KNOXout - Bypassing Samsung KNOX™

Lev Aronsky // Viral Security Group

Introduction

Samsung KNOX™ is an umbrella name used by Samsung for a collection of security features deployed on its Android devices. While some of its modules are user-facing (such as the *My KNOX™* app), others perform their tasks in the background. One such module, called TIMA[1] RKP (short for Real-time Kernel Protection), is responsible for defending the system in case of a successful kernel exploit.

In this paper, we explain how a standard root exploit subverts the kernel, and explore the protection mechanisms of the RKP module that prevent this kind of exploitation. We then avoid these protections, and execute code in the *system* user context. Malicious access to the *system* account can be used, for instance, to replace legitimate applications with rogue versions, with access to all available permissions, without the user's notice.

Next, we delve deeper into the RKP module, to identify the specific tests performed to prevent privilege escalation. We leverage this newfound knowledge to subvert the RKP module and achieve root privileges. Furthermore, we disable additional kernel protections, and finish off with loading a kernel module, in order to remount the `/system` partition as writable.

A prerequisite for subverting the RKP module is a *write-what-where* kernel vulnerability. While any such vulnerability can be used, for this example we will be using CVE-2015-1805. This vulnerability in the processing of vectored pipes by the Linux kernel is exploitable on recent Samsung devices (such as Galaxy S6 and Galaxy Note 5), and has an open-source exploit implementation named *iovyroot*[2].

[1] ARM TrustZone-based Integrity Measurement Architecture
[2] https://github.com/dosomder/iovyroot

All rights reserved to: Viral Security Group Ltd.

Step 0: Adjusting *iovyroot* for Samsung devices

iovyroot Execution Flow

iovyroot is a pretty straightforward exploitation of a *write-what-where* vulnerability to achieve root:

1. Use the vulnerability to overwrite the `ptmx_fops->check_flags`[3] function pointer with the location of a code gadget which retrieves the current thread info structure. (its execution is triggered by invoking the `fcntl(2)` system call with the *cmd* argument set to `F_SETFL`).
2. Use the vulnerability (again) to patch `addr_info` field of the thread info structure (which, in turn, allows arbitrary reads and writes from/to any kernel address via pipes).
3. Use the access to kernel space via pipes to find the task structure, and overwrite the credentials structure inside with root credentials[4].

iovyroot Prerequisites

While *iovyroot* is a generic Linux kernel exploit, it relies on the prior knowledge of certain locations in the targeted kernel. Normally, these locations are provided by the kernel at runtime (by reading `/proc/kallsyms`) - but on certain Linux systems (Android included), this method requires root access. Fortunately, there is an alternative approach - extracting the symbols offline from a kernel image (obtainable from a firmware update matching the version of the software running on the device).

Since the format of the compressed kernel image differs from that of other Linux executables, its symbols cannot be extracted using regular tools (such as *nm*). Instead, we use *kallsymsprint*[5], developed specifically for the extraction of symbols from a compressed Linux kernel image. However, since the 64-bit Samsung kernel is loaded to a

[3] `ptmx_fops` is a structure containing function pointers, triggered when file operations are carried out on `/dev/ptmx`.
[4] https://github.com/dosomder/iovyroot/blob/master/jni/getroot.c#L61
[5] https://github.com/fi01/kallsymsprint

All rights reserved to: Viral Security Group Ltd.

non-standard address (`0xffffffc0`00205000`, as opposed to `0xffffffc0`00080000`), the tool required some minor modifications to work with the Samsung kernel.

Once we could parse the Samsung kernel properly, another obstacle became apparent. Oftentimes, all the kernel symbols are embedded inside the image, and the required locations can be easily found via *kallsymsprint*. However, Samsung strips all the data symbols from its kernels, leaving only the function names - and *iovyroot* requires the locations of several data structures in the kernel.

To find those locations, we first compiled a Samsung kernel of a version similar to the one found on the device (and in the firmware update file). By compiling the kernel, we had access to the System.map file, which contains the location of all the symbols for the compiled kernel (including the data symbols stripped from the final image).

Next, we disassembled the compiled kernel and found references in code to the data symbols we were looking for. By looking at functions which access the data right in the beginning, we were able to find similar accesses in the same functions in the production kernel. Those accesses finally provided us with the addresses of the data structures in the production kernel.[6]

Example: Finding the location of ptmx_fops:

Looking at the compiled kernel with symbols, we see that there are no cross-references to the `ptmx_fops` variable. However, it is close (16 bits away, to be exact) from another variable, `ptm_driver`. We make an assumption that this will be the case in the stripped kernel, as well.

[6] We later managed to find the source of the kernel matching exactly the device we've been working on. This allowed us to compile a kernel matching the one on device perfectly, and simply look up all of its symbols, making the aforementioned method obsolete. We did, however, want to include it in the paper, since this might not always be the case, and it might prove useful in cases where the source code of the matching version is unavailable.

All rights reserved to: Viral Security Group Ltd.

Disassembling the code that references `ptm_driver` gives us the following excerpt from `ptmx_open`:

vmlinux w/symbols:

```
0xFFFFFFC0004ED244 (ptmx_open+0x6C):
MOV             X0, X19
ADRP            X23, #tty_mutex@PAGE
ADD             X23, X23, #tty_mutex@PAGEOFF
ADRP            X25, #ptm_driver@PAGE
BL              mutex_unlock
MOV             X0, X23
BL              mutex_lock
LDR             X0, [X25,#ptm_driver@PAGEOFF]
MOV             W1, W20
BL              tty_init_dev
MOV             X19, X0
CMN             X19, #1,LSL#12
MOV             X0, X23
B.LS            loc_FFFFFFC0004ED294
```

Armed with this knowledge, we disassemble `ptmx_open` in the stripped kernel (its address is available to us via *kallsymsprint*), and so we arrive at the following code:

vmlinux w/out symbols:

```
FFFFFFC0004E3214 (ptmx_open+0x6C):
MOV             X0, X19
ADRP            X23, #dword_FFFFFFC00132CDD0@PAGE
ADD             X23, X23, #dword_FFFFFFC00132CDD0@PAGEOFF
ADRP            X25, #0xFFFFFFC001795A08@PAGE
BL              sub_FFFFFFC0009C7750
MOV             X0, X23
BL              sub_FFFFFFC0009C797C
LDR             X0, [X25,#0xFFFFFFC001795A08@PAGEOFF]
MOV             W1, W20
BL              sub_FFFFFFC0004DB120
MOV             X19, X0
CMN             X19, #1,LSL#12
MOV             X0, X23
B.LS            loc_FFFFFFC0004E3264
```

The code is pretty much identical to the one in the compiled version, so we can easily match the symbols and calculate that `ptm_driver` is located at `0xFFFFFFC001795A08`. Adding 16-bits, we arrive at the location of `ptmx_fops`, `0xFFFFFFC001795A18`.

All rights reserved to: Viral Security Group Ltd.

Step 1: Testing the limits of RKP

After finding the required locations via our modified *kallsymsprint*, we could run *iovyroot* on a Samsung device, hoping to achieve root access (which was attainable on a multitude of devices from Sony, LG, Huawei and other manufacturers). We were quickly presented with a failed attempt - but the exploitation of the vulnerability itself worked flawlessly. The failure stemmed from the method for obtaining root, which is based on rewriting the credentials of the executing process.

We thoroughly investigated the flow of the exploitation, and attempted to introduce changes to the method (such as using kernel functions to build the root credentials, copy the root credentials from the init process, and more), to no avail. Any attempt to overwrite the process credentials with UID 0 (the root UID), for that matter, any other credentials, failed.

By diving into the source of Samsung's Android kernel, we were finally able to locate the issue - the RKP module.

The RKP module

The RKP module consists of 2 layers: the first layer is interwoven with the Linux kernel, and simply adds or replaces code in strategic places. Meanwhile, the second layer (which is not open-sourced) resides in the ARM TrustZone as a hypervisor. The communication between the two layers is carried out via the `rkp_call` function, which is basically a wrapper around the `hvc` opcode[7]:

arch/arm64/kernel/rkp_entry.S

```
ENTRY(rkp_call)
        hvc     #0
        ret
ENDPROC(rkp_call)
```

[7] The `hvc` opcode (short for Hypervisor Call) is used to jump into the TrustZone hypervisor. It can only be executed in a privileged (kernel) context.

All rights reserved to: Viral Security Group Ltd.

The idea behind RKP is to mask and protect certain areas of kernel memory (such as the "cred area" of process structures). These areas are marked in the kernel source with the `RKP_RO_AREA` prefix, and as the name suggests - they are stored in read-only pages:

kernel/cred.c

```
/*
 * The initial credentials for the initial task
 */
RKP_RO_AREA struct cred init_cred = {
        .usage                  = ATOMIC_INIT(4),
#ifdef CONFIG_DEBUG_CREDENTIALS
        .subscribers            = ATOMIC_INIT(2),
        .magic                  = CRED_MAGIC,
#endif
        .uid                    = GLOBAL_ROOT_UID,
        .gid                    = GLOBAL_ROOT_GID,
        .suid                   = GLOBAL_ROOT_UID,
        .sgid                   = GLOBAL_ROOT_GID,
        .euid                   = GLOBAL_ROOT_UID,
        .egid                   = GLOBAL_ROOT_GID,
        .fsuid                  = GLOBAL_ROOT_UID,
        .fsgid                  = GLOBAL_ROOT_GID,
        .securebits             = SECUREBITS_DEFAULT,
        .cap_inheritable        = CAP_EMPTY_SET,
        .cap_permitted          = CAP_FULL_SET,
        .cap_effective          = CAP_FULL_SET,
        .cap_bset               = CAP_FULL_SET,
        .user                   = INIT_USER,
        .user_ns                = &init_user_ns,
        .group_info             = &init_groups,
#ifdef CONFIG_RKP_KDP
        .use_cnt                = &init_cred_use_cnt,
        .bp_task                = &init_task,
        .bp_pgd                 = (void *) 0,
        .type                   = 0,
#endif /*CONFIG_RKP_KDP*/
};
```

When write access is required (such as when a new process is created, and its cred structure is to be populated), `rkp_call` is invoked with the relevant parameters to provide writing services to the protected area[8]. However, the hypervisor layer of RKP does not provide simple memory write implementations. Instead, it receives more complex requests (such as writing new credentials into a read-only area), and therefore it can perform its own checks and validations, hidden and independent of the kernel. As a

[8] In 8086 architecture, bit 16 of CR0 dictates whether write-protected pages are writable by the kernel. However, no mechanism for writing to write-protected pages without changing the PTEs is available on ARM.

All rights reserved to: Viral Security Group Ltd.

result, code execution in kernel does not guarantee arbitrary writes, and read-only memory areas in the kernel are still protected.

Step 2: Asking RKP nicely

As we could see, both via examining the kernel source code and via actual tests, it is impossible to write to the cred storage, and any such attempt has the cred storage remain unchanged. Furthermore, attempts to point the credentials structure to a different structure (such as that of the *init* process, which has root credentials), were futile, as well.

However, upon further study of the kernel code, we found a special RKP function, `rkp_override_creds`, which replaces the regular kernel function `override_creds`:

kernel/cred.c

```
/**
 * override_creds - Override the current process's subjective credentials
 * @new: The credentials to be assigned
 *
 * Install a set of temporary override subjective credentials on the current
 * process, returning the old set for later reversion.
 */
#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 = kmalloc(sizeof(atomic_t),GFP_KERNEL);
```


All rights reserved to: Viral Security Group Ltd.

```
                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);

                if(!rkp_ro_page((unsigned long)new)){
                        if(atomic_read(&new->usage) == 1) {
                                rkp_free_security((unsigned long)new->security);
                                kmem_cache_free(cred_jar, (void *)(*cnew));
                                *cnew = new_ro;
                        }
                }
        }
        else {
                get_cred(new);
                alter_cred_subscribers(new, 1);
                rcu_assign_pointer(current->cred, new);
        }
#else
        get_cred(new);
        alter_cred_subscribers(new, 1);
        rcu_assign_pointer(current->cred, new);
#endif  /* CONFIG_RKP_KDP */
        alter_cred_subscribers(old, -1);

        kdebug("override_creds() = %p{%d,%d}", old,
               atomic_read(&old->usage),
               read_cred_subscribers(old));
        return old;
}
#ifdef CONFIG_RKP_KDP
EXPORT_SYMBOL(rkp_override_creds);
#else
EXPORT_SYMBOL(override_creds);
#endif  /* CONFIG_RKP_KDP */
```

This kernel function is usually used to achieve a temporary override of the current process's credentials. And, as we can see above, this is exactly what the RKP version does, the difference being that the creds are not written directly, but instead copied into a read-only memory area, via a combination of `rkp_cred_fill_params` and `rkp_call`. Essentially, the work for invoking `rkp_call` is cut out for us, and all we need to do is call `rkp_override_creds` with a pointer to a newly allocated cred structure.

Furthermore, and luckily for us, the RKP version of the function receives a pointer to a cred structure pointer (***cnew*), as opposed to just a cred structure pointer (**new*) in the

All rights reserved to: Viral Security Group Ltd.

original function. This allows us to pass a user-mode pointer, which will be dereferenced into a kernel-mode pointer inside the function - which is crucial, since the argument we provide when invoking `fcntl(2)` (which triggers the kernel function) is stripped to 32 bits.

At this point, the execution flow becomes straightforward (provided that a *write-what-where* capability has been achieved as previously explained in step 0). In the following steps, "executing *xxx* via `check_flags`" refers to replacing the `ptmx_fops->check_flags` function pointer and invoking `fcntl(2)` with the required argument.

1. Execute `prepare_kernel_cred` via `check_flags`. This provides us with a pointer to a valid credentials structure in kernel space, which is writable (not protected by RKP).
2. Update the above credentials with root values (UID=0, GID=0, etc.).
3. Put the pointer to the credentials into a user-mode memory allocated at an address lower that 32 bits (not higher than `0xffffffff`).
4. Pass the above user-mode location to `rkp_override_creds`, executing it via `check_flags`.
5. Enjoy root, as RKP overrides the credentials with *root* values.

Step 3: Finding the middle ground

Unfortunately, the aforementioned scenario does not work. While perfect in theory, it assumes that the hypervisor side of RKP does not validate the overriding credentials, and blindly installs them in place. This assumption proved to be wrong. However, upon further examining both the scarce documentation available about Samsung KNOX™, and the kernel sources, it was apparent that sometimes, *root* credentials receive a special treatment (see, for example, the source of `cap_bprm_set_cred` function, located in `security/commoncap.c`). It is, therefore, not surprising, that the hypervisor side does not take nicely to attempts to override process credentials with *root* values.

All rights reserved to: Viral Security Group Ltd.

However, we found that replacing the *root* values with *system* values (UID=1000, GID=1000, etc.) works perfectly well in the execution flow described in step 2. Upon completion, our process is running with system permissions, and is capable of reading and writing any file that the *system* user has access to (this includes, but not limited to, installed applications).

In conclusion, while achieving *root* credentials proved impossible (so far), the defense mechanisms provided by RKP can be circumvented to achieve *system* permissions, and abuse those to install/replace applications on the device.

Step 4: But all the cool kids have root!

Attempts to fine-tune the code to achieve higher than *system* privileges led us to the conclusion that RKP only checks the UID and EUID fields of the cred structure. Therefore, it is possible to use the vulnerability and get into the *root* group. However, that gives us little benefit, as files accessible by the *root* group and not by *system* are few. Most of them are also read-only to the group, and writable only by a *root* user. It was clear that in order to attempt a bypass of the limitations set in place by the RKP, we'd have to see its code.

After a thorough search through the sources, we found a suspiciously named file, *vmm.elf*. The file resides in Samsung's kernel sources, and during compilation time is inserted, as-is, into the kernel binary. *init/_vmm.S* includes it (and sets the symbols `_svmm` and `_evmm` to mark its start and end). *init/vmm.c* is responsible for using those symbols to copy the binary into an allocated memory and run it. The function, `vmm_init` is called during kernel initialization.

init/_vmm.S

```
#include <linux/vmm.h>

#define vmm_ELF_PATH "init/vmm.elf"

#define SMC_64BIT_RET_MAGIC 0xC2000401

.global _vmm_goto_EL2
_vmm_goto_EL2:
        smc #0
        isb
        ret
```


All rights reserved to: Viral Security Group Ltd.

```
.global _vmm_disable
_vmm_disable:
        ldr x0, =SMC_64BIT_RET_MAGIC
        smc #0
        isb

.section .vmm, "ax"
.global _svmm
_svmm:
.incbin vmm_ELF_PATH
.global _evmm
_evmm:
.section .text
```

init/vmm.c

```
int vmm_init(void) {

        size_t size;
        char *name;
        void *base;

        printk(KERN_ALERT "%s\n", __FUNCTION__);

        if(smp_processor_id() != 0) { return 0; }

        printk(KERN_ALERT "bin 0x%p, 0x%x\n", &_svmm, (int)(&_evmm - &_svmm));

        memcpy((void *)phys_to_virt(VMM_RUNTIME_BASE),  &_svmm, (size_t)(&_evmm - &_svmm));

        vmm = (void *)phys_to_virt(VMM_RUNTIME_BASE);
        vmm_size = VMM_RUNTIME_SIZE;

        printk(KERN_ALERT "ram 0x%p, 0x%x\n", vmm, (int)vmm_size);

        if(ld_get_size(vmm, &size)) { return -1; }

        if(ld_get_name(vmm, &name)) { return -1; }

        printk(KERN_ALERT "%s, %d\n", name, (int)size);

        if(ld_fixup_dynamic_relatab(vmm, &vmm_resolve, &vmm_translate)) { return -1; }

        if(ld_fixup_dynamic_plttab(vmm, &vmm_resolve, &vmm_translate)) { return -1; }

        if(ld_get_sect(vmm, ".bss", &base, &size)) { return -1; }

        memset(base, 0, size);

        vmm_entry();

        return 0;
}
```

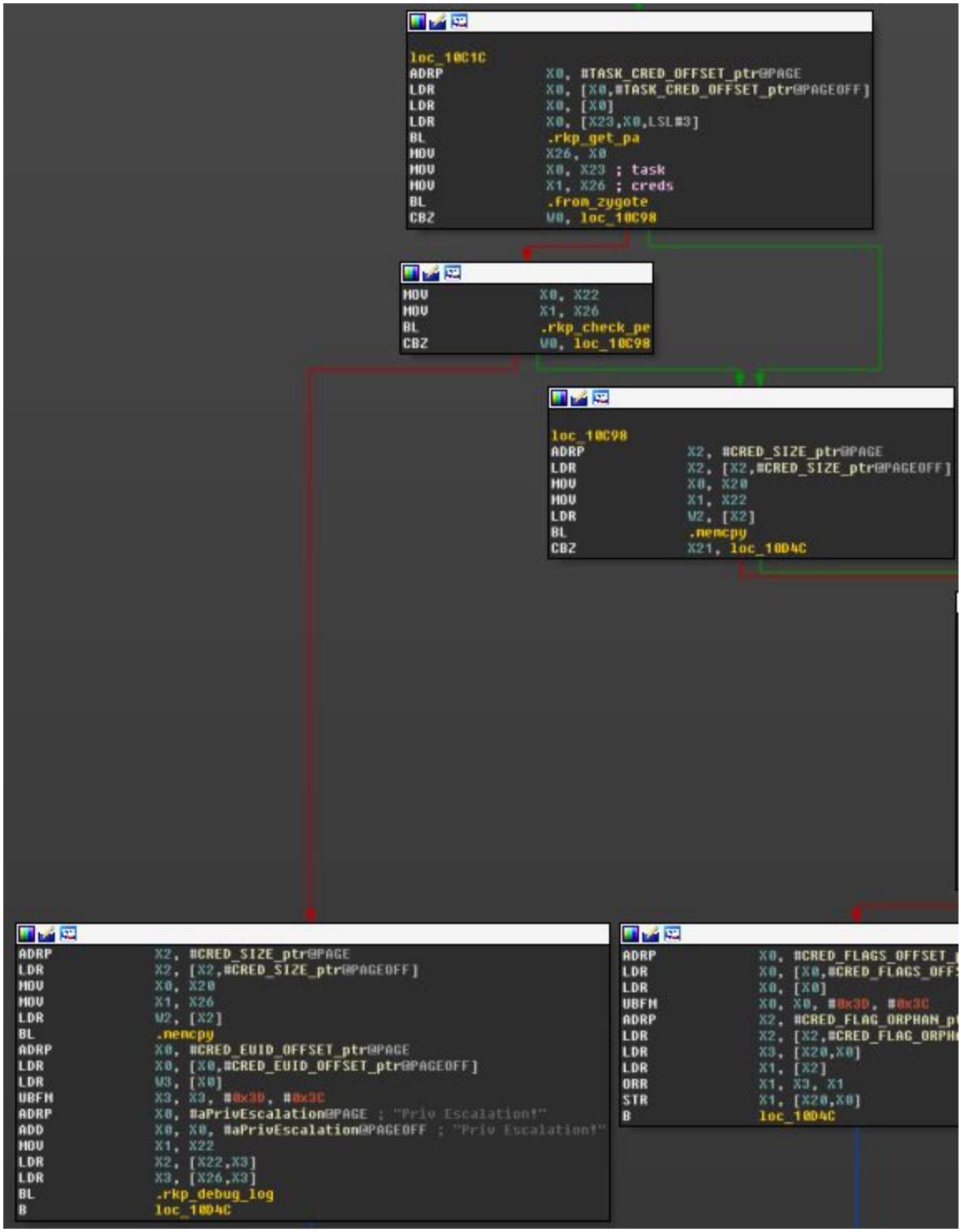
Disassembling the file made it clear beyond any doubt that we were looking at the RKP module, as to our surprise, the file contained all the symbols.

All rights reserved to: Viral Security Group Ltd.

We decided to focus our attention on the function `rkp_assign_creds`. As the name suggests (and the flow that leads to its execution supports), this function is called when the kernel performs an RKP call to change the cred structure of a running process.

All rights reserved to: Viral Security Group Ltd.

It's not a small function, but the presence of a "*Priv Escalation!*" string made it easy to know where to look:



All rights reserved to: Viral Security Group Ltd.

As we can see at the top of the chart, the function `.from_zygote` is called first. If it returns true, the left branch is taken, and `.rkp_check_pe` is called, which can lead us to the bottom left branch, complaining about privilege escalation (and, upon further examination, simply not copying the credentials).

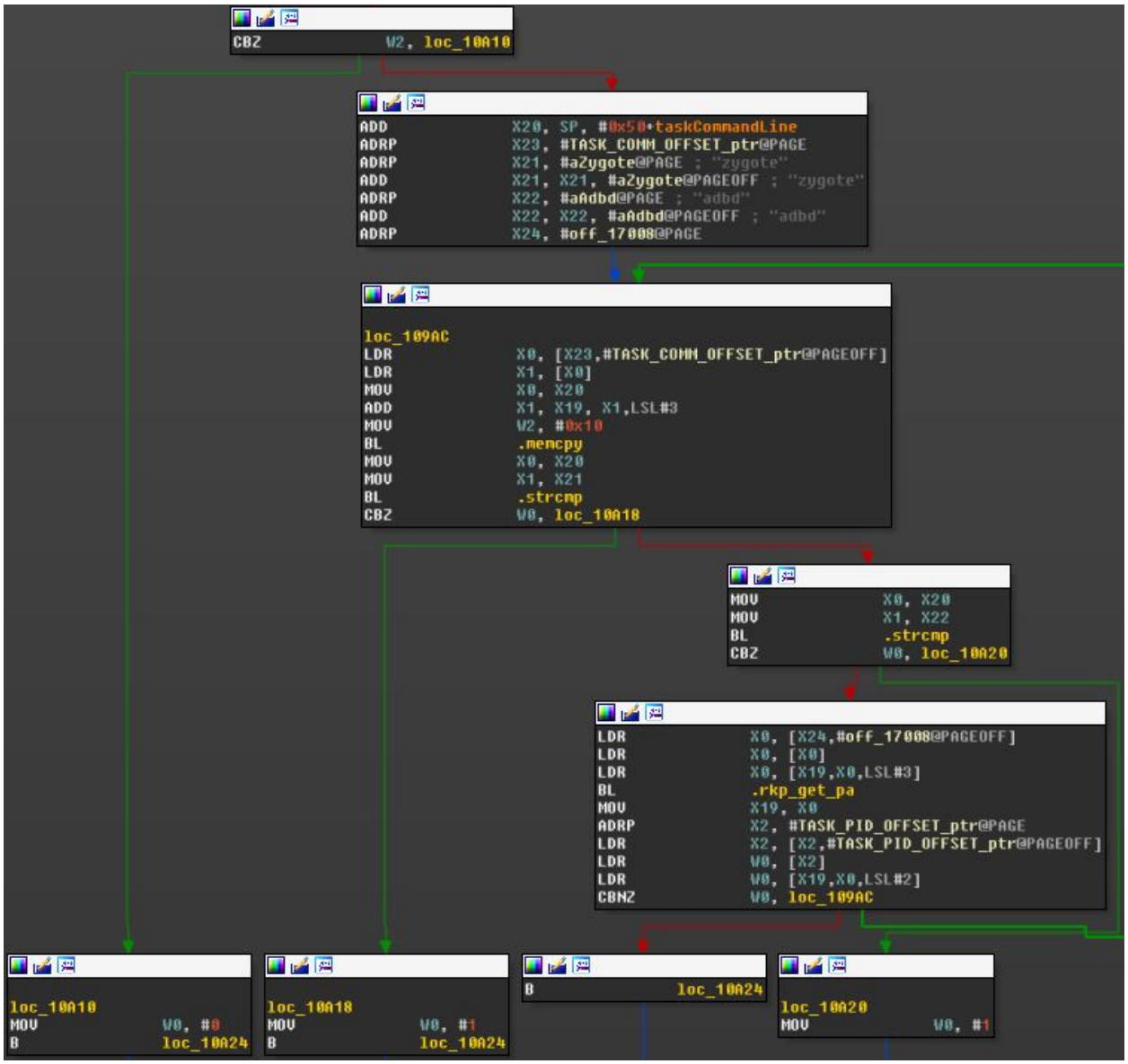
The function `.rkp_check_pe` is small, and just as we'd expect, it simply checks whether the UID and the EUID offsets in the requested cred structure are 0.

The really interesting function, however, is `.from_zygote`. That function tests whether one of the parents of the process, the cred structure of which is to be updated, is *zygote* or *adbd* (in fact, in later versions of vmm.elf, this function is renamed to `.from_zyg_adbd`, to reflect that both of those cases are handled by the function).

The function performs a loop, going up the tree of parent PIDs, and checking the executable path of those parents. Upon reaching *zygote* or *adbd*, the function stops and returns 1, leading us to the check of privilege escalation. Theoretically, since all the APKs are children of *zygote*, and all the commands run in *adb* are children of *adbd*, it seems that the PE check is unescapable.

However, a peculiar test at the beginning of `.from_zygote` makes its bypass trivial (given our kernel memory reading/writing capability). The function starts with the comparison of the current PID to 0, and if that is the case, it returns 0!

All rights reserved to: Viral Security Group Ltd.



We can see the test at the top (the `w2` register holds the current PID), and if it's 0, the return value (stored in `w0`) is 0.

All we have to do to overcome this test is to change our PID to 0 before the RKP call (this can be done by directly writing 0 into the PID offset of our process's). After a successful RKP call (and the setting of *root* UID in our process's cred structure), we restore the PID to its previous value, to avoid any chaos in the kernel scheduling and such.

Voila!

All rights reserved to: Viral Security Group Ltd.

Step 5: Root is for kids. Real men run kernel modules

Upon achieving *root* in the previous step, we found that its permissions are limited, nonetheless. While some of the permissions are limited by SE Linux (which can be disabled), other limitations are incorporated into the kernel at a much deeper level. It was clear that running a kernel module would be a lot more effective at this point.

The kernel in Samsung's Galaxy S6 comes with the option of inserting kernel modules into the kernel. However, the module has to be signed, and the signature verification is performed by an external agent (in Samsung's case, the Mobicore micro-kernel residing in ARM's TrustZone).

Taking a look at the call to Mobicore's signature verification, we see a peculiar test:

kernel/module.c

```
#ifdef TIMA_LKM_AUTH_ENABLED
        if (lkmauth_bootmode != BOOTMODE_RECOVERY &&
            lkmauth(info->hdr, info->len) != RET_LKMAUTH_SUCCESS) {
                pr_err
                    ("TIMA: lkmauth--unable to load kernel module; module len is %lu.\n",
                     info->len);
                return -ENOEXEC;
        }
#endif
```

The `lkmauth` function (which jumps into Mobicore's signature verification) is only called when the `lkmauth_bootmode` variable is set to `BOOTMODE_RECOVERY` (2). However, it is a global variable. Using the methods described in Step 0, we easily found the location of the variable, and used the kernel writing vulnerability to overwrite it with the `BOOTMODE_RECOVERY` value, effectively disabling the signature verification. At this point, we could easily load any kernel module we desired.

All rights reserved to: Viral Security Group Ltd.

Step 6: Possible solutions

Privilege escalation to *system* permissions

As was discussed before, *system* permissions provide the attacker with ample possibilities to compromise the device. Therefore, the RKP module should treat it similar to *root*, and deny credential updates to *system* level.

Privilege escalation to *root* permissions

It is not clear whether the granting of *root* privileges by RKP to processes with PID 0 is necessary, or simply an overlooked mistake. It seems that neither *zygote* nor *adbd* would ever run with PID 0, so at the very least, the PID check (if needed) could be performed later.

Unsigned module loading

The *lkmauth_bootmode* variable should be placed in an RKP-protected, readonly page (similar to the cred structures).

SE Linux[9]

The security_ops structure should be placed in an RKP-protected readonly page (similar to the cred structures). According to the source code, the structure is currently protected by RKP, but real-world tests show otherwise.

Epilogue: Acknowledgements

Thanks go out to @dosomder for the great iovyroot tool exploiting the iovecs vulnerability, as well as @fi01 for the development of kallsymsprint.

[9] See Appendix A

All rights reserved to: Viral Security Group Ltd.

Appendix A: Better kernel-mode execution

While the exploited vulnerability initially provides us with a *write-what-where* capability, it is possible to escalate that into an actual kernel-mode code execution. *iovyroot* does exactly that, by manipulating the `ptmx_fops->check_flags` function pointer, and later invoking a system call, `fcntl(2)`, which ends up executing the code pointed by `check_flags`.

include/linux/fs.h

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, loff_t, loff_t, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned
long, unsigned long);
        int (*check_flags)(int);
        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned
int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned
int);
        int (*setlease)(struct file *, long, struct file_lock **);
        long (*fallocate)(struct file *file, int mode, loff_t offset,
                          loff_t len);
        int (*show_fdinfo)(struct seq_file *m, struct file *f);
        /* get_lower_file is for stackable file system */
        struct file* (*get_lower_file)(struct file *f);
};
```


All rights reserved to: Viral Security Group Ltd.

This method has the following limitations:

- The code has to point to a kernel function (or to any location inside a kernel function, as long as it resides in an executable kernel memory).
- The function has to take one 32-bit-wide argument. Due to the way the `fcntl(2)` system call is handled, the function pointed by `check_flags` is passed one argument, trimmed to 32 bits (in `r0`, as per ARM64 ABI).
- The function's return value is also treated as a 32-bit integer.

These limitations are not a problem for *iovyroot*'s regular flow of execution. However, seeing as we had to beat a complex security system, we decided to look for a more robust execution method, in order to overcome some of the aforementioned limitations.

After searching through a multitude of function pointers, the execution of which could be triggered via a system call, we finally found the `task_prctl` pointer inside the `security_operations` structure:

include/linux/security.h

```
struct security_operations {
        char name[SECURITY_NAME_MAX + 1];
/* … */
        int (*task_wait) (struct task_struct *p);
        int (*task_prctl) (int option, unsigned long arg2,
                           unsigned long arg3, unsigned long arg4,
                           unsigned long arg5);
        void (*task_to_inode) (struct task_struct *p, struct inode *inode);
/* … */
};
```

This function is triggered via the `prctl(2)` system call, and while the `security_ops` structure is marked as read-only by the RKP, to our surprise we were able to override the pointer! While the `task_prctl` function still takes a 32-bit integer as its first argument, it additionally takes 4 64-bit arguments, which coincidentally reach it without any changes from the `prctl(2)` system call originating in our code.

As a result, we achieved a second method of kernel-mode code execution, which allows us to pass up to 5 arguments (with the 1st one being limited to 32 bits), considerably expanding the selection of kernel functions available to us.

All rights reserved to: Viral Security Group Ltd.