It also displays fork only can be called when MMU exist.
Before entering to _do_fork(), You’d better pasting the parameters, SIGCHLD, 0, 0, NULL, NULL, 0

```
1720 /*  SIGCHLD, 0, 0, NULL, NULL, 0*/
1721 long _do_fork(unsigned long clone_flags,
1722           unsigned long stack_start,
1723           unsigned long stack_size,
1724           int __user *parent_tidptr,
1725           int __user *child_tidptr,
1726           unsigned long tls)
1727 {
```

Now, We’ve just known what the parameters mean.

```
1728     struct task_struct *p;
```

The pointer variable has a type of struct task_struct is right here, Maybe it is for “child process”.

```
1738     /*
1739         clone_flags == SIGCHLD, 0x00000011
1740         stack_start == 0
1741         stack_size  == 0
1742         parent_tidptr == NULL
1743         child_tidptr == NULL
1744         tls == 0
1745     */
```

Before processing, make a footnote what the variables(parameters) mean or have.

```
1747     if (!(clone_flags & CLONE_UNTRACED)) {
1748         /* CLONE_VFORK = 0x00004000
1749            clone_flags로 들어온 것에 SIGCHLD가 있는지 없는지에 따라
1750             tracce가 PTRACE_EVENT_FORK = 1 혹은 PTRACE_EVENT_CLONE= 3이 됨.*/
1751         if (clone_flags & CLONE_VFORK)
1752             trace = PTRACE_EVENT_VFORK;
1753         /* CSIGNAL = 0x000000ff */
1754         else if ((clone_flags & CSIGNAL) != SIGCHLD)
1755             trace = PTRACE_EVENT_CLONE;
1756         else
1757             trace = PTRACE_EVENT_FORK;
1758         /*current는 현재 구동중인 task_struct
1759         trace == PTRACE_EVENT_FORK ( == 1)
1760         */
1761         /*
1762             task->ptrace & 0x00000009 값을 리턴,
1763             ptrace : 디버깅용, 조사 더 필요, 지금은 관심무
1764         */
1765         if (likely(!ptrace_event_enabled(current, trace)))
1766             trace = 0;
1767     }
```

CLONE_UNTRACED is shown, We don't know what it is.

```
Cscope tag: CLONE_UNTRACED                                  1747,27-30    81%
   #   line  filename / context / line
   1     22  include/uapi/linux/sched.h <<CLONE_UNTRACED>>
             #define CLONE_UNTRACED 0x00800000
   2     22  /usr/include/linux/sched.h <<CLONE_UNTRACED>>
             #define CLONE_UNTRACED 0x00800000
```

I don't know what it means, but I can find out what it has. It is defined as 0x00800000.
clone_flags is SIGCHLD, and SIGCHLD == 0x00000011.
!(clone_flags&CLONE_UNTRACED) is TRUE, So, the If state should be checked.
We can predict the "trace" variable's value will be fixed according to clone_flags
trace will be fixed as PTRACE_EVENT_FORK, which is 1.

```
Cscope tag: PTRACE_EVENT_FORK                               1757,21-30    81%
   #   line  filename / context / line
   1     73  include/uapi/linux/ptrace.h <<PTRACE_EVENT_FORK>>
             #define PTRACE_EVENT_FORK 1
   2     73  /usr/include/linux/ptrace.h <<PTRACE_EVENT_FORK>>
             #define PTRACE_EVENT_FORK 1
```

We get the trace value. likely() means something in here as a parameter would be TRUE
ptrace_event_enabled(current, trace) function may check "current" task has no problem to do fork (when trace == 1) then, trace will be fixed as 0.

```
1776     /*0x00000011, 0, 0, NULL, NULL, 0, 0*/
1777     p = copy_process(clone_flags, stack_start, stack_size,
1778              child_tidptr, NULL, trace, tls);
```

in the below, copy_process() is shown. We can guess "child process" would be created, saved to p

```
1254 /*0x00000011, 0, 0, NULL, NULL, 0, 0*/
1255 static struct task_struct *copy_process(unsigned long clone_flags,
1256                     unsigned long stack_start,
1257                     unsigned long stack_size,
1258                     int __user *child_tidptr,
1259                     struct pid *pid,
1260                     int trace,
1261                     unsigned long tls)
1262 {
```

paste parameter's values.

```
1263     int retval;
1264     struct task_struct *p;
1265     /*      CGROUP_CANFORK_COUNT == 0    */
1266     void *cgrp_ss_priv[CGROUP_CANFORK_COUNT] = {};
```

Some variables is here. This function would return struct task_struct *p

```
1268     /*
1269     CLONE_NEWNS == 0x00020000
1270     CLONE_FS    == 0x00000200
1271     CLONE_NEWSER== 0x10000000
1272     */
1273     if ((clone_flags & (CLONE_NEWNS|CLONE_FS)) == (CLONE_NEWNS|CLONE_FS))
1274         return ERR_PTR(-EINVAL);
1275
1276     if ((clone_flags & (CLONE_NEWUSER|CLONE_FS)) == (CLONE_NEWUSER|CLONE_FS))
1277         return ERR_PTR(-EINVAL);
```

A sort of If state checks clone_flags's value is valid. If it is not, return error.
The error lists are described in "A man page for clone".

```
ERRORS
       EAGAIN Too many processes are already running; see fork(2).

       EINVAL CLONE_SIGHAND was specified, but CLONE_VM was not.  (Since Linux
              2.6.0-test6.)

       EINVAL CLONE_THREAD was specified, but CLONE_SIGHAND was  not.   (Since
              Linux 2.5.35.)

       EINVAL Both CLONE_FS and CLONE_NEWNS were specified in flags.

       EINVAL (since Linux 3.9)
              Both CLONE_NEWUSER and CLONE_FS were specified in flags.

       EINVAL Both CLONE_NEWIPC and CLONE_SYSVSEM were specified in flags.

       EINVAL One (or both) of CLONE_NEWPID or CLONE_NEWUSER and one (or both)
              of CLONE_THREAD or CLONE_PARENT were specified in flags.

       EINVAL Returned  by  clone()  when  a  zero  value  is  specified   for
              child_stack.
```

After checking the errors, We can find security_task_create() function

```
1324     retval = security_task_create(clone_flags);
1325     if (retval)
1326         goto fork_out;
```

cscope caught 2 pieces of the funcion define.

```
Cscope tag: security_task_create                                1324,29-32    61%
   #   line  filename / context / line
   1    813  include/linux/security.h <<security_task_create>>
             static inline int security_task_create(unsigned long clone_flags)
   2    847  security/security.c <<security_task_create>>
             int security_task_create(unsigned long clone_flags)
```

First result has inline option, which means it only can be used in the path <include/linux/security.h>
So, let's go with second result.

```
846 /*clone_flags == 0x00000011 , SIGCHLD*/
847 int security_task_create(unsigned long clone_flags)
848 {
849     return call_int_hook(task_create, 0, clone_flags);
850 }
```

It looks simple, but is not.
We should find what task_create is

```
Cscope tag: task_create                                         849,25-28     44%
   #   line  filename / context / line
   1   1444  include/linux/lsm_hooks.h <<task_create>>
             int (*task_create)(unsigned long clone_flags);
   2   1705  include/linux/lsm_hooks.h <<task_create>>
             struct list_head task_create;
   3   1676  security/security.c <<task_create>>
             .task_create = LIST_HEAD_INIT(security_hook_heads.task_create),
```

task_create is a function pointer, initialized as LIST_HEAD_INIT()
third one might be our finds

```
20 #define LIST_HEAD_INIT(name) { &(name), &(name) }
```

LIST_HEAD_INIT() is a macro to initialize the parameter, in this case,
initialize security_hook_heads.task_create
Let's get in call_int_hook()

```
116 /*  task_create, 0, clone_flags */
117 #define call_int_hook(FUNC, IRC, ...) ({            \
118     int RC = IRC;                                   \
119     do {                                            \
120         struct security_hook_list *P;               \
121                                                     \
122         list_for_each_entry(P, &security_hook_heads.FUNC, list) { \
123             RC = P->hook.FUNC(__VA_ARGS__);         \
124             if (RC != 0)                            \
125                 break;                              \
126         }                                           \
127     } while (0);                                    \
128     RC;                                             \
129 })
```

this macro returns RC value.
The RC is initialized 0
list_for_each_entry could change the value.
In the backward of the macro, there is {}.
We can predict list_for_each_enty(,,) macro is consist of for(;;)
Check the list_for_each_entry !

```
Cscope tag: list_for_each_entry                                  122,10-16      5%
   #   line  filename / context / line
   1    314  drivers/gpu/drm/nouveau/include/nvif/list.h <<list_for_each_entry>>
             #define list_for_each_entry(pos, head, member) \
   2    411  drivers/gpu/drm/radeon/mkregtable.c <<list_for_each_entry>>
             #define list_for_each_entry(pos, head, member) \
   3    446  include/linux/list.h <<list_for_each_entry>>
             #define list_for_each_entry(pos, head, member) \
   4     48  scripts/kconfig/list.h <<list_for_each_entry>>
             #define list_for_each_entry(pos, head, member) \
   5     58  tools/firewire/list.h <<list_for_each_entry>>
             #define list_for_each_entry(pos, list, member) \
   6    116  tools/virtio/linux/kernel.h <<list_for_each_entry>>
             #define list_for_each_entry(a, b, c) while (0)
```

We can easily determine that we get in third one.

```
447 /*P, &security_hook_heads.FUNC, list*/
448 #define list_for_each_entry(pos, head, member)              \
449     for (pos = list_first_entry(head, typeof(*pos), member);    \
450          &pos->member != (head);                    \
451          pos = list_next_entry(pos, member))
```

that's right!

It is defined for().

Initial value is pos = list_first_entry(head, typeof(*pos), member);

which is pos = list_first_entry(&security_hook_heads.tast_create, struct security_hook_list, list);

condition is &pos->member != (head);

repeated is pos = list_net_enty(pos, member);

that is,

for(pos = list_first_entry(&security_hook_heads.tast_create, struct security_hook_list, list);
    pos = list_first_entry(&security_hook_heads.tast_create, struct security_hook_list, list);
    pos = list_next_enty(pos, member)
)

then, let's determine what the macros(list_first_entry, list_next_entry) are.

```
362 /* &security_hook_heads.tast_create, struct security_hook_list, list */
363 #define list_first_entry(ptr, type, member) \
364     list_entry((ptr)->next, type, member)
```

```
351 /* &security_hook_heads.tast_create->next, struct security_hook_list, list */
352 #define list_entry(ptr, type, member) \
353     container_of(ptr, type, member)
```

```
812 /* &security_hook_heads.tast_create->next, struct security_hook_list, list */
813 #define container_of(ptr, type, member) ({          \
814     const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
815     (type *)( (char *)__mptr - offsetof(type,member) );})
```

These may return the first entry in list

We can analyze list_next_enty() as same method

```
447 /*P, &security_hook_heads.FUNC, list*/
448 #define list_for_each_entry(pos, head, member)              \
449     for (pos = list_first_entry(head, typeof(*pos), member);    \
450          &pos->member != (head);                    \
451          pos = list_next_entry(pos, member))
```

that is, for() would search every entry of list

```
116 /*  task_create, 0, clone_flags */
117 #define call_int_hook(FUNC, IRC, ...) ({            \
118     int RC = IRC;                                   \
119     do {                                            \
120         struct security_hook_list *P;               \
121                                                     \
122         list_for_each_entry(P, &security_hook_heads.FUNC, list) { \
123             RC = P->hook.FUNC(__VA_ARGS__);         \
124             if (RC != 0)                            \
125                 break;                              \
126         }                                           \
127     } while (0);                                    \
128     RC;                                             \
129 })
```

then, what is RC = P→hook.task_create(SIG_CHLD) ?

```
1833 struct security_hook_list {
1834     struct list_head        list;
1835     struct list_head        *head;
1836     union security_list_options hook;
1837 };
```

I think this process(for(){ }) would prevent fork from creating a child process recursively.
That is, “child process” created from this fork() function CAN’T make another “child process”.

```
1324     retval = security_task_create(clone_flags);
1325     if (retval)
1326         goto fork_out;
```

Back to the copy_process, If retval is not 0, which means the current task is the task created this fork function, go to fork_out;

```
1328     retval = -ENOMEM;
1329     p = dup_task_struct(current);
1330     if (!p)
1331         goto fork_out;
```

Through a series of procedures, We created a new task, filled nothing.
So, We have to duplicate the current, and set to p

```
335 //struct task_struct *orig == current
336 static struct task_struct *dup_task_struct(struct task_struct *orig)
337 {
338     struct task_struct *tsk;
339     struct thread_info *ti;
340     /*node = current->pref_node_fork, NUMA 관련 변수*/
341     int node = tsk_fork_get_node(orig);
342     int err;
```

There are some variables.
Can guess struct task_struct *tsk would be returned, struct thread_info *ti is saved to tsk.
Let’s find out what tsk_fork_get_node(current) is. But the name implies this is for NUMA management.

```
215 /* called from do_fork() to get node information for about to be created task */
216 int tsk_fork_get_node(struct task_struct *tsk)
217 {
218 #ifdef CONFIG_NUMA
219     if (tsk == kthreadd_task)
220         return tsk->pref_node_fork;
221 #endif
222     return NUMA_NO_NODE;
223 }
```

The note states our prediction is correct.
So, node = current→pref_node_fork;

```
344     tsk = alloc_task_struct_node(node);
345     if (!tsk)
346         return NULL;
```

tsk is child process, it needs to get allocated some memory.
So, alloc_task_strcut_node(node) will do that.

```
Cscope tag: alloc_task_struct_node                                  344,24-27     15%
   #   line  filename / context / line
   1     76  arch/ia64/include/asm/thread_info.h <<alloc_task_struct_node>>
             #define alloc_task_struct_node(node) \
   2    139  kernel/fork.c <<alloc_task_struct_node>>
             static inline struct task_struct *alloc_task_struct_node(int node)
```

Second one includes static inline, but the path is same to current position. Enter it.

```
136 #ifndef CONFIG_ARCH_TASK_STRUCT_ALLOCATOR
137 static struct kmem_cache *task_struct_cachep;
138
139 static inline struct task_struct *alloc_task_struct_node(int node)
140 {
141     /*node == current->pref_node_fork*/
142     return kmem_cache_alloc_node(task_struct_cachep, GFP_KERNEL, node);
143 }
```

task_struct_cachep is declared above, node == current→pref_node_fork.
But, GFP_KERNEL's value is so complicated.

```
   1    238  include/linux/gfp.h <<GFP_KERNEL>>
             #define GFP_KERNEL (__GFP_RECLAIM | __GFP_IO | __GFP_FS)
```

If the value block me, I'll find out that again. Let's proceed.

```
Cscope tag: kmem_cache_alloc_node                                   142,15-18      5%
   #   line  filename / context / line
   1    323  include/linux/slab.h <<kmem_cache_alloc_node>>
             void *kmem_cache_alloc_node(struct...int node) __assume_slab_alignment;
   2    330  include/linux/slab.h <<kmem_cache_alloc_node>>
             static __always_inline void *kmem_...m_cache *s, gfp_t flags, int node)
   3   3456  mm/slab.c <<kmem_cache_alloc_node>>
             void *kmem_cache_alloc_node(struct... *cachep, gfp_t flags, int nodeid)
   4    575  mm/slob.c <<kmem_cache_alloc_node>>
             void *kmem_cache_alloc_node(struct...ache *cachep, gfp_t gfp, int node)
   5   2598  mm/slub.c <<kmem_cache_alloc_node>>
             void *kmem_cache_alloc_node(struct...ache *s, gfp_t gfpflags, int node)
```

We use slab allocator, go with number 3.

```
3144 slab_alloc_node(struct kmem_cache *cachep, gfp_t flags, int nodeid,
3145             unsigned long caller)
3146 {
3147     unsigned long save_flags;
3148     void *ptr;
3149     int slab_node = numa_mem_id();
3150
3151     flags &= gfp_allowed_mask;
3152
3153     lockdep_trace_alloc(flags);
3154
3155     if (slab_should_failslab(cachep, flags))
3156         return NULL;
3157
3158     cachep = memcg_kmem_get_cache(cachep, flags);
3159
3160     cache_alloc_debugcheck_before(cachep, flags);
3161     local_irq_save(save_flags);
3162
3163     if (nodeid == NUMA_NO_NODE)
3164         nodeid = slab_node;
3165
3166     if (unlikely(!get_node(cachep, nodeid))) {
3167         /* Node not bootstrapped yet */
3168         ptr = fallback_alloc(cachep, flags);
3169         goto out;
3170     }
3171
3172     if (nodeid == slab_node) {
3173         /*
3174          * Use the locally cached objects if possible.
3175          * However ____cache_alloc does not allow fallback
```

This is out of my knowledge. Back to dup_task_struct

```
344     tsk = alloc_task_struct_node(node);
345     if (!tsk)
346         return NULL;
347
348     ti = alloc_thread_info_node(tsk, node);
349     if (!ti)
350         goto free_tsk;
351
352     err = arch_dup_task_struct(tsk, orig);
353     if (err)
354         goto free_ti;
355
356     tsk->stack = ti;
```

Simply, Allocate some memory to child process, paste the parent's task to child.

```
373     tsk->stack_canary = get_random_int();
```

Set the stack protector randomly.

```
376     /*
377      * One for us, one for whoever does the "release_task()" (usually
378      * parent)
379      */
380     atomic_set(&tsk->usage, 2);
381 #ifdef CONFIG_BLK_DEV_IO_TRACE
382     tsk->btrace_seq = 0;
383 #endif
384     tsk->splice_pipe = NULL;
385     tsk->task_frag.page = NULL;
386     tsk->wake_q.next = NULL;
387
388     account_kernel_stack(ti, 1);
389
390     return tsk;
391
392 free_ti:
393     free_thread_info(ti);
394 free_tsk:
395     free_task_struct(tsk);
396     return NULL;
397 }
```

And return tsk, which is newly allocated task. Back to copy_process.

```
1350     retval = copy_creds(p, clone_flags);
1351     if (retval < 0)
1352         goto bad_fork_free;
```

Copy credential information to child process.
And after doing many other things, return p (child process)
back to _do_fork.

```
1784     if (!IS_ERR(p)) {
1785         struct completion vfork;
1786         struct pid *pid;
1787 
1788         trace_sched_process_fork(current, p);
1789         /*task_struct* p, PIDTYPE_PID(0)*/
1790         pid = get_task_pid(p, PIDTYPE_PID);
1791         nr = pid_vnr(pid);
1792 
1793         if (clone_flags & CLONE_PARENT_SETTID)
1794             put_user(nr, parent_tidptr);
1795 
1796         if (clone_flags & CLONE_VFORK) {
1797             p->vfork_done = &vfork;
1798             init_completion(&vfork);
1799             get_task_struct(p);
1800         }
1801 
1802         wake_up_new_task(p);
1803 
1804         /* forking complete and child started to run, tell ptracer */
1805         if (unlikely(trace))
1806             ptrace_event_pid(trace, pid);
1807 
1808         if (clone_flags & CLONE_VFORK) {
1809             if (!wait_for_vfork_done(p, &vfork))
1810                 ptrace_event_pid(PTRACE_EVENT_VFORK_DONE, pid);
1811         }
1812 
1813         put_pid(pid);
1814     } else {
1815         nr = PTR_ERR(p);
1816     }
1817     return nr;
```

After debugging scheduling problem, set the new process on run queue.
If all the progress was successful, return nr value
else, return error.