


Xilinx Zynq FPGA,TI DSP, MCU 기반의 프로그래밍 전문가 과정


날 짜 : 2018 . 4. 19

강사 – Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 – 정한별
hanbulkr@gmail.com

< _do_fork() 드라이빙 하기!>

먼저 두포크 함수를 찾는다.

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

```
1741 /* SIGCHLD, 0, 0, NULL, NULL, 0 */
1742 long _do_fork(unsigned long clone_flags,
1743           unsigned long stack_start,
1744           unsigned long stack_size,
1745           int __user *parent_tidptr,
1746           int __user *child_tidptr,
1747           unsigned long tls)
1748 {
1749     struct task_struct *p;
1750     int trace = 0;
1751     long nr;
1752
1753     /*
1754      * Determine whether and which event to report to ptracer.  When
1755      * called from kernel_thread or CLONE_UNTRACED is explicitly
1756      * requested, no event is reported; otherwise, report if the event
1757      * for the type of forking is enabled.
1758      */
1759
1760     /* CLONE_UNTRACED = 0x00800000 */
1761     if (!(clone_flags & CLONE_UNTRACED)) {
1762         /* CLONE_VFORK = 0x00004000 */
1763         if (clone_flags & CLONE_VFORK)
1764             trace = PTRACE_EVENT_VFORK; // =2
1765         /* CSIGNAL = 0x000000ff
1766            clone_flags 로 들어온것에 SIGCHLD 가 있는지 없는지에 따라
1767            trace 가 PTRACE_EVENT_FORK = 1  혹은 PTRACE_EVENT_CLONE = 3 이 됨 */
1768         else if ((clone_flags & CSIGNAL) != SIGCHLD)
1769             trace = PTRACE_EVENT_CLONE; // =3
1770         else
1771             trace = PTRACE_EVENT_FORK;  // =1
1772
1773         /* current 는 현재 구동중인 task */
1774         if (likely(!ptrace_event_enabled(current, trace)))
1775             trace = 0;
1776     }
1777
1778     /* clone_flags = SIGCHLD, stack_start = 0, stack_size = 0,
1779        child_tidptr = NULL, NULL, trace = 0, tls = 0
1780
1781        부모 프로세스의 task_struct 정보를 기반으로
1782        자식 프로세스를 만들었고 그것은 p 에 해당함 */
1783     p = copy_process(clone_flags, stack_start, stack_size,
1784              child_tidptr, NULL, trace, tls);
1785     /*
1786      * Do this prior waking up the new thread - the thread pointer
1787      * might get invalid after that point, if the thread exits quickly.
1788      */
1789     if (!IS_ERR(p)) {
1790         struct completion vfork;
1791         struct pid *pid;
1792
1793         trace_sched_process_fork(current, p);
1794
1795         pid = get_task_pid(p, PIDTYPE_PID);
1796         nr = pid_vnr(pid);
1797
1798         if (clone_flags & CLONE_PARENT_SETTID)
1799             put_user(nr, parent_tidptr);
1800
1801         if (clone_flags & CLONE_VFORK) {
1802             p->vfork_done = &vfork;
1803             init_completion(&vfork);
1804             get_task_struct(p);
1805         }
1806
                                                    1769,13-22    82%
```

1.처음 살펴볼 부분. clone_flags

```
    /* CLONE_UNTRACED = 0x00800000 */
    if (!(clone_flags & CLONE_UNTRACED)) {
        /* CLONE_VFORK = 0x00004000 */
        if (clone_flags & CLONE_VFORK)
            trace = PTRACE_EVENT_VFORK; // =2
        /* CSIGNAL = 0x000000ff
           clone_flags 로 들어온것에 SIGCHLD 가 있는지 없는지에 따라
           trace 가 PTRACE_EVENT_FORK = 1  혹은 PTRACE_EVENT_CLONE = 3 이 됨 */
        else if ((clone_flags & CSIGNAL) != SIGCHLD)
            trace = PTRACE_EVENT_CLONE; // =3
        else
            trace = PTRACE_EVENT_FORK;  // =1

        /* current 는 현재 구동중인 task */
        if (likely(!ptrace_event_enabled(current, trace)))
            trace = 0;
    }
```

if (!(clone_flags & CLONE_UNTRACED))

/* CLONE_UNTRACED = 0x00800000 */

```
Cscope tag: CLONE_UNTRACED
   #   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
```

(clone_flags & CLONE_VFORK)

/* CLONE_VFORK = 0x00004000 */

```
Cscope tag: CLONE_VFORK
   #   line  filename / context / line
   1     13  include/uapi/linux/sched.h <<CLONE_VFORK>>
             #define CLONE_VFORK 0x00004000
   2     13  /usr/include/linux/sched.h <<CLONE_VFORK>>
             #define CLONE_VFORK 0x00004000
```

trace = PTRACE_EVENT_VFORK; // =2

```
Cscope tag: PTRACE_EVENT_VFORK
   #   line  filename / context / line
   1     74  include/uapi/linux/ptrace.h <<PTRACE_EVENT_VFORK>>
             #define PTRACE_EVENT_VFORK 2
   2     74  /usr/include/linux/ptrace.h <<PTRACE_EVENT_VFORK>>
             #define PTRACE_EVENT_VFORK 2
```

else if ((clone_flags & CSIGNAL) != SIGCHLD)

        trace = PTRACE_EVENT_CLONE; // =3

else

        trace = PTRACE_EVENT_FORK; // =1

/* CSIGNAL = 0x000000ff

clone_flags 로 들어온것에 SIGCHLD 가 있는지 없는지에 따라

trace 가 PTRACE_EVENT_FORK = 1 혹은 PTRACE_EVENT_CLONE = 3 이 됨 */

```
Cscope tag: CSIGNAL
   #   line  filename / context / line
   1      7  include/uapi/linux/sched.h <<CSIGNAL>>
             #define CSIGNAL 0x000000ff
   2      7  /usr/include/linux/sched.h <<CSIGNAL>>
             #define CSIGNAL 0x000000ff
```

```
Cscope tag: SIGCHLD
   #   line  filename / context / line
   1     41  arch/alpha/include/uapi/asm/signal.h <<SIGCHLD>>
             #define SIGCHLD 20
```

```
Cscope tag: PTRACE_EVENT_CLONE
   #   line  filename / context / line
   1     75  include/uapi/linux/ptrace.h <<PTRACE_EVENT_CLONE>>
             #define PTRACE_EVENT_CLONE 3
   2     75  /usr/include/linux/ptrace.h <<PTRACE_EVENT_CLONE>>
             #define PTRACE_EVENT_CLONE 3
```

```
Cscope tag: PTRACE_EVENT_FORK
   #   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
```

/* current 는 현재 구동중인 task */
if (likely(!ptrace_event_enabled(current, trace)))
trace = 0;

```
        if (likely(!ptrace_event_enabled(current, trace)))
            trace = 0;
```

→ 함수로 진입

```
static inline bool ptrace_event_enabled(struct task_struct *task, int event)
{
    // PT_EVENT_FLAG(1) = 16
    return task->ptrace & PT_EVENT_FLAG(event);
}
```

#define PT_EVENT_FLAG(event) (1 << (PT_OPT_FLAG_SHIFT + (event)))

```
#define PT_EVENT_FLAG(event)    (1 << (PT_OPT_FLAG_SHIFT + (event)))
```

→ trace = event 라는 변수는 위에서 디버깅 상황에서 나타내야 할 행동 번호를 담아자기고 간다.

2. copy_process()

/* clone_flags = SIGCHLD, stack_start = 0, stack_size = 0,
child_tidptr = NULL, NULL, trace = 0, tls = 0

부모 프로세스의 task_struct 정보를 기반으로
자식 프로세스를 만들었고 그것은 p 에 해당함 */

p = copy_process(clone_flags, stack_start, stack_size,
child_tidptr, NULL, trace, tls);

```
 p = copy_process(clone_flags, stack_start, stack_size,
          child_tidptr, NULL, trace, tls);
```

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

```
1266 static struct task_struct *copy_process(unsigned long clone_flags,
1267                     unsigned long stack_start,
1268                     unsigned long stack_size,
1269                     int __user *child_tidptr,
1270                     struct pid *pid,
1271                     int trace,
1272                     unsigned long tls)
1273 {
1274     int retval;
1275     struct task_struct *p;
1276     /* CFS 스케줄러와 관련된 인덱스 값 */
1277     void *cgrp_ss_priv[CGROUP_CANFORK_COUNT] = {};
1278
1279     /* CLONE_NEWNS = 0x00020000
1280        CLONE_FS = 0x00000200 */
1281     if ((clone_flags & (CLONE_NEWNS|CLONE_FS)) == (CLONE_NEWNS|CLONE_FS))
1282         return ERR_PTR(-EINVAL);
1283
1284     /* CLONE_NEWUSER = 0x10000000 */
1285     if ((clone_flags & (CLONE_NEWUSER|CLONE_FS)) == (CLONE_NEWUSER|CLONE_FS))
1286         return ERR_PTR(-EINVAL);
1287
1288     /*
1289      * Thread groups must share signals as well, and detached threads
1290      * can only be started up within the thread group.
1291      */
1292
1293     /* CLONE_THREAD = 0x00010000
1294        CLONE_SIGHAND = 0x00000800 */
1295     if ((clone_flags & CLONE_THREAD) && !(clone_flags & CLONE_SIGHAND))
1296         return ERR_PTR(-EINVAL);
1297
1298     /*
1299      * Shared signal handlers imply shared VM. By way of the above,
1300      * thread groups also imply shared VM. Blocking this case allows
1301      * for various simplifications in other code.
1302      */
1303
1304     /* CLONE_VM = 0x00000100 */
1305     if ((clone_flags & CLONE_SIGHAND) && !(clone_flags & CLONE_VM))
1306         return ERR_PTR(-EINVAL);
1307
1308     /*
1309      * Siblings of global init remain as zombies on exit since they are
1310      * not reaped by their parent (swapper). To solve this and to avoid
1311      * multi-rooted process trees, prevent global and container-inits
1312      * from creating siblings.
1313      */
1314
1315     /* CLONE_PARENT = 0x00008000
1316        SIGNAL_UNKILLABLE = 0x00000040 */
1317     if ((clone_flags & CLONE_PARENT) &&
1318                 current->signal->flags & SIGNAL_UNKILLABLE)
1319         return ERR_PTR(-EINVAL);
1320
1321     /*
1322      * If the new process will be in a different pid or user namespace
1323      * do not allow it to share a thread group with the forking task.
1324      */
1325
1326     /* CLONE_NEWPID = 0x20000000 */
1327     if (clone_flags & CLONE_THREAD) {
1328         if ((clone_flags & (CLONE_NEWUSER | CLONE_NEWPID)) ||
1329             (task_active_pid_ns(current) !=
1330 ▉               current->nsproxy->pid_ns_for_children))
1331             return ERR_PTR(-EINVAL);
                                                          1330,2-8      59%
```

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

```
1333
1334     /* 프로세스에 대한 권한 체크 */
1335     retval = security_task_create(clone_flags);
1336     if (retval)
1337         goto fork_out;
1338
1339     retval = -ENOMEM;
1340     /* 현재 프로세스 복사하여 자식 프로세스를 생성함 */
1341     p = dup_task_struct(current);
1342     if (!p)
1343         goto fork_out;
1344
1345     ftrace_graph_init_task(p);
1346
1347     rt_mutex_init_task(p);
1348
1349 #ifdef CONFIG_PROVE_LOCKING
1350     DEBUG_LOCKS_WARN_ON(!p->hardirqs_enabled);
1351     DEBUG_LOCKS_WARN_ON(!p->softirqs_enabled);
1352 #endif
1353     retval = -EAGAIN;
1354     if (atomic_read(&p->real_cred->user->processes) >=
1355             task_rlimit(p, RLIMIT_NPROC)) {
1356         if (p->real_cred->user != INIT_USER &&
1357             !capable(CAP_SYS_RESOURCE) && !capable(CAP_SYS_ADMIN))
1358             goto bad_fork_free;
1359     }
1360     current->flags &= ~PF_NPROC_EXCEEDED;
1361
1362     /* 프로세스의 보안과 관련된 부분을 복사함 */
1363     retval = copy_creds(p, clone_flags);
1364     if (retval < 0)
1365         goto bad_fork_free;
1366
1367     /*
1368      * If multiple threads are within copy_process(), then this check
1369      * triggers too late. This doesn't hurt, the check is only there
1370      * to stop root fork bombs.
1371      */
1372     retval = -EAGAIN;
1373     if (nr_threads >= max_threads)
1374         goto bad_fork_cleanup_count;
1375
1376     /* Demand On Paging - Copy on Write */
1377     delayacct_tsk_init(p);  /* Must remain after dup_task_struct() */
1378     p->flags &= ~(PF_SUPERPRIV | PF_WQ_WORKER);
1379     p->flags |= PF_FORKNOEXEC;
1380     INIT_LIST_HEAD(&p->children);
1381     INIT_LIST_HEAD(&p->sibling);
1382     rcu_copy_process(p);
1383     p->vfork_done = NULL;
1384     spin_lock_init(&p->alloc_lock);
1385
1386     init_sigpending(&p->pending);
1387
1388     p->utime = p->stime = p->gtime = 0;
1389     p->utimescaled = p->stimescaled = 0;
1390     prev_cputime_init(&p->prev_cputime);
1391
1392 #ifdef CONFIG_VIRT_CPU_ACCOUNTING_GEN
1393     seqlock_init(&p->vtime_seqlock);
1394     p->vtime_snap = 0;
1395     p->vtime_snap_whence = VTIME_SLEEPING;
1396 #endif
1397
1398 #if defined(SPLIT_RSS_COUNTING)
                                                    1398,5        63%
```

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

```
1397
1398 #if defined(SPLIT_RSS_COUNTING)
1399     memset(&p->rss_stat, 0, sizeof(p->rss_stat));
1400 #endif
1401
1402     p->default_timer_slack_ns = current->timer_slack_ns;
1403
1404     task_io_accounting_init(&p->ioac);
1405     acct_clear_integrals(p);
1406
1407     posix_cpu_timers_init(p);
1408
1409     p->start_time = ktime_get_ns();
1410     p->real_start_time = ktime_get_boot_ns();
1411     p->io_context = NULL;
1412     p->audit_context = NULL;
1413     threadgroup_change_begin(current);
1414     cgroup_fork(p);
1415 #ifdef CONFIG_NUMA
1416     p->mempolicy = mpol_dup(p->mempolicy);
1417     if (IS_ERR(p->mempolicy)) {
1418         retval = PTR_ERR(p->mempolicy);
1419         p->mempolicy = NULL;
1420         goto bad_fork_cleanup_threadgroup_lock;
1421     }
1422 #endif
1423 #ifdef CONFIG_CPUSETS
1424     p->cpuset_mem_spread_rotor = NUMA_NO_NODE;
1425     p->cpuset_slab_spread_rotor = NUMA_NO_NODE;
1426     seqcount_init(&p->mems_allowed_seq);
1427 #endif
1428 #ifdef CONFIG_TRACE_IRQFLAGS
1429     p->irq_events = 0;
1430     p->hardirqs_enabled = 0;
1431     p->hardirq_enable_ip = 0;
1432     p->hardirq_enable_event = 0;
1433     p->hardirq_disable_ip = _THIS_IP_;
1434     p->hardirq_disable_event = 0;
1435     p->softirqs_enabled = 1;
1436     p->softirq_enable_ip = _THIS_IP_;
1437     p->softirq_enable_event = 0;
1438     p->softirq_disable_ip = 0;
1439     p->softirq_disable_event = 0;
1440     p->hardirq_context = 0;
1441     p->softirq_context = 0;
1442 #endif
1443
1444     p->pagefault_disabled = 0;
1445
1446 #ifdef CONFIG_LOCKDEP
1447     p->lockdep_depth = 0; /* no locks held yet */
1448     p->curr_chain_key = 0;
1449     p->lockdep_recursion = 0;
1450 #endif
1451
1452 #ifdef CONFIG_DEBUG_MUTEXES
1453     p->blocked_on = NULL; /* not blocked yet */
1454 #endif
1455 #ifdef CONFIG_BCACHE
1456     p->sequential_io    = 0;
1457     p->sequential_io_avg    = 0;
1458 #endif
1459
1460     /* Perform scheduler related setup. Assign this task to a CPU. */
1461
1462     /* 실제 CPU 에서 구동될 수 있도록 스케쥴러에 배치함 */
                                                    1462,2-5      66%
```

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

```
1462     /* 실제 CPU 에서 구동될 수 있도록 스케쥴러에 배치함 */
1463     retval = sched_fork(clone_flags, p);
1464     if (retval)
1465         goto bad_fork_cleanup_policy;
1466
1467     retval = perf_event_init_task(p);
1468     if (retval)
1469         goto bad_fork_cleanup_policy;
1470     retval = audit_alloc(p);
1471     if (retval)
1472         goto bad_fork_cleanup_perf;
1473     /* copy all the process information */
1474     shm_init_task(p);
1475     retval = copy_semundo(clone_flags, p);
1476     if (retval)
1477         goto bad_fork_cleanup_audit;
1478
1479     /* 기본적으로 열어야 하는 파일들 복사 */
1480     retval = copy_files(clone_flags, p);
1481     if (retval)
1482         goto bad_fork_cleanup_semundo;
1483     retval = copy_fs(clone_flags, p);
1484     if (retval)
1485         goto bad_fork_cleanup_files;
1486     retval = copy_sighand(clone_flags, p);
1487     if (retval)
1488         goto bad_fork_cleanup_fs;
1489     retval = copy_signal(clone_flags, p);
1490     if (retval)
1491         goto bad_fork_cleanup_sighand;
1492     /* task_struct->mm_struct 도 복사 */
1493     retval = copy_mm(clone_flags, p);
1494     if (retval)
1495         goto bad_fork_cleanup_signal;
1496     retval = copy_namespaces(clone_flags, p);
1497     if (retval)
1498         goto bad_fork_cleanup_mm;
1499     retval = copy_io(clone_flags, p);
1500     if (retval)
1501         goto bad_fork_cleanup_namespaces;
1502     retval = copy_thread_tls(clone_flags, stack_start, stack_size, p, tls);
1503     if (retval)
1504         goto bad_fork_cleanup_io;
1505
1506     if (pid != &init_struct_pid) {
1507         /* Task 에게 pid 값 할당 */
1508         pid = alloc_pid(p->nsproxy->pid_ns_for_children);
1509         if (IS_ERR(pid)) {
1510             retval = PTR_ERR(pid);
1511             goto bad_fork_cleanup_io;
1512         }
1513     }
1514
1515     p->set_child_tid = (clone_flags & CLONE_CHILD_SETTID) ? child_tidptr : NULL;
1516     /*
1517      * Clear TID on mm_release()?
1518      */
1519     p->clear_child_tid = (clone_flags & CLONE_CHILD_CLEARTID) ? child_tidptr : NULL;
1520 #ifdef CONFIG_BLOCK
1521     p->plug = NULL;
1522 #endif
1523 #ifdef CONFIG_FUTEX
1524     p->robust_list = NULL;
1525 #ifdef CONFIG_COMPAT
1526     p->compat_robust_list = NULL;
1527 #endif
                                                                    1462,2-5      69%
```

– 엄청 많다… 여기 까지 함.

2-1. clone_flags 애러체크를 한다.

if ((clone_flags & (CLONE_NEWNS|CLONE_FS)) == (CLONE_NEWNS|CLONE_FS))

/* CLONE_NEWNS = 0x00020000

CLONE_FS = 0x00000200 */

```
Cscope tag: CLONE_NEWNS
   #   line  filename / context / line
   1     16  include/uapi/linux/sched.h <<CLONE_NEWNS>>
             #define CLONE_NEWNS 0x00020000
```

```
Cscope tag: CLONE_FS
   #   line  filename / context / line
   1      9  include/uapi/linux/sched.h <<CLONE_FS>>
             #define CLONE_FS 0x00000200
```

if ((clone_flags & (CLONE_NEWUSER|CLONE_FS)) == (CLONE_NEWUSER|CLONE_FS))

/* CLONE_NEWUSER = 0x10000000 */

```
1     28  include/uapi/linux/sched.h <<CLONE_NEWUSER>>
          #define CLONE_NEWUSER 0x10000000
```

if ((clone_flags & CLONE_THREAD) && !(clone_flags & CLONE_SIGHAND))

/* CLONE_THREAD = 0x00010000

CLONE_SIGHAND = 0x00000800 */

```
1     15  include/uapi/linux/sched.h <<CLONE_THREAD>>
          #define CLONE_THREAD 0x00010000
```

```
1     11  include/uapi/linux/sched.h <<CLONE_SIGHAND>>
          #define CLONE_SIGHAND 0x00000800
```

if ((clone_flags & CLONE_SIGHAND) && !(clone_flags & CLONE_VM))

/* CLONE_VM = 0x00000100 */

```
1      8  include/uapi/linux/sched.h <<CLONE_VM>>
          #define CLONE_VM 0x00000100
```

if ((clone_flags & CLONE_PARENT) && current->signal->flags & SIGNAL_UNKILLABLE)

/* CLONE_PARENT = 0x00008000

SIGNAL_UNKILLABLE = 0x00000040 */

```
1     14  include/uapi/linux/sched.h <<CLONE_PARENT>>
          #define CLONE_PARENT 0x00008000
```

```
#define SIGNAL_UNKILLABLE   0x00000040 /* for init: ignore fatal signals */
```

if (clone_flags & CLONE_THREAD) {

if ((clone_flags & (CLONE_NEWUSER | CLONE_NEWPID)) || (task_active_pid_ns(current) !=

current->nsproxy->pid_ns_for_children))

/* CLONE_NEWPID = 0x20000000 */

```
1     29  include/uapi/linux/sched.h <<CLONE_NEWPID>>
          #define CLONE_NEWPID 0x20000000
```

2-1. 프로세스 권한 체크 security_task_create()

/* 프로세스에 대한 권한 체크 */

retval = security_task_create(clone_flags);

```
retval = security_task_create(clone_flags);
if (retval)
    goto fork_out;
```

int security_task_create(unsigned long clone_flags)

/* clone_flags = SIGCHLD
   list_head 형태로 구조체 내부 정보는 아래와 같음
   next = security_hook_heads.task_create,
   prev = security_hook_heads.task_create */

```
int security_task_create(unsigned long clone_flags)
{
    /* clone_flags = SIGCHLD
    list_head 형태로 구조체 내부 정보는 아래와 같음
    next = security_hook_heads.task_create,
    prev = security_hook_heads.task_create */
    return call_int_hook(task_create, 0, clone_flags);
}
```

//

task_create → security_hook_heads → list_head task_create 로 있음 → list_head → next, prev

```
struct security_hook_heads {
```

```
struct list_head mmap_addr;
struct list_head mmap_file;
struct list_head file_mprotect;
struct list_head file_lock;
struct list_head file_fcntl;
struct list_head file_set_fowner;
struct list_head file_send_sigiotask;
struct list_head file_receive;
struct list_head file_open;
struct list_head task_create;
struct list_head task_free;
struct list_head cred_alloc_blank;
struct list_head cred_free;
struct list_head cred_prepare;
struct list_head cred_transfer;
struct list_head kernel_act_as;
struct list_head kernel_create_files_as;
struct list_head kernel_fw_from_file;
```

```
struct list_head {
    struct list_head *next, *prev;
};
```

//

return call_int_hook(task_create, 0, clone_flags);

#define call_int_hook(FUNC, IRC, ...)

```
#define call_int_hook(FUNC, IRC, ...) ({            \
    int RC = IRC;                                   \
    do {                                            \
        struct security_hook_list *P;               \
                                                    \
        list_for_each_entry(P, &security_hook_heads.FUNC, list) { \
            RC = P->hook.FUNC(__VA_ARGS__);         \
            if (RC != 0)                            \
                break;                              \
        }                                           \
    } while (0);                                    \
    RC;                                             \
})
```

*struct security_hook_list *P;*

```
struct security_hook_list {
    struct list_head        list;
    struct list_head        *head;
    union security_list_options hook;
};
```

함수로 들어올 때 task_create 를 FUNC 자리에 가지고 들어감.
FUNC = task_create
security_hook_heads.task_create

#define list_for_each_entry(pos, head, member)
/ head = &security_hook_heads.task_create → 주소를 찾을 때까지 반복을 한다임.*
*typeof(*pos) = struct security_hook_list */*

```
#define list_for_each_entry(pos, head, member)              \
    for (pos = list_first_entry(head, typeof(*pos), member);    \
         &pos->member != (head);                    \
         pos = list_next_entry(pos, member))
```

#define list_first_entry(ptr, type, member)

```
#define list_first_entry(ptr, type, member) \
    list_entry((ptr)->next, type, member)
```

#define list_entry(ptr, type, member)

```
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
```

*#define container_of(ptr, type, member) ({ *
*const typeof(((type *)0)->member) *__mptr = (ptr); *
*(type *)((char *)__mptr - offsetof(type,member));})*

```
#define container_of(ptr, type, member) ({          \
    const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
    (type *)( (char *)__mptr - offsetof(type,member) );})
```

/* ptr = &security_hook_heads.task_create
type = struct security_hook_list
typeof(((type *)0)->member) = struct list_head */

/* ptr = entry
type = struct page
member = lru

const list_head *__mptr = entry
(struct page *)((char *)__mptr - page 구조체 내에서 lru 멤버의 옵셋) */
→> 여기서 ((type) *)0) 은 첫번째 원소를 뜻한다.

2-2. task_create 이 어디있는가?

- 태그로는 찾기 힘들기 때문에 grep 으로 찾는다.

```
hanbulkr@onestar-com: ~/kernel/linux-4.4
hanbulkr@onestar-com:~/kernel/linux-4.4$ grep -rn task_create
kernel/fork.c:1335:     retval = security_task_create(clone_flags);
^C
hanbulkr@onestar-com:~/kernel/linux-4.4$ grep -rn task_create ./
./kernel/fork.c:1335:   retval = security_task_create(clone_flags);
./tags:2616588:security_task_create     include/linux/security.h        /^static
 inline int security_task_create(unsigned long clone_flags)$/;" f
./tags:2616589:security_task_create     security/security.c     /^int security_t
ask_create(unsigned long clone_flags)$/;"       f
./tags:2618134:selinux_task_create      security/selinux/hooks.c        /^static
 int selinux_task_create(unsigned long clone_flags)$/;" f       file:
./tags:2723533:task_create      include/linux/lsm_hooks.h       /^      int (*ta
sk_create)(unsigned long clone_flags);$/;"      m       union:security_list_opti
ons
./tags:2723534:task_create      include/linux/lsm_hooks.h       /^      struct l
ist_head task_create;$/;"       m       struct:security_hook_heads      typeref:
struct:security_hook_heads::list_head
./security/selinux/hooks.c:3547:static int selinux_task_create(unsigned long clo
ne_flags)
./security/selinux/hooks.c:5955:        LSM_HOOK_INIT(task_create, selinux_task_
create),
./security/security.c:117:/* security_hook_heads.task_create
./security/security.c:125:FUNC = task_create
./security/security.c:858:int security_task_create(unsigned long clone_flags)
./security/security.c:862:      next = security_hook_heads.task_create,
./security/security.c:863:      prev = security_hook_heads.task_create */
./security/security.c:864:      return call_int_hook(task_create, 0, clone_flags
);
./security/security.c:1691:     .task_create =  LIST_HEAD_INIT(security_hook_hea
ds.task_create),
./include/linux/security.h:293:int security_task_create(unsigned long clone_flag
s);
./include/linux/security.h:813:static inline int security_task_create(unsigned l
ong clone_flags)
./include/linux/kernel.h:813:/* ptr = &security_hook_heads.task_create
./include/linux/list.h:352:/* ptr = &security_hook_heads.task_create
./include/linux/list.h:370:/* ptr = &security_hook_heads.task_create
./include/linux/list.h:457:/* head = &security_hook_heads.task_create
Binary file ./include/linux/.kernel.h.swp matches
./include/linux/lsm_hooks.h:505: * @task_create:
./include/linux/lsm_hooks.h:1444:       int (*task_create)(unsigned long clone_f
lags);
./include/linux/lsm_hooks.h:1705:       struct list_head task_create;
hanbulkr@onestar-com:~/kernel/linux-4.4$
```

security/selinux/hooks.cstatic:3547: int selinux_task_create(unsigned long clone_flags)
./security/selinux/hooks.c:5955:   LSM_HOOK_INIT(task_create, selinux_task_create),

```
static int selinux_task_create(unsigned long clone_flags)
{
    /* TODO:
       PROCESS__FORK ??? */
    return current_has_perm(current, PROCESS__FORK);
}
```

```
LSM_HOOK_INIT(task_create, selinux_task_create),
```

grep -rn “selinux_hooks” ./

```
hanbulkr@onestar-com:~/kernel/linux-4.4$ grep -rn "selinux_hooks" ./
./tags:2617949:selinux_hooks    security/selinux/hooks.c        /^static struct sec
urity_hook_list selinux_hooks[] = {$/;" v       typeref:struct:security_hook_list f
ile:
./security/selinux/hooks.c:5876:static struct security_hook_list selinux_hooks[] =
{
./security/selinux/hooks.c:6114:        security_add_hooks(selinux_hooks, ARRAY_SIZ
E(selinux_hooks));
./security/selinux/hooks.c:6237:        security_delete_hooks(selinux_hooks, ARRAY_
SIZE(selinux_hooks));
./cscope.out:73114013:(selinux_hooks));
./cscope.out:73114377:(selinux_hooks));
```

./security/selinux/hooks.c:6114: security_add_hooks(selinux_hooks, ARRAY_SIZE(selinux_hooks));

→ 여기 안에 security_add_hooks 있었다.

```
static __init int selinux_init(void)
```

→ 찾는다.grep -rn "selinux_init" ./

```
hanbulkr@onestar-com:~/kernel/linux-4.4$ grep -rn "selinux_init" ./
./tags:2617954:selinux_init     security/selinux/hooks.c        /^security_initcall(selinux_init);$/;" v
./tags:2617955:selinux_init     security/selinux/hooks.c        /^static __init int selinux_init(void)$/;"  f
file:
./security/selinux/hooks.c:6087:static __init int selinux_init(void)
./security/selinux/hooks.c:6143:security_initcall(selinux_init);
```

./security/selinux/hooks.c:6143:security_initcall(selinux_init);

```
#define security_initcall(fn) \
    static initcall_t __initcall_##fn \
    __used __section(.security_initcall.init) = fn
```

→ 찾는다.grep -rn "security_initcall.init" ./

```
hanbulkr@onestar-com:~/kernel/linux-4.4$ grep -rn "security_initcall" ./
./include/asm-generic/vmlinux.lds.h:416:                VMLINUX_SYMBOL(__security_initcall_start) = .;  \
./include/asm-generic/vmlinux.lds.h:417:                *(.security_initcall.init)                      \
./include/asm-generic/vmlinux.lds.h:418:                VMLINUX_SYMBOL(__security_initcall_end) = .;    \
./include/asm-generic/vmlinux.lds.h:679:                VMLINUX_SYMBOL(__security_initcall_start) = .;  \
./include/asm-generic/vmlinux.lds.h:680:                *(.security_initcall.init)                      \
./include/asm-generic/vmlinux.lds.h:681:                VMLINUX_SYMBOL(__security_initcall_end) = .;
```

./include/asm-generic/vmlinux.lds.h:415: .security_initcall.init : AT(ADDR(.security_initcall.init) - LOAD_OFFSET) { \

```
#define SECURITY_INIT                           \
    .security_initcall.init : AT(ADDR(.security_initcall.init) - LOAD_OFFSET) { \
        VMLINUX_SYMBOL(__security_initcall_start) = .;      \
        *(.security_initcall.init)              \
        VMLINUX_SYMBOL(__security_initcall_end) = .;        \
    }
```

-이게 아님.

```
#define SECURITY_INITCALL                       \
        VMLINUX_SYMBOL(__security_initcall_start) = .;      \
        *(.security_initcall.init)              \
        VMLINUX_SYMBOL(__security_initcall_end) = .;
```

→ 이거다. SECURITY_INITCALL

SECURITY_INITCALL

```
hanbulkr@onestar-com:~/kernel/linux-4.4$ grep -rn "SECURITY_INITCALL" ./
./arch/blackfin/kernel/vmlinux.lds.S:155:                SECURITY_INITCALL
./arch/arc/kernel/vmlinux.lds.S:74:                SECURITY_INITCALL
./arch/xtensa/kernel/vmlinux.lds.S:179:    SECURITY_INITCALL
./arch/arm/kernel/vmlinux.lds.S:217:                SECURITY_INITCALL
./arch/arm64/kernel/vmlinux.lds.S:137:                SECURITY_INITCALL
./tags:1179586:SECURITY_INITCALL        include/asm-generic/vmlinux.lds.h       678;"   d
./include/asm-generic/vmlinux.lds.h:678:#define SECURITY_INITCALL                                \
./include/asm-generic/vmlinux.lds.h:830:                SECURITY_INITCALL
```

```
#define INIT_DATA_SECTION(initsetup_align)                \
    .init.data : AT(ADDR(.init.data) - LOAD_OFFSET) {       \
        INIT_DATA                       \
        INIT_SETUP(initsetup_align)             \
        INIT_CALLS                      \
        CON_INITCALL                        \
        SECURITY_INITCALL                   \
        INIT_RAM_FS                     \
    }
```

…….???? 여기서 selinux_init 으로 갔음.(이해가 잘 안됨...)

```
static __init int selinux_init(void)
{
    if (!security_module_enable("selinux")) {
        selinux_enabled = 0;
        return 0;
    }

    if (!selinux_enabled) {
        printk(KERN_INFO "SELinux:  Disabled at boot.\n");
        return 0;
    }

    printk(KERN_INFO "SELinux:  Initializing.\n");

    /* Set the security state for the initial task. */
    cred_init_security();

    default_noexec = !(VM_DATA_DEFAULT_FLAGS & VM_EXEC);

    sel_inode_cache = kmem_cache_create("selinux_inode_security",
                          sizeof(struct inode_security_struct),
                          0, SLAB_PANIC, NULL);
    file_security_cache = kmem_cache_create("selinux_file_security",
                          sizeof(struct file_security_struct),
                          0, SLAB_PANIC, NULL);
    avc_init();

    security_add_hooks(selinux_hooks, ARRAY_SIZE(selinux_hooks));

    if (avc_add_callback(selinux_netcache_avc_callback, AVC_CALLBACK_RESET))
        panic("SELinux: Unable to register AVC netcache callback\n");

    if (selinux_enforcing)
        printk(KERN_DEBUG "SELinux:  Starting in enforcing mode\n");
    else
        printk(KERN_DEBUG "SELinux:  Starting in permissive mode\n");

    return 0;
}
```

결국 task_create 에서 selinux_hooks → static __init int selinux_init(void)를 찾아온 과정...

2-3. selinux_init 을 파헤쳐 보자.

/* Set the security state for the initial task. */
cred_init_security();
* initialise the security for the init task 라는 설명으로 보아 보안쪽 시작 task 에 첫부분이다
→ 기본 메모리 공간들을 초기화 하고 있다.

```
static void cred_init_security(void)
{
    struct cred *cred = (struct cred *) current->real_cred;
    struct task_security_struct *tsec;

    tsec = kzalloc(sizeof(struct task_security_struct), GFP_KERNEL);
    if (!tsec)
        panic("SELinux:  Failed to initialize initial task.\n");

    tsec->osid = tsec->sid = SECINITSID_KERNEL;
    cred->security = tsec;
}
```

* kzalloc - allocate memory. The memory is set to zero. 라고 설명됨 동적 메모리가 0 으로 초기화.

```
static inline void *kzalloc(size_t size, gfp_t flags)
{
    return kmalloc(size, flags | __GFP_ZERO);
}
```

#define ___GFP_ZERO   0x8000u
→ 빠져나옴.

default_noexec = !(VM_DATA_DEFAULT_FLAGS & VM_EXEC);
VM_DATA_DEFAULT_FLAGS

```
#define VM_DATA_DEFAULT_FLAGS \
    (((current->personality & READ_IMPLIES_EXEC) ? VM_EXEC : 0 ) | \
     VM_READ | VM_WRITE | VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC)
```

```
#define personality(pers)   (pers & PER_MASK)
```

```
24  24728  crypto/testmgr.h <<pers>>
           .pers = NULL,
```

personality 는 결국 pers 가 NULL 이니 숏컷으로 NULL 이 나옴.
결국,
default_noexec = NULL;

sel_inode_cache = kmem_cache_create("selinux_inode_security",
sizeof(struct inode_security_struct),
0, SLAB_PANIC, NULL);

```
struct inode_security_struct {
    struct inode *inode;    /* back pointer to inode object */
    union {
        struct list_head list;  /* list of inode_security_struct */
        struct rcu_head rcu;    /* for freeing the inode_security_struct */
    };
    u32 task_sid;       /* SID of creating task */
    u32 sid;        /* SID of this object */
    u16 sclass;     /* security class of this object */
    unsigned char initialized;  /* initialization flag */
    struct mutex lock;
};
```

*kmem_cache_create(const char *name, size_t size, size_t align,*
*unsigned long flags, void (*ctor)(void *))*

slab_common.c (~/kernel/linux-4.4/mm) - VIM

```
382 */
383 struct kmem_cache *
384 kmem_cache_create(const char *name, size_t size, size_t align,
385           unsigned long flags, void (*ctor)(void *))
386 {
387     struct kmem_cache *s = NULL;
388     const char *cache_name;
389     int err;
390
391     get_online_cpus();
392     get_online_mems();
393     memcg_get_cache_ids();
394
395     mutex_lock(&slab_mutex);
396
397     err = kmem_cache_sanity_check(name, size);
398     if (err) {
399         goto out_unlock;
400     }
401
402     /*
403      * Some allocators will constraint the set of valid flags to a subset
404      * of all flags. We expect them to define CACHE_CREATE_MASK in this
405      * case, and we'll just provide them with a sanitized version of the
406      * passed flags.
407      */
408     flags &= CACHE_CREATE_MASK;
409
410     s = __kmem_cache_alias(name, size, align, flags, ctor);
411     if (s)
412         goto out_unlock;
413
414     cache_name = kstrdup_const(name, GFP_KERNEL);
415     if (!cache_name) {
416         err = -ENOMEM;
417         goto out_unlock;
418     }
419
420     s = create_cache(cache_name, size, size,
421              calculate_alignment(flags, align, size),
422              flags, ctor, NULL, NULL);
423     if (IS_ERR(s)) {
424         err = PTR_ERR(s);
425         kfree_const(cache_name);
426     }
427
428 out_unlock:
429     mutex_unlock(&slab_mutex);
430
431     memcg_put_cache_ids();
432     put_online_mems();
433     put_online_cpus();
434
435     if (err) {
436         if (flags & SLAB_PANIC)
437             panic("kmem_cache_create: Failed to create slab '%s'. Error %d\n",
438                 name, err);
439         else {
440             printk(KERN_WARNING "kmem_cache_create(%s) failed with error %d",
441                 name, err);
442             dump_stack();
443         }
444         return NULL;
445     }
446     return s;
447 }
                                                    447,1         31%
```

get_online_cpus() // 아무것도 안함.

```
  1    247  include/linux/cpu.h <<get_online_cpus>>
            #define get_online_cpus() do { } while (0)
```

get_online_mems(void) // 아무것도 안함.

```
  1    235  include/linux/memory_hotplug.h <<get_online_mems>>
            static inline void get_online_mems(void ) {}
```

get_online_mems(void) // 이것도 아무것도 안하는 함수.

```
static inline void memcg_get_cache_ids(void)
{
}
```

static int kmem_cache_sanity_check(const char *name, size_t size)
// 캐시 메모리가 제대로 들어가는지 확인해주는 함수 이다.

```
static int kmem_cache_sanity_check(const char *name, size_t size)
{
    struct kmem_cache *s = NULL;

    if (!name || in_interrupt() || size < sizeof(void *) ||
        size > KMALLOC_MAX_SIZE) {
        pr_err("kmem_cache_create(%s) integrity check failed\n", name);
        return -EINVAL;
    }

    list_for_each_entry(s, &slab_caches, list) {
        char tmp;
        int res;

        /*
         * This happens when the module gets unloaded and doesn't
         * destroy its slab cache and no-one else reuses the vmalloc
         * area of the module.  Print a warning.
         */
        res = probe_kernel_address(s->name, tmp);
        if (res) {
            pr_err("Slab cache with size %d has lost its name\n",
                   s->object_size);
            continue;
        }
    }

    WARN_ON(strchr(name, ' ')); /* It confuses parsers */
    return 0;
}
```

kmem_cache * __kmem_cache_alias(const char *name, size_t size, size_t align,
unsigned long flags, void (*ctor)(void *))
// 케시 메모리에 할당된 크기를 최적화 해주는 함수?

```
struct kmem_cache *
__kmem_cache_alias(const char *name, size_t size, size_t align,
           unsigned long flags, void (*ctor)(void *))
{
    struct kmem_cache *cachep;

    cachep = find_mergeable(size, align, flags, name, ctor);
    if (cachep) {
        cachep->refcount++;

        /*
         * Adjust the object sizes so that we clear
         * the complete object on kzalloc.
         */
        cachep->object_size = max_t(int, cachep->object_size, size);
    }
    return cachep;
}
```

kstrdup_const(name, GFP_KERNEL);

```
static void *__ac_get_obj(struct kmem_cache *cachep, struct array_cache *ac,
                          gfp_t flags, bool force_refill)
{
    int i;
    void *objp = ac->entry[--ac->avail];
```

타고타고타고 들어가면 결국이게 나온다.

*static struct kmem_cache *create_cache(const char *name,*
size_t object_size, size_t size, size_t align,
*unsigned long flags, void (*ctor)(void *),*
*struct mem_cgroup *memcg, struct kmem_cache *root_cache)*

// 케시 공간을 만들어 주고 내부의 것들을 0 으로 초기화 해주는 함수.

```
static struct kmem_cache *create_cache(const char *name,
        size_t object_size, size_t size, size_t align,
        unsigned long flags, void (*ctor)(void *),
        struct mem_cgroup *memcg, struct kmem_cache *root_cache)
{
    struct kmem_cache *s;
    int err;

    err = -ENOMEM;
    s = kmem_cache_zalloc(kmem_cache, GFP_KERNEL);
    if (!s)
        goto out;

    s->name = name;
    s->object_size = object_size;
    s->size = size;
    s->align = align;
    s->ctor = ctor;

    err = init_memcg_params(s, memcg, root_cache);
    if (err)
        goto out_free_cache;

    err = __kmem_cache_create(s, flags);
    if (err)
        goto out_free_cache;

    s->refcount = 1;
    list_add(&s->list, &slab_caches);
out:
    if (err)
        return ERR_PTR(err);
    return s;

out_free_cache:
    destroy_memcg_params(s);
    kmem_cache_free(kmem_cache, s);
    goto out;
}
```

kmem_cache_zalloc(kmem_cache, GFP_KERNEL);

// 케시를 0 으로 초기화 해준다.

```
 s = kmem_cache_zalloc(kmem_cache, GFP_KERNEL);
 if (!s)
     goto out;

 s->name = name;
 s->object_size = object_size;
 s->size = size;
 s->align = align;
 s->ctor = ctor;
```

- **get_online_cpus();**
    - CONFIG_HOTPLUG_CPU 커널 옵션을 사용하는 경우에만 cpu_hotplug.refcount 를 증가시켜 cpu 를 분리하는 경우 동기화(지연)시킬 목적으로 설정한다.
- **get_online_mems();**

- CONFIG_MEMORY_HOTPLUG 커널 옵션을 사용하는 경우에만 mem_hotplug.refcount 를 증가시켜 memory 를 분리하는 경우 동기화(지연)시킬 목적으로 설정한다.

- **memcg_get_cache_ids();**
  - MEMCG_KMEM 커널 옵션을 사용하여 Slab(Slub) 커널 메모리 사용량을 제어하고자 할 목적으로 read 세마포어 락을 사용한다.

```
get_online_cpus();
get_online_mems();
memcg_get_cache_ids();
```

케시,처음 케시 공간 만들 준비할때, 인듯하다.

```
memcg_put_cache_ids();
put_online_mems();
put_online_cpus();
```

케시, 메모리 설정 다하고 내보낼 때, 인듯하다.

void __init avc_init(void)
// avc(access vector cache) :주체 (예 : 응용 프로그램)가 객체 (예 : 파일)에 액세스하려고할 때, 커널의 정책 집행 서버가 주체 , 객체 권한이 캐시

```
void __init avc_init(void)
{
    int i;

    for (i = 0; i < AVC_CACHE_SLOTS; i++) {
        INIT_HLIST_HEAD(&avc_cache.slots[i]);
        spin_lock_init(&avc_cache.slots_lock[i]);
    }
    atomic_set(&avc_cache.active_nodes, 0);
    atomic_set(&avc_cache.lru_hint, 0);

    avc_node_cachep = kmem_cache_create("avc_node", sizeof(struct avc_node),
                    0, SLAB_PANIC, NULL);
    avc_xperms_cachep = kmem_cache_create("avc_xperms_node",
                    sizeof(struct avc_xperms_node),
                    0, SLAB_PANIC, NULL);
    avc_xperms_decision_cachep = kmem_cache_create(
                    "avc_xperms_decision_node",
                    sizeof(struct avc_xperms_decision_node),
                    0, SLAB_PANIC, NULL);
    avc_xperms_data_cachep = kmem_cache_create("avc_xperms_data",
                    sizeof(struct extended_perms_data),
                    0, SLAB_PANIC, NULL);

    audit_log(current->audit_context, GFP_KERNEL, AUDIT_KERNEL, "AVC INITIALIZED\n");
}
```

한마디로 캐시를 할수 있게 객체 권한을 줄지 말지 해주는 녀석인 듯 하다.
selinux_init 끝 근데 끝네고 보니 이게 아니라 selinux_task_create 인듯 하다.

2-4. selinux_task_create

중간에 한번 찾았다가 지나갔지만 사실 이곳에 정보가 있다.

```
static int selinux_task_create(unsigned long clone_flags)
{
    /* TODO:
       PROCESS__FORK ??? */
    return current_has_perm(current, PROCESS__FORK);
}
```

current_has_perm(current, PROCESS__FORK);

```
static int current_has_perm(const struct task_struct *tsk,
                u32 perms)
{
    u32 sid, tsid;

    sid = current_sid();
    tsid = task_sid(tsk);

    /* TODO:
       SECCLASS_PROCESS = ??? */
    return avc_has_perm(sid, tsid, SECCLASS_PROCESS, perms, NULL);
}
```

current_sid()

```
static inline u32 current_sid(void)
{
#if 0   // current_security()
    task_struct->cred->security

    struct task_security_struct {
    u32 osid;       /* SID prior to last execve */
    u32 sid;        /* current SID */
    u32 exec_sid;       /* exec SID */
    u32 create_sid;     /* fscreate SID */
    u32 keycreate_sid;  /* keycreate SID */
    u32 sockcreate_sid; /* fscreate SID */
    };
#endif
    const struct task_security_struct *tsec = current_security();

    return tsec->sid;
}
```

RCU History

- RCU 는 읽기 동작에서 블러킹 되지 않는 read/write 동기화 메커니즘

장/단점

RCU 는 read-side overhead 를 최소화하는데 목적이 있기 때문에 동기화 로직이 읽기 동작에 더 많은 비율로 사용되는 경우에만 사용한다. 수정 동작이 10%이상인 경우 오히려 성능이 떨어지므로 RCU 대신 다른 동기화 기법을 선택해야 한다.

위에서 RCT 기법에 의해서 현재 태스크의 읽기정보들을 읽어온다고 보면 될거 같다.

```
#define current_security()  (current_cred_xxx(security))
```

```
#define current_cred_xxx(xxx)           \
({                      \
    current_cred()->xxx;            \
})
```

```
 * current_cred - Access the current task's subjective credentials
 *
 * Access the subjective credentials of the current task.  RCU-safe,
 * since nobody else can modify it.
 */
#define current_cred() \
    rcu_dereference_protected(current->cred, 1)
```

계속 cred 구조체가 나오고 있다. cred 는 자신의 보안 정보를 나타내는 녀석이다.

```
static inline u32 task_sid(const struct task_struct *task)
{
    u32 sid;

    rcu_read_lock();
    sid = cred_sid(__task_cred(task));
    rcu_read_unlock();
    return sid;
}
```

avc_has_perm(sid, tsid, SECCLASS_PROCESS, perms, NULL); // 밑에는 설명

```
    /* TODO:
       SECCLASS_PROCESS = ??? */
    return avc_has_perm(sid, tsid, SECCLASS_PROCESS, perms, NULL);
```

```
int avc_has_perm(u32 ssid, u32 tsid, u16 tclass,
         u32 requested, struct common_audit_data *auditdata)
{
    /* struct av_decision {
        u32 allowed;
        u32 auditallow;
        u32 auditdeny;
        u32 seqno;
        u32 flags;
    }; */
    struct av_decision avd;
    int rc, rc2;

    rc = avc_has_perm_noaudit(ssid, tsid, tclass, requested, 0, &avd);

    rc2 = avc_audit(ssid, tsid, tclass, requested, &avd, rc, auditdata, 0);
    if (rc2)
        return rc2;
    return rc;
}
```

```
/**
 * avc_has_perm - Check permissions and perform any appropriate auditing.
 * @ssid: source security identifier
 * @tsid: target security identifier
 * @tclass: target security class
 * @requested: requested permissions, interpreted based on @tclass
 * @auditdata: auxiliary audit data
 *
 * Check the AVC to determine whether the @requested permissions are granted
 * for the SID pair (@ssid, @tsid), interpreting the permissions
 * based on @tclass, and call the security server on a cache miss to obtain
 * a new decision and add it to the cache.  Audit the granting or denial of
 * permissions in accordance with the policy.  Return %0 if all @requested
 * permissions are granted, -%EACCES if any permissions are denied, or
 * another -errno upon other errors.
 */
```

→ 허가권에 대해 확인하고 적절한 감사를 진행 하는 함수.

inline int avc_has_perm_noaudit(u32 ssid, u32 tsid,
u16 tclass, u32 requested,
unsigned flags,
*struct av_decision *avd)*

```
inline int avc_has_perm_noaudit(u32 ssid, u32 tsid,
                u16 tclass, u32 requested,
                unsigned flags,
                struct av_decision *avd)
{
    struct avc_node *node;
    struct avc_xperms_node xp_node;
    int rc = 0;
    u32 denied;

    BUG_ON(!requested);

    rcu_read_lock();

    node = avc_lookup(ssid, tsid, tclass);
    if (unlikely(!node))
        node = avc_compute_av(ssid, tsid, tclass, avd, &xp_node);
    else
        memcpy(avd, &node->ae.avd, sizeof(*avd));

    denied = requested & ~(avd->allowed);
    if (unlikely(denied))
        rc = avc_denied(ssid, tsid, tclass, requested, 0, 0, flags, avd);

    rcu_read_unlock();
    return rc;
}
```

→ 허가권에 대해 확인한다. (적절한 감사를 진행하지 않는다)

static inline int avc_audit(u32 ssid, u32 tsid,
u16 tclass, u32 requested,
*struct av_decision *avd,*
int result,
*struct common_audit_data *a,*
int flags)

```
static inline int avc_audit(u32 ssid, u32 tsid,
                u16 tclass, u32 requested,
                struct av_decision *avd,
                int result,
                struct common_audit_data *a,
                int flags)
{
    u32 audited, denied;
    audited = avc_audit_required(requested, avd, result, 0, &denied);
    if (likely(!audited))
        return 0;
    return slow_avc_audit(ssid, tsid, tclass,
                  requested, audited, denied, result,
                  a, flags);
}
```

*static void avc_audit_post_callback(struct audit_buffer *ab, void *a)*

```
/**
 * avc_audit_post_callback - SELinux specific information
 * will be called by generic audit code
 * @ab: the audit buffer
 * @a: audit_data
 */
static void avc_audit_post_callback(struct audit_buffer *ab, void *a)
{
    struct common_audit_data *ad = a;
    audit_log_format(ab, " ");
    avc_dump_query(ab, ad->selinux_audit_data->ssid,
                ad->selinux_audit_data->tsid,
                ad->selinux_audit_data->tclass);
    if (ad->selinux_audit_data->denied) {
        audit_log_format(ab, " permissive=%u",
                 ad->selinux_audit_data->result ? 0 : 1);
    }
}
```

→ selinux 의 특정 정보를 불러옴.

avc_audit_required()

```
static inline u32 avc_audit_required(u32 requested,
                 struct av_decision *avd,
                 int result,
                 u32 auditdeny,
                 u32 *deniedp)
{
    u32 denied, audited;
    denied = requested & ~avd->allowed;
    if (unlikely(denied)) {
        audited = denied & avd->auditdeny;
        /*
         * auditdeny is TRICKY!  Setting a bit in
         * this field means that ANY denials should NOT be audited if
         * the policy contains an explicit dontaudit rule for that
         * permission.  Take notice that this is unrelated to the
         * actual permissions that were denied.  As an example lets
         * assume:
         *
         * denied == READ
         * avd.auditdeny & ACCESS == 0 (not set means explicit rule)
         * auditdeny & ACCESS == 1
         *
         * We will NOT audit the denial even though the denied
         * permission was READ and the auditdeny checks were for
         * ACCESS
         */
        if (auditdeny && !(auditdeny & avd->auditdeny))
            audited = 0;
    } else if (result)
        audited = denied = requested;
    else
        audited = requested & avd->auditallow;
    *deniedp = denied;
    return audited;
}
```

→ 감사에 대한 허가권 등을 수행하는 녀석인것 같다.

*** 이제 selinux 탈출

3. dup_task_struct(current) - 현재프로세스 복사해서 자식 프로세스를 생성함.

```
    /* 현재 프로세스 복사하여 자식 프로세스를 생성함 */
    p = dup_task_struct(current);
```

static struct task_struct *dup_task_struct(struct task_struct *orig)

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

```
339
340 static struct task_struct *dup_task_struct(struct task_struct *orig)
341 {
342     struct task_struct *tsk;
343     struct thread_info *ti;
344     int node = tsk_fork_get_node(orig);
345     int err;
346
347     /* Slab 을 통해 새로 만든 task_struct 를 위한 메모리 할당 받음 */
348     tsk = alloc_task_struct_node(node);
349     if (!tsk)
350         return NULL;
351
352     /* Buddy 를 통해 thread_info 와 Kernel Stack 을 위한 메모리 할당 받음 */
353     ti = alloc_thread_info_node(tsk, node);
354     if (!ti)
355         goto free_tsk;
356
357     /* 본격적인 복사 = 부모 프로세스가 자식 프로세스에 복사됨 */
358     err = arch_dup_task_struct(tsk, orig);
359     if (err)
360         goto free_ti;
361
362     /* Kernel Stack 은 결국 thread_info 임 */
363     tsk->stack = ti;
364 #ifdef CONFIG_SECCOMP
365     /*
366      * We must handle setting up seccomp filters once we're under
367      * the sighand lock in case orig has changed between now and
368      * then. Until then, filter must be NULL to avoid messing up
369      * the usage counts on the error path calling free_task.
370      */
371     tsk->seccomp.filter = NULL;
372 #endif
373
374     setup_thread_stack(tsk, orig);
375     /* 현재 막 만든 따끈따끈한 Task 이므로 재 스케쥴링은 어차피 필요없다.
376        이미 제어권이 자식 프로세스인 tsk 한테 주어져 있기 때문임 */
377     clear_user_return_notifier(tsk);
378     clear_tsk_need_resched(tsk);
379     /* 프로세스 = 프로그램이 메모리에 올라간 형태
380        Stack Overflow 를 통해 root 권한을 획득한다던지
381        악성코드를 실행시킬 수 있는데 Magic Number 를 설정하여
382        Stack Overflow 공격을 감지할 수 있도록 설정해주는 부분임 */
383     set_task_stack_end_magic(tsk);
384
385 #ifdef CONFIG_CC_STACKPROTECTOR
386     /* C 언어 학습할 때 기계어 분석을 했었음
387        실수를 해서 다시 구동 시키면 sp, bp 값이 변동되었음
388        이 값을 바꿔주는 녀석이 바로 이 코드임
389        Random Stack 기법이라고 부름 */
390     tsk->stack_canary = get_random_int();
391 #endif
392
393     /*
394      * One for us, one for whoever does the "release_task()" (usually
395      * parent)
396      */
397     atomic_set(&tsk->usage, 2);
398 #ifdef CONFIG_BLK_DEV_IO_TRACE
399     tsk->btrace_seq = 0;
400 #endif
401     tsk->splice_pipe = NULL;
402     tsk->task_frag.page = NULL;
403     tsk->wake_q.next = NULL;
                                                    403,2-5       16%
```

```
    account_kernel_stack(ti, 1);

    return tsk;

free_ti:
    free_thread_info(ti);
free_tsk:
    free_task_struct(tsk);
    return NULL;
}
```

길다…

tsk = alloc_task_struct_node(node);

```
    /* Slab 을 통해 새로 만든 task_struct 를 위한 메모리 할당 받음 */
    tsk = alloc_task_struct_node(node);
```

```
static inline struct task_struct *alloc_task_struct_node(int node)
{
    return kmem_cache_alloc_node(task_struct_cachep, GFP_KERNEL, node);
}
```

위에 보면 “task_struct_cachep” 가 보이는데 “케쉬의 슬랩” 을 의미한다.
“ GFP_KERNEL” 가 보이는데 “끝날 때 까지 건드리지 마라” 라는 의미.

*void *kmem_cache_alloc_node(struct kmem_cache *cachep, gfp_t flags, int nodeid)*

```
void *kmem_cache_alloc_node(struct kmem_cache *cachep, gfp_t flags, int nodeid)
{
    /* Slab 을 통해 메모리 할당 받음 */
    void *ret = slab_alloc_node(cachep, flags, nodeid, _RET_IP_);

    trace_kmem_cache_alloc_node(_RET_IP_, ret,
                    cachep->object_size, cachep->size,
                    flags, nodeid);

    return ret;
}
```

→ kmem_cache_alloc_node - Allocate an object on the specified node 라는 설명이 있음.
// 어찌 되었든 Slab 을 통해 메모리 할당을 받음.

3-1. slab_alloc_node(cachep, flags, nodeid, _RET_IP_);

*slab_alloc_node(struct kmem_cache *cachep, gfp_t flags, int nodeid,*
unsigned long caller)

slab.c (~/kernel/linux-4.4/mm) - VIM

```
3143 }
3144
3145 static __always_inline void *
3146 slab_alloc_node(struct kmem_cache *cachep, gfp_t flags, int nodeid,
3147            unsigned long caller)
3148 {
3149     unsigned long save_flags;
3150     void *ptr;
3151     /* NUMA ID 값 설정 */
3152     int slab_node = numa_mem_id();
3153
3154     /* flags = GFP_KERNEL */
3155     flags &= gfp_allowed_mask;
3156
3157     lockdep_trace_alloc(flags);
3158
3159     if (slab_should_failslab(cachep, flags))
3160         return NULL;
3161
3162     cachep = memcg_kmem_get_cache(cachep, flags);
3163
3164     cache_alloc_debugcheck_before(cachep, flags);
3165     local_irq_save(save_flags);
3166
3167     if (nodeid == NUMA_NO_NODE)
3168         nodeid = slab_node;
3169
3170     if (unlikely(!get_node(cachep, nodeid))) {
3171         /* Node not bootstrapped yet */
3172         ptr = fallback_alloc(cachep, flags);
3173         goto out;
3174     }
3175
3176     if (nodeid == slab_node) {
3177         /*
3178          * Use the locally cached objects if possible.
3179          * However ____cache_alloc does not allow fallback
3180          * to other nodes. It may fail while we still have
3181          * objects on other nodes available.
3182          */
3183         ptr = ____cache_alloc(cachep, flags);
3184         if (ptr)
3185             goto out;
3186     }
3187     /* ___cache_alloc_node can fall back to other nodes */
3188
3189     /* 실제 slab 을 통해 페이지 할당 */
3190     ptr = ____cache_alloc_node(cachep, flags, nodeid);
3191   out:
3192     local_irq_restore(save_flags);
3193     ptr = cache_alloc_debugcheck_after(cachep, flags, ptr, caller);
3194     kmemleak_alloc_recursive(ptr, cachep->object_size, 1, cachep->flags,
3195                  flags);
3196
3197     if (likely(ptr)) {
3198         kmemcheck_slab_alloc(cachep, flags, ptr, cachep->object_size);
3199         if (unlikely(flags & __GFP_ZERO))
3200             memset(ptr, 0, cachep->object_size);
3201     }
3202
3203     memcg_kmem_put_cache(cachep);
3204     return ptr;
3205 }
3206
3207 static __always_inline void *
                                                    3207,1         74%
```

static inline int numa_mem_id(void)

```
static inline int numa_mem_id(void)
{
    return raw_cpu_read(_numa_mem_);
}
```

```
 * N.B., Do NOT reference the '_numa_mem_' per cpu variable directly.
 * It will not be defined when CONFIG_HAVE_MEMORYLESS_NODES is not defined.
 * Use the accessor functions set_numa_mem(), numa_mem_id() and cpu_to_mem()
```

_numa_mem_ : 위 설명에서 보듯이 cpu 마다 생기는 디랙토리 메모리, 섹션별 메모리를 건드리지 말란 의미.

/* flags = GFP_KERNEL */
flags &= gfp_allowed_mask; → *tags 가 안됨. 인터넷에 직접 찾기.*

→ 인터넷 설명.
flags &= gfp_allowed_mask;

- slab 을 할당 받을 때 gfp_allowed_mask 를 사용하여 허용되지 않는 플래그 비트를 제거한다.
  - 처음 부트업 프로세스를 진행 중에는 __GFP_WAIT, __GFP_IO, __GFP_FS 플래그 요청을 허용하지 않게한다.
  - hibernation 또는 suspend 기능이 동작중인 경우 __GFP_IO, __GFP_FS 플래그 요청을 허용하지 않게한다.

lockdep_trace_alloc(flags); → *얘는 어찌 되었던 디버깅을 해줌.*

```
void lockdep_trace_alloc(gfp_t gfp_mask)
{
    unsigned long flags;

    if (unlikely(current->lockdep_recursion))
        return;

    raw_local_irq_save(flags);
    check_flags(flags);
    current->lockdep_recursion = 1;
    __lockdep_trace_alloc(gfp_mask, flags);
    current->lockdep_recursion = 0;
    raw_local_irq_restore(flags);
}
```

slab_should_failslab(cachep, flags)

```
static bool slab_should_failslab(struct kmem_cache *cachep, gfp_t flags)
{
    if (unlikely(cachep == kmem_cache))
        return false;

    return should_failslab(cachep->object_size, flags, cachep->flags);
}
```

→ 어찌되든 slab 할당이 잘 안이루어질 때 동작하는 것 같다.

```
memcg_kmem_get_cache(struct kmem_cache *cachep, gfp_t gfp)
{
    if (__memcg_kmem_bypass(gfp))
        return cachep;
    return __memcg_kmem_get_cache(cachep);
}
```

→ 메모리를 컨트롤 하는 녀석이다.

→ 들어가보면 이런 녀석이 있다.

struct kmem_cache *__memcg_kmem_get_cache(struct kmem_cache *cachep)

```
struct kmem_cache *__memcg_kmem_get_cache(struct kmem_cache *cachep)
{
    struct mem_cgroup *memcg;
    struct kmem_cache *memcg_cachep;
    int kmemcg_id;

    VM_BUG_ON(!is_root_cache(cachep));

    if (current->memcg_kmem_skip_account)
        return cachep;

    memcg = get_mem_cgroup_from_mm(current->mm);
    kmemcg_id = READ_ONCE(memcg->kmemcg_id);
    if (kmemcg_id < 0)
        goto out;

    memcg_cachep = cache_from_memcg_idx(cachep, kmemcg_id);
    if (likely(memcg_cachep))
        return memcg_cachep;

    /*
     * If we are in a safe context (can wait, and not in interrupt
     * context), we could be be predictable and return right away.
     * This would guarantee that the allocation being performed
     * already belongs in the new cache.
     *
     * However, there are some clashes that can arrive from locking.
     * For instance, because we acquire the slab_mutex while doing
     * memcg_create_kmem_cache, this means no further allocation
     * could happen with the slab_mutex held. So it's better to
     * defer everything.
     */
    memcg_schedule_kmem_cache_create(memcg, cachep);
out:
    css_put(&memcg->css);
    return cachep;
}
```

static inline void cache_alloc_debugcheck_before(struct kmem_cache *cachep,
gfp_t flags)

```
static inline void cache_alloc_debugcheck_before(struct kmem_cache *cachep,
                                gfp_t flags)
{
    might_sleep_if(gfpflags_allow_blocking(flags));
#if DEBUG
    kmem_flagcheck(cachep, flags);
#endif
}
```

local_irq_save(save_flags)

```
#define local_irq_save(flags)                   \
    do {                                \
        raw_local_irq_save(flags);      \
        trace_hardirqs_off();           \
    } while (0)
```

raw_local_irq_save(flags)

```
#define raw_local_irq_save(flags)           \
    do {                                \
        typecheck(unsigned long, flags);    \
        flags = arch_local_irq_save();      \
    } while (0)
```

static inline notrace unsigned long arch_local_irq_save(void)

```
static inline notrace unsigned long arch_local_irq_save(void)
{
    unsigned long flags = arch_local_save_flags();
    arch_local_irq_disable();
    return flags;
}
```

static inline notrace unsigned long arch_local_save_flags(void)

```
static inline notrace unsigned long arch_local_save_flags(void)
{
    return native_save_fl();
}
```

static inline unsigned long native_save_fl(void)

```
static inline unsigned long native_save_fl(void)
{
    unsigned long flags;

    /*
     * "=rm" is safe here, because "pop" adjusts the stack before
     * it evaluates its effective address -- this is part of the
     * documented behavior of the "pop" instruction.
     */

    /* EFLAGS 레지스터는 사용자가 직접 제어할 수 없는 레지스터다.
       그러므로 pushf 를 통해 Stack 에 넣고 pop 을 통해
       flags 변수에 EFLAGS 레지스터의 값을 저장하는 부분이다. */
    asm volatile("# __raw_save_flags\n\t"
             "pushf ; pop %0"
             : "=rm" (flags)
             : /* no input */
             : "memory");

    return flags;
}
```

static inline notrace void arch_local_irq_disable(void)

```
static inline notrace void arch_local_irq_disable(void)
{
    native_irq_disable();
}
```

static inline notrace void arch_local_irq_disable(void)

```
static inline void native_irq_disable(void)
{
    /* cli = Clear Interrupt Flags
       EFLAGS 레지스터의 9 번 비트 치우기
       Interrupt 를 꺼버렸다. */
    asm volatile("cli": : :"memory");
}
```

*static void *____cache_alloc_node(struct kmem_cache *cachep, gfp_t flags,*
int nodeid)

```
  /* 실제 slab 을 통해 페이지 할당 */
  ptr = ____cache_alloc_node(cachep, flags, nodeid);
```

```
static void *____cache_alloc_node(struct kmem_cache *cachep, gfp_t flags,
                int nodeid)
{
    struct list_head *entry;
    struct page *page;
    struct kmem_cache_node *n;
    void *obj;
    int x;

    VM_BUG_ON(nodeid < 0 || nodeid >= MAX_NUMNODES);
    n = get_node(cachep, nodeid);
    BUG_ON(!n);

retry:
    check_irq_off();
    spin_lock(&n->list_lock);
    entry = n->slabs_partial.next;
    if (entry == &n->slabs_partial) {
        n->free_touched = 1;
        entry = n->slabs_free.next;
        if (entry == &n->slabs_free)
            goto must_grow;
    }

    page = list_entry(entry, struct page, lru);
    check_spinlock_acquired_node(cachep, nodeid);

    STATS_INC_NODEALLOCS(cachep);
    STATS_INC_ACTIVE(cachep);
    STATS_SET_HIGH(cachep);

    BUG_ON(page->active == cachep->num);

    obj = slab_get_obj(cachep, page, nodeid);
    n->free_objects--;
    /* move slabp to correct slabp list: */
    list_del(&page->lru);

    if (page->active == cachep->num)
        list_add(&page->lru, &n->slabs_full);
    else
        list_add(&page->lru, &n->slabs_partial);

    spin_unlock(&n->list_lock);
    goto done;

must_grow:
    spin_unlock(&n->list_lock);
    x = cache_grow(cachep, gfp_exact_node(flags), nodeid, NULL);
    if (x)
        goto retry;

    return fallback_alloc(cachep, flags);

done:
    return obj;
}
```

→ Slab 할당자로 실제 페이지를 할당하는 부분이다.

*static struct thread_info *alloc_thread_info_node(struct task_struct *tsk,*
int node)

```
static struct thread_info *alloc_thread_info_node(struct task_struct *tsk,
                          int node)
{
    /* Lazy Buddy 에 의해 16K 메모리 할당 (Order = 2) */
    struct page *page = alloc_kmem_pages_node(node, THREADINFO_GFP,
                          THREAD_SIZE_ORDER);

    /* 물리 메모리와 맵핑되어 있는 가상 주소를 반환함 */
    return page ? page_address(page) : NULL;
}
```

*struct page *alloc_kmem_pages_node(int nid, gfp_t gfp_mask, unsigned int order)*

```
struct page *alloc_kmem_pages_node(int nid, gfp_t gfp_mask, unsigned int order)
{
    struct page *page;

    /* Lazy Buddy 알고리즘에 의해 16 KB 메모리 할당 받음
       그 외의 메모리 크기에 대해서 할당 받을 수도 있음
       현재는 task_struct 를 생성하므로 thread_info 와
       Kernel Stack 때문에 메모리를 할당 받는 것임 */
    page = alloc_pages_node(nid, gfp_mask, order);
    if (page && memcg_kmem_charge(page, gfp_mask, order) != 0) {
        __free_pages(page, order);
        page = NULL;
    }
    return page;
}
```

→ 계속 타고 들어가며 node 를 zone 이 관리 하는 것 까지 볼 수 있다.
→ Lazy Buddy 는 16k 의 메모리 공간을 thread_info 와 kernel stack 에 의해 할당 받는다.

*int arch_dup_task_struct(struct task_struct *dst, struct task_struct *src)*

```
    /* 본격적인 복사 = 부모 프로세스가 자식 프로세스에 복사됨 */
    err = arch_dup_task_struct(tsk, orig);
```

```
int arch_dup_task_struct(struct task_struct *dst, struct task_struct *src)
{
    memcpy(dst, src, arch_task_struct_size);
#ifdef CONFIG_VM86
    dst->thread.vm86 = NULL;
#endif

    /* 현재 부동 소수점과 관련된 내용들이 존재할 수도 있음 */
    return fpu__copy(&dst->thread.fpu, &src->thread.fpu);
}
```

→ fpu 는 부동 소수점 관련한 녀석이라고 했던걸로 기억함.

/* Kernel Stack 은 결국 thread_info 임 */
tsk->stack = ti;

setup_thread_stack(tsk, orig);

```
static inline void setup_thread_stack(struct task_struct *p, struct task_struct *org)
{
    /* 새로 만든 자식의 Kernel Stack 과
       부모의 Kernel Stack 을 동일하게 만들어줌 */
    *task_thread_info(p) = *task_thread_info(org);
    /* 새로 만든 자식의 Kernel Stack 을 들여다보면
       thread_info 가 있고 그 안에는 task 가 들어있다.
       여기에 현재 구동중인 task_struct 가 존재하는데
       현재 구동중인 Task 를 새로 만든 자식으로 지정하는 것임
       즉 current 매크로를 상기해보면 되겠다 */
    task_thread_info(p)->task = p;
}
```

```
#define task_thread_info(task)  ((struct thread_info *)(task)->stack)
```

→ 결론적으로 자식과 부모의 커널 공간을 같게 하고 자식을 현재 구동중인 task 로 만듬.

setup_thread_stack(tsk, orig); *→ 사실 막 만든 task 이기에 스케줄링 필요 x.*

```
static inline void clear_tsk_need_resched(struct task_struct *tsk)
{
    clear_tsk_thread_flag(tsk,TIF_NEED_RESCHED);
}
```

```
static inline void clear_tsk_thread_flag(struct task_struct *tsk, int flag)
{
    clear_ti_thread_flag(task_thread_info(tsk), flag);
}
```

setup_thread_stack(tsk, orig);

```
    /* 프로세스 = 프로그램이 메모리에 올라간 형태
       Stack Overflow 를 통해 root 권한을 획득한다던지
       악성코드를 실행시킬 수 있는데 Magic Number 를 설정하여
       Stack Overflow 공격을 감지할 수 있도록 설정해주는 부분임 */
    set_task_stack_end_magic(tsk);
```

```
void set_task_stack_end_magic(struct task_struct *tsk)
{
    unsigned long *stackend;

    /* Kernel Stack 의 끝을 기록함 */
    stackend = end_of_stack(tsk);
    /* Kernel stack 의 끝에 0x57AC6E9D 을 기록한다.
       이 값을 발견하면 Stack Overflow 공격으로
       운영체제를 공격했음을 감지할 수 있음 */
    *stackend = STACK_END_MAGIC;    /* for overflow detection */
}
```

```
  1     59  include/uapi/linux/magic.h <<STACK_END_MAGIC>>
            #define STACK_END_MAGIC 0x57AC6E9D
```

→ magic 넘버를 기록해서 스텍오버 플로우 공격을 차단해 준다.

4. copy_creds() - 보안과 관련이 있는 녀석이다.

*int copy_creds(struct task_struct *p, unsigned long clone_flags)*

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

```
319  * The new process gets the current process's subjective credentials as its
320  * objective and subjective credentials
321  */
322 int copy_creds(struct task_struct *p, unsigned long clone_flags)
323 {
324     struct cred *new;
325     int ret;
326
327     if (
328 #ifdef CONFIG_KEYS
329         !p->cred->thread_keyring &&
330 #endif
331         clone_flags & CLONE_THREAD
332         ) {
333         p->real_cred = get_cred(p->cred);
334         get_cred(p->cred);
335         alter_cred_subscribers(p->cred, 2);
336         kdebug("share_creds(%p{%d,%d})",
337                p->cred, atomic_read(&p->cred->usage),
338                read_cred_subscribers(p->cred));
339         atomic_inc(&p->cred->user->processes);
340         return 0;
341     }
342
343     new = prepare_creds();
344     if (!new)
345         return -ENOMEM;
346
347     if (clone_flags & CLONE_NEWUSER) {
348         ret = create_user_ns(new);
349         if (ret < 0)
350             goto error_put;
351     }
352
353 #ifdef CONFIG_KEYS
354     /* new threads get their own thread keyrings if their parent already
355      * had one */
356     if (new->thread_keyring) {
357         key_put(new->thread_keyring);
358         new->thread_keyring = NULL;
359         if (clone_flags & CLONE_THREAD)
360             install_thread_keyring_to_cred(new);
361     }
362
363     /* The process keyring is only shared between the threads in a process;
364      * anything outside of those threads doesn't inherit.
365      */
366     if (!(clone_flags & CLONE_THREAD)) {
367         key_put(new->process_keyring);
368         new->process_keyring = NULL;
369     }
370 #endif
371
372     atomic_inc(&new->user->processes);
373     p->cred = p->real_cred = get_cred(new);
374     alter_cred_subscribers(new, 2);
375     validate_creds(new);
376     return 0;
377
378 error_put:
379     put_cred(new);
380     return ret;
381 }
382
383 static bool cred_cap_issubset(const struct cred *set, const struct cred *subset)
                                                                    383,3         42%
```

GROW_up 일때 ,
stack 사이즈에 +1 gkaus 최대 쌓인지점이다.

라고 하셨는데… 왜 그럴까???

→ 우리가 배열 인덱스를 0 부터라고 가정 할 때 마지막 녀석은 size -1 이 된다.
하지만, 반대로 자라는 stack 의 경우는 +1 을 해야 마지막 인덱스(주소)의 위치가 되는 것이다.