# Defeating Samsung KNOX with zero privilege

Di Shen (@returnsme)
KEEN LAB TENCENT (@KEEN_LAB)

KEEN security lab

# Defeating Samsung KNOX with zero privilege

Di Shen (@returnsme)
Keen Lab Tencent (@keen_lab)

## Overview of KNOX 2.6

Bypassing KNOX is necessary if you are trying to apply a rooting exploit on Samsung devices. In April 2016 I found CVE-2016-6787 affected large numbers of Android devices shipped with 3.10 & 3.18 Linux kernel, and successfully make out the rooting exploit. However, the original exploit wasn't working on Samsung Galaxy S7 edge. KNOX introduced many mitigations in Android kernel to prevent from local privilege escalation, including KASLR, DFI, and SELinux enhancement.

In this section we will have a look at the kernel defence implemented by Samsung KNOX 2.6, all analyses are based on Galaxy S7 edge, the Qualcomm-based devices, Hong Kong version.(SM-G9350)

## KASLR (Samsung's implementation)

Samsung implemented its own KASLR for arm64 Linux kernel earlier than UPSTREAM. By enabling CONFIG_RELOCATABLE_KERNEL, kernel will be complied as a PIE executable. Bootloader will pass two parameters to the start entry of Linux kernel, one is the physical address of kernel, another is the actual load address of kernel. Kernel may save the two address to __boot_kernel_offset[3], calculate the randomized offset of kernel.

```
ENTRY(stext)
#ifdef CONFIG_RKP_CFP_ROPP
    /* Must intialize RRK to zero before any RET/BL
    mov RRK, #0
#endif
#ifdef CONFIG_RKP_CFP_JOPP
    /* We need RRS to be loaded before we take our
    load_function_entry_magic_number_before_reloc h
#endif
#ifdef CONFIG_RELOCATABLE_KERNEL
    mov x22, x1             // x1=PHYS_OFFSET
    mov x19, x2             // x2=real TEXT_OFFSET
    adr x21, __boot_kernel_offset
    stp x1, x2, [x21]
#endif
```

Then __relocate_kernel() handles kernel relocating,it's very similar to a aarch64 linker in user space. There is a '.rela' section, contains entries of relative addresses.

```
#ifdef CONFIG_RELOCATABLE_KERNEL

#define R_AARCH64_RELATIVE  0x403
#define R_AARCH64_ABS64     0x101

__relocate_kernel:
    sub x23, x19, #TEXT_OFFSET
    adrp    x8, __dynsym_start
    add x8, x8, :lo12:__dynsym_start    //x8: start of symbol table
    adrp    x9, __reloc_start
    add     x9, x9, :lo12:__reloc_start //x9: start of relocation table
    adrp    x10, __reloc_end
    add x10, x10, :lo12:__reloc_end     //x10: end of relocation table
```

## Real-time kernel protection (RKP)

RKP is implemented in both Linux kernel and secure world. The secure world can be TrustZone or hypervisor, it depends on devices model, for S7 the secure world is TrustZone. According to samsungknox.com, RKP provides following security features:

1. "completely prevents running unauthorized privileged code"
2. "prevents kernel data from being directly accessed by user processes"
3. "monitors some critical kernel data structures to verify that they are not exploited by attacks"

rpk_call() is the syscall entry of RKP. Many critical kernel functions call this function to enter the secure world, including SLAB allocation and deallocation routines, page table operations,and  copy/override/commit credential routines.

### Kernel code protection

This is not an exclusive feature for KNOX 2.6. Most 64 bits Android devices had enabled "KERNEL_TEXT_RDONLY" while compiling, so that the ".text" section is not writable. ".data" section is not executable as well. Based on ARM's feature Privileged eXecute Never (PXN), user code is never executable in kernel mode.

## Kernel page and page table protection

RKP provides read-only kernel pages for sensitive kernel data and objects, only secure world can allocate,de-allocate and manipulate these kernel pages. So these pages' table entries should be protected from page attribute manipulation as well. When kernel need to access protected PGD/PTE/PMD/PUD, related routines will call rpk_call() to enter the secure world.

```
static inline void set_pte(pte_t *ptep, pte_t pte)
{
#ifdef CONFIG_TIMA_RKP
    if (pte && rkp_is_pg_dbl_mapped((u64)(pte))) {
        panic("TIMA RKP : Double mapping Detected pte =
                return;
        }
    if (rkp_is_pte_protected((u64)ptep)) {
        rkp_flush_cache((u64)ptep);
        rkp_call(RKP_PTE_SET, (unsigned long)ptep, pte_v
        rkp_flush_cache((u64)ptep);
    } else {
        asm volatile(
            "mov x1, %0\n"
            "mov x2, %1\n"
            "str x2, [x1]\n"
            :
            : "r" (ptep), "r" (pte)
            : "x1", "x2", "memory" );
    }
#else
    *ptep = pte;
#endif /* CONFIG_TIMA_RKP */
```

## Kernel data protection

Data protection is based on page protection. Some critical global variables are stored in section ".rkp.prot.page", pages in this section cannot be overwritten any more after kernel initialization.

```
#define RKP_RO_AREA __attribute__((section (".rkp.prot.page")))
extern int rkp_cred_enable;
extern char __rkp_ro_start[], __rkp_ro_end[];

extern struct cred init_cred;
```

So far following variables are protected by RKP:

*struct cred init_cred*

*struct task_secrity_struct init_sec*

*struct security_oprations security_ops*

## Kernel object protection

The kernel objects in kernel heap also can be protected by RKP. So far following objects (and their kmem_cache) are protected:

*cred_jar_ro : credential of processes*

*tsec_jar:   security context*

*vfsmnt_cache: struct vfsmount – mount namespace*

These objects are all read-only in kernel/user mode. Allocation, de-allocation and overwriting must be done in secure world. For example, in original Linux kernel, kernel can call override_creds() to update a process's credential. But in Samsung's repository, this function is replaced by rkp_override_creds() , it will allocate credential and security context in read-only kmem_cache then call rkp_call(cmid=0x46) to ask secure world to update process's credential.

```
#ifdef CONFIG_RKP_KDP
const struct cred *rkp_override_creds(struct cred **cnew)
#else
const struct cred *override_creds(const struct cred *new)
#endif  /* CONFIG_RKP_KDP */
{
    const struct cred *old = current->cred;
#ifdef CONFIG_RKP_KDP
    struct cred *new = *cnew;
    struct cred *new_ro;
    volatile unsigned int rkp_use_count = rkp_get_usecount(new);
    void *use_cnt_ptr = NULL;
    void *tsec = NULL;
#endif  /* CONFIG_RKP_KDP */

    kdebug("override_creds(%p{%d,%d})", new,
           atomic_read(&new->usage),
           read_cred_subscribers(new));

    validate_creds(old);
    validate_creds(new);
#ifdef CONFIG_RKP_KDP
    if(rkp_cred_enable) {
        cred_param_t cred_param;
        new_ro = kmem_cache_alloc(cred_jar_ro, GFP_KERNEL);
        if (!new_ro)
            panic("override_creds(): kmem_cache_alloc() failed");

        use_cnt_ptr = kmem_cache_alloc(usecnt_jar,GFP_KERNEL);
        if(!use_cnt_ptr)
            panic("override_creds() : Unable to allocate usage pointer\n");

        tsec = kmem_cache_alloc(tsec_jar, GFP_KERNEL);
        if(!tsec)
            panic("override_creds() : Unable to allocate security pointer\n");

        rkp_cred_fill_params(new,new_ro,use_cnt_ptr,tsec,RKP_CMD_OVRD_CREDS,rkp_use_count);
        rkp_call(RKP_CMDID(0x46),(unsigned long long)&cred_param,0,0,0,0);

        rocred_uc_set(new_ro,2);
        rcu_assign_pointer(current->cred, new_ro);
```

## Credential verifying in secure world

On Galaxy S6, attacker can simply call rkp_override_creds() to bypass the kernel object protection and escalate privilege, but this trick isn't working for S7 any more. RKP add another checking to verify if the submitted new credential is a legal one.

```
if ( !v1 )
  return rkp_printk("NULL pData", 0LL, 0LL, 0LL);
__memcpy((__int64)&v11, v1, 0x30u);
result = intergrity_chk();
if ( result )
{
  v3 = get_caller_thread_info();
  pcb = get_physical_addr(v3);
  old_cred = get_physical_addr(*(_QWORD *)(pcb + 8 * hardcode_table[17]));
  new_cred = get_physical_addr(v11);
  v7 = get_physical_addr(v12);
  new_cred_copy = v7;
  if ( new_cred && v7 && !sub_85803838(v7) )
  {
    if ( (unsigned int)check_there_is_adbd_zygote(pcb, old_cred) && (unsigned int)uid_checking(new_cred, old_cred) )
    {
      rkp_printk(
        "Priv Escalation!",
        new_cred,
        *(_QWORD *)(new_cred + 8LL * HIDWORD(hardcode_table[19])),
        *(_QWORD *)(old_cred + 8LL * HIDWORD(hardcode_table[19])));
      result = priv_escalation_abort(new_cred, old_cred, 1LL);
    }
```

### *Uid_checking()*

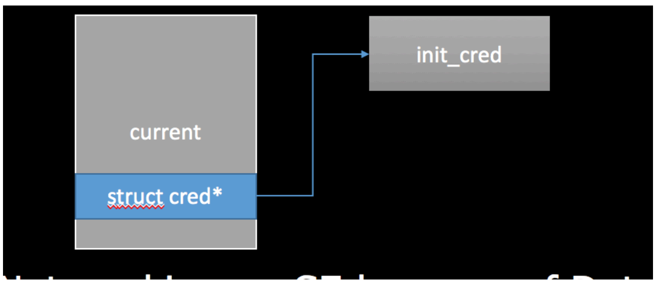
Before adbd and zygote start up, uid_checking will always return ALLOW; after that unprivileged process(uid>1000) cannot override the credential with high privilege (uid 0~1000) any more. That is why the old tricks on S6 was not working any more. However, in fact, on S7 you call still use this trick to modify the kernel capabilities of your current credential, even changing uid is not permitted.

### *Integrity_checking()*

This checking will check if current credential belongs to current process, and check if current security context belongs to current credential. This is very similar to function security_integrity_current() in Linux kernel. We'll analyze this function in next section "Data Flow Integrity".

## Data Flow Integrity (DFI)

There is another old trick to manipulate current credential. For now, we know that credentials are read-only, what if we reuse init process's credential in current context?



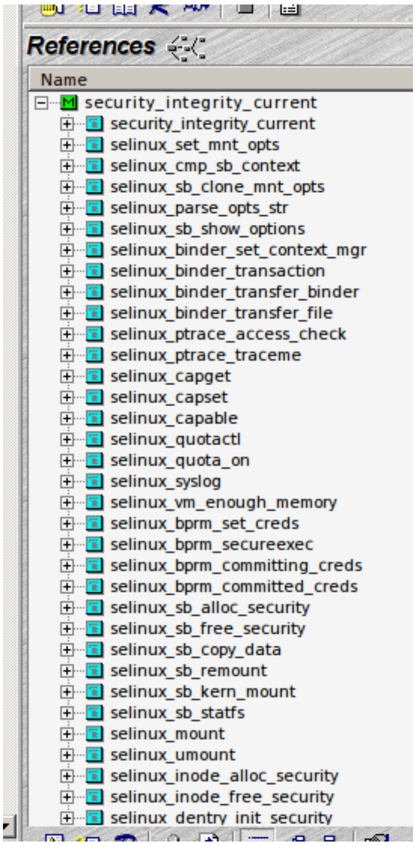
Actually it’s not working because of Data Flow Integrity. DFI defines additional members in struct cred{}

```
#endif
    struct user_struct *user;    /
    struct user_namespace *user_n
    struct group_info *group_info
    struct rcu_head rcu;         /
#ifdef CONFIG_RKP_KDP
    atomic_t *use_cnt;
    struct task_struct *bp_task;
    void *bp_pgd;
    unsigned long long type;
#endif /*CONFIG_RKP_KDP*/
};
#ifdef CONFIG_RKP_KDP
```

bp_task is a pointer to this cred’s owner, bp_pgd is a pointer to process’s PGD. During committing/overriding a new credential in secure world, RKP will record the owner of this credential in bp_task. RKP also record the owner of struct task_security_struct{} in bp_cred.

security_integirity_current() is a hard-coded hooking in every SELinux routines, so almost every Linux syscall will at least call this checking function once to check data’s integrity.

```
}
/* Main function to verify cred security context of a process */
int security_integrity_current(void)
{
    if ( rkp_cred_enable &&
        (rkp_is_valid_cred_sp((u64)current_cred(),(u64)current_cred()->security)||
        cmp_sec_integrity(current_cred(),current->mm)||
        cmp_ns_integrity())) {
        rkp_print_debug();
        panic("RKP CRED PROTECTION VIOLATION\n");
    }
    return 0;
}
unsigned int rkp_get_task_sec_size(void)
```




# Summary of RKP and DFI

With RKP enabled, even we achieved arbitrary kernel memory overwriting, we cannot 1) manipulate credentials and security context in kernel mode; 2) point current credential to init_cred; 3) call rkp_override_creds() to ask secure world to help us override credential with uid 0~1000. But we still can: 1) invoke kernel functions from user mode by hijacking ptmx_fops->check_flags(int flag), note that the number of parameters is limited, only low 32bit of X0 is controllable; 2) Override credential with full kernel capabilities (cred->cap_**); 3) overwrite other unprotected data in kernel.

# SELinux enhancement

## Removed selinux_enforcing

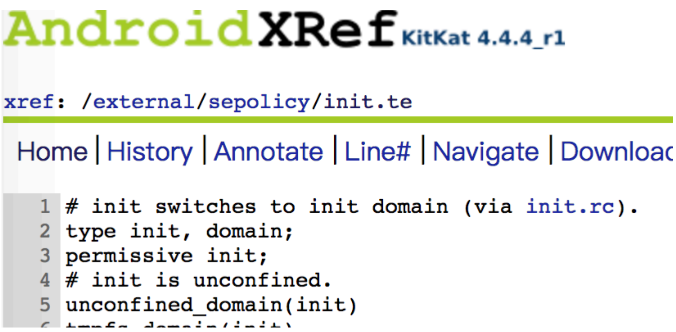
On other Android devices, SELinux can be simply disabled by overwriting “selinux_enforcing” to 0 in Linux kernel. Samsung removed this global variable in kernel by disabling CONFIG_SECURITY_SELINUX_DEVELOP long time ago.

### Disability of policy reloading

And also init process cannot reload SELinux policy after system initialized, which means after Android initialization, attacker cannot simply change its domain to init and reload a customized policy to bypass SELinux.

## Removed support of permissive domain

Furthermore, permissive domain is not allowed neither. The permissive domain was officially used by Google before Lollipop for policy developing purpose. On KitKat you can see that init is a permissive domain, which means even SELinux is enforcing, init process still can do everything it want without a permission deny from kernel.


AndroidXRef KitKat 4.4.4_r1

xref: /external/sepolicy/init.te

Home | History | Annotate | Line# | Navigate | Downloac

```
# init switches to init domain (via init.rc).
type init, domain;
permissive init;
# init is unconfined.
unconfined_domain(init)
```

After that Google remove the permissive domain on Lollipop’s SELinux policy, but permissive domain is still allowed by kernel’s SELinux access vector checking routing. If you can reload a customized SELinux policy with permissive domain declared, it’s still a good way to bypass SELinux. Permissive domain’s access vector database will be marked as AVD_FLAGS_PERMISSIVE, as you can see in avc_denied(), with this flags all denied operation can be allowed.

```
static noinline int avc_denied(u32 ssid, u32 tsid,
                u16 tclass, u32 requested,
                u8 driver, u8 xperm, unsigned flags,
                struct av_decision *avd)
{
    if (flags & AVC_STRICT)
        return -EACCES;

    if (selinux_enforcing && !(avd->flags & AVD_FLAGS_PERMISSIVE))
        return -EACCES;

    avc_update_node(AVC_CALLBACK_GRANT, requested, driver, xperm, ssid,
                tsid, tclass, avd->seqno, NULL, flags);
    return 0;
}
```

Samsung modified the function avc_denied(), this function always returns –EACCESS without any exception on S7.

```
avc_denied.isra.0
MOV                 W0, #0xFFFFFFF3
RET
; End of function avc_denied.isra.0
```

# Bypassing techniques

## Requirements

To build an exploit chain to root Galaxy S7, at least you need two vulnerabilities, one for leak kernel information, another for arbitrary kernel memory overwriting. Combined with following bypass techniques, a fully working exploit chain will be explained in this section.

### Vulnerabilities I used in this chain

The information leaking vulnerability will be disclosed in section “KASLR bypassing”.

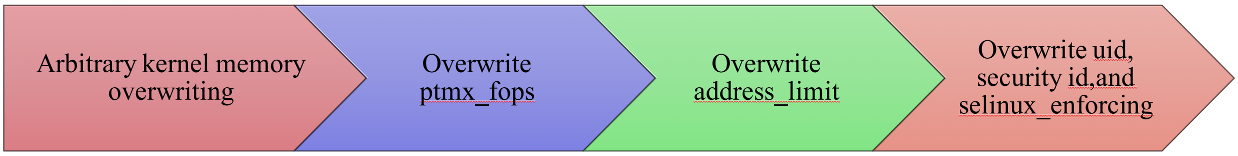
Another one is CVE-2016-6787 found by myself in April 2016, an use-after-free due to race condition in perf subsystem. Note that the patch for “kernel.perf_event_paranoid” was not applied on Android at that time, so that this bug could be triggered by any local application. And also you can use any other exploitable kernel memory corruption bugs instead of this one.

The root cause of this vulnerability is that moving group in sys_perf_event_open() is not locked by mutex correctly. By spraying kernel memory you call refill struct perf_event_context{} and control code flow by triggering ctx->pmu->pmu_disable(X0).To make this exploit 100%

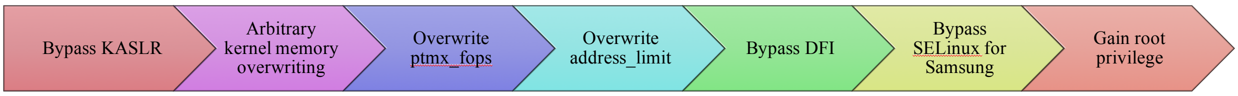
reliable is another long story. A full description of exploiting CVE-2016-6787 may disclose in the future.

# Exploit chain

Rooting a standard Android device normally requires 4 steps.




Rooting S7 requires some additional steps to bypass KNOX mitigation




## KASLR bypassing

On S7 there are some debugging files in /proc fs, the following are TIMA logs.

```
hell@hero2qltechn:/proc $ ls -l | grep tz
|shell@hero2qltechn:/proc $ ls -l | grep tima
rw-r--r-- root     root          0 2016-05-14 16:52 tima_debug_log
rw-r--r-- root     root          0 2016-05-14 16:52 tima_debug_rkp_log
rw-r--r-- root     root          0 2016-05-12 19:44 tima_secure_rkp_log
hell@hero2qltechn:/proc $
```

Kernel pointers leaked in global readable file /proc/tima_secure_rkp_log. At 0x13B80 of this file, it leaked the actual address of init_user_ns. “init_user_ns” is a global variable in kernel’s data section, so with the leaked information, we can calculate the loading offset of Linux kernel, and bypass KASLR.

## DFI bypassing

The main idea is asking kernel to create a privileged process for me, so that I'll not break any checking rules defined by RKP and DFI. However I cannot call call_userermodehelper() via ptmx_fops->check_flags(int), as I've explained above, this function have 4 parameters while I can only pass one from user mode. So I choose to call orderly_poweroff() instead.

```
/**
 * orderly_poweroff - Trigger an orderly system poweroff
 * @force: force poweroff if command execution fails
 *
 * This may be called from any context to trigger a system shutdown.
 * If the orderly shutdown fails, it will force an immediate shutdown
 */
int orderly_poweroff(bool force)
{
    if (force) /* do not override the pending "true" */
        poweroff_force = true;
    schedule_work(&poweroff_work);
    return 0;
}
EXPORT_SYMBOL_GPL(orderly_poweroff);
```

orderly_poweroff() will create a worker thread to create a new user mode process "/sbin/poweroff". The path of this executable file is poweroff_cmd which can be manipulated.

```
char poweroff_cmd[POWEROFF_CMD_PATH_LEN] = "/sbin/poweroff";

static int __orderly_poweroff(bool force)
{
    char **argv;
    static char *envp[] = {
        "HOME=/",
        "PATH=/sbin:/bin:/usr/sbin:/usr/bin",
        NULL
    };
    int ret;

    argv = argv_split(GFP_KERNEL, poweroff_cmd, NULL);
    if (argv) {
        ret = call_usermodehelper(argv[0], argv, envp, UMH_WAIT_EXEC);
        argv_free(argv);
    } else {
```

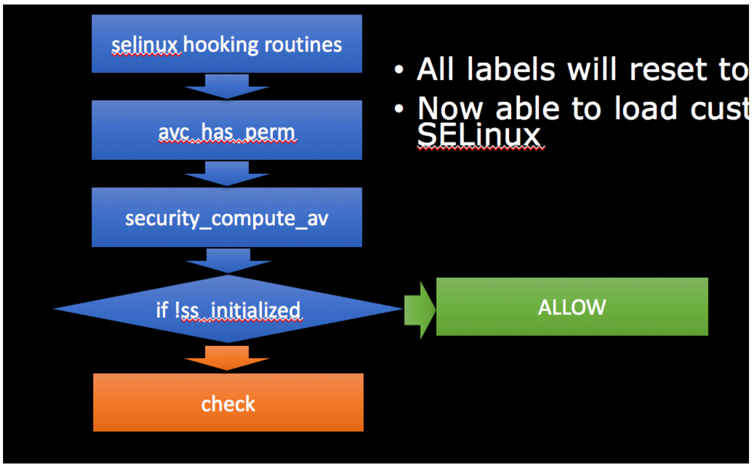
So the bypassing steps are 1) Call rpk_override_creds() via ptmx_fops->check_flags() to override own cred to gain full kernel capabilities 2) Overwrite poweroff_cmd with "/data/data/***/ss7kiler" 3) Call orderly_poweroff() via ptmx_fops->check_flags() 4) Modify ss7killer's thread_info->address_limit 5) ss7killer call rpk_override_creds() to change its context from u:r:kernel:s0 to u:r:init:s0

```
shell@hero2qltechn:/ $ ps -Z | grep init
u:r:init:s0                  root      1     0     /init
u:r:init:s0                  root      20592 1     /data/local/tmp/ss7killer
u:r:init:s0                  root      20608 20592 su
u:r:init:s0                  root      20611 1     daemonsu:mount:master
u:r:init:s0                  root      20614 1     daemonsu:master
u:r:init:s0                  root      20797 20614 daemonsu:0
u:r:init:s0                  root      21161 20614 daemonsu:10193
u:r:init:s0                  root      21163 21161 daemonsu:10193:21157
```

Till now we ask kernel thread to create a root process “ss7killer” with u:r:init:s0 security domain. However, this process is still limited by SELinux, to gain full access we need to bypass SELinux.

## SELinux bypassing

We need to cheat kernel that SELinux is not initialized yet, this status depends on global variable ss_initialized, which is not protected by RKP. If ss_initialized is set to 0, all security labels will be reset to none except kernel domain, all operations can be allowed by SELinux hooking routines, loading customized policy and reinitializing SELinux can be possible.




```
kernel                        root      900   2     dhd_watchdog_th
kernel                        root      901   2     dhd_rpm_state_t
-                             system    903   1     /system/bin/factory.adsp
-                             net_admin 910   1     /system/bin/ipacm
-                             radio     914   1     /system/vendor/bin/qti
-                             radio     922   1     /system/bin/netmgrd
-                             radio     947   1     /system/bin/rild
-                             jack      950   694   androidshmservice
kernel                        root      1157  2     irq/140-arm-smm
kernel                        root      1160  2     irq/141-arm-smm
-                             system    1161  1     /system/bin/mcDriverDaemon
kernel                        root      1209  2     tee_scheduler
kernel                        root      1218  2     irq/162-arm-smm
kernel                        root      1219  2     irq/167-arm-smm
kernel                        root      1221  2     irq/163-arm-smm
kernel                        root      1223  2     irq/168-arm-smm
-                             system    1250  728   system_server
```

After setting ss_initialized to 0, we need to load SELinux policy in user space, modify it with libsepol API. The policy database locates at /sys/fs/selinux/policy. Then insert allow rules into the database, allow domains including “untrusted_app”, init, toolbox to do everything. Finally, we should recover ss_initialized ASAP, otherwise other process with none security label may create none label files and corrupt the file system.

## Gain root

Finally we got a full root access on Samsung Galaxy S7 with KNOX 2.6.