

# Breaking Samsung Galaxy Secure Boot through Download mode

Frédéric Basse  
contact@fredericb.info

## Abstract

A bootloader bug in Samsung Galaxy smartphones allows an attacker with physical access to execute arbitrary code. Protections like OS lock screen and reactivation lock can be defeated. Several attacks are possible, including memory dump. Fortunately countermeasures exist for unpatched devices.

## 1 Introduction

Modern mobile devices implement many security features to protect stored user data. Among them, secure boot is critical to prevent unauthorized code from being run. It defines a boot sequence in which each software image loads and authenticates the next one before executing it. Since the first one is not verified, it has to be trusted. A common implementation is to bury it in hardware (BootROM). All other software images can be stored on untrusted storage like internal flash memory.

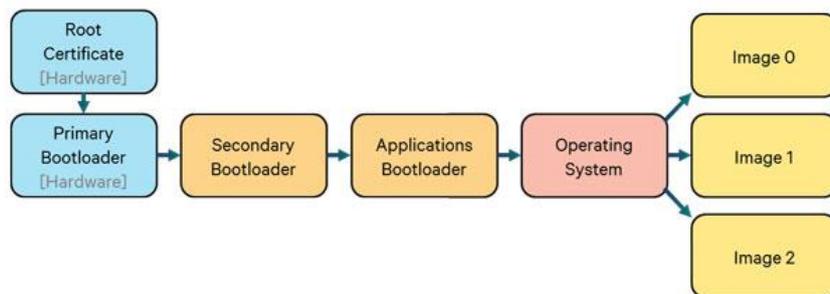


Figure 1: Secure boot sequence (source: [8])

In Android, the Applications Bootloader (*aboot*) is based on Little Kernel (LK), a tiny open-source operating system. One of its main functions is to load and authenticate the Android Linux kernel. Some devices allow to skip OS kernel authentication if *aboot* bootloader is in unlocked state (or unlocked bootloader). But on most retail devices, *aboot* is locked.

OEMs like Samsung frequently introduce proprietary customizations in *aboot* to add new features. For Galaxy devices, Samsung has implemented an alternative boot mode called Download mode (or ODIN). It can be entered by pressing some physical buttons during boot. It gives ability to flash the device with new software images over USB. With a locked bootloader, digital signature of downloaded images are checked.

This paper presents a memory corruption vulnerability in ODIN that can circumvent secure boot. Technical details of bugs are presented in sections 2 and 3. Ultimately, an attacker with physical access can execute arbitrary code to conduct attacks described in section 4. Finally, section 5 explores potential mitigations.

Those experiments have been conducted on Galaxy S5 (SM-G900A), but other models are affected as well.

## 2 Vulnerabilities

The bugs lie in the ODIN flash handler, which writes downloaded image to device storage. They can be triggered by flashing a carefully crafted image. Signature verification should prevent us from reaching this code with a modified image. However, this check is bypassed when device is in T-Flash mode.

### 2.1 T-Flash mode

This mode sets the SD-card as the active storage memory, instead of the internal flash used by default. It means all read and write accesses are performed on the SD-card. This setting is not persistent and only valid in Download mode. The specific ODIN command {0x64, 0x8} switches the device in T-Flash mode:

```
switch ( cmd_id )
{
    case 0x64:
        switch ( sub_cmd_id )
        {
            [...]
            case 0x8:
                display_msg(0xFF00, "T-FLASH MODE");
                if ( init_sdcard() ) // Init SD-card as active storage
                {
                    res = display_msg(8355711, "Failed to Initialize SD card");
                }
                else
                {
                    TFLASH_mode = 1; // Enable T-Flash mode
                    res = odin_upload(0x64);
                }
                break;
            }
        }
}
```

Listing 1: Pseudocode from ODIN handler for T-Flash command

Once SD-card is initialized, **TFLASH\_mode** is set. This variable is read in several functions to bypass signature check. For example, in function that loads the Partition Information Table (PIT) from flash:

```
signed int read_pit_partition_from_mmc()
{
[...]
```

```
    if ( mmc_read(v4, &pit_buffer, 0x2000u) ) {
        display_error(16711680, "ODIN : flash read failure");
        result = -1;
    } else {
        result = check_pit_integrity();
        if ( !result ) {
            if ( !TFLASH_mode ) { // SKIP SIGNATURE CHECK IF T-FLASH MODE
                if ( check_signature(&pit_buffer) == 1 ) {
                    dprintf("SECURE : Signature verification succeed\n");
                    result = 0;
                } else {
                    dprintf("SECURE : Signature verification failed\n");
                    result = -1;
                }
            }
        }
    }
}
return result;
}
```

Listing 2: Pseudocode of function that reads PIT partition from flash

A similar verification bypass is also present in the ODIN flash handler. So T-Flash mode allows to write unsigned arbitrary data on SD-card. Which is not that impressive for a removable storage.

In addition, Download mode doesn't allow to boot the device (with T-Flash enabled or not), so this is definitely not enough to run custom images. But this bypass allows to feed the flawed flash function with arbitrary data.

## 2.2 Buffer overflow to integer overflow to buffer overflow

When the device is booted in Download mode, and once T-Flash mode is enabled, it is possible to download and flash unsigned images on the SD-card. Even if signature verification is not performed, the flashing code still does some parsing of downloaded image.

The structure of boot and recovery images is *Android boot image*. This format is composed of the following sections:

- Header, starting with magic value "ANDROID!"
- Kernel binary
- Ramdisk
- Second stage
- Device Tree Blob (DTB)

But it is also possible to download only the kernel binary. In this specific case, the `update_kernel_in_booting` function is called:

```

if ( !strcmp(part->name, "boot") )
{
    if ( !memcmp(scratch_address, "ANDROID!", 8) )
        goto FLASH_IMAGE; // valid boot image
    }
    else if ( strcmp(part->name, "recovery") || !memcmp(scratch_address,
        "ANDROID!", 8) )
    {
        goto FLASH_IMAGE; // valid recovery image & others
    }
    if ( update_kernel_in_booting(part, &input_data_size) )
    {
        dprintf("ODIN : image is not a boot image");
        return -1;
    }
FLASH_IMAGE:
[...]
```

Listing 3: Pseudocode to handle flashing of kernel binary only

The `update_kernel_in_booting` function reads the current boot (or recovery) partition from flash and updates it with the downloaded kernel binary to obtain a full Android boot image.

```

1 #define ROUND_TO_PAGE(x, y) (__udivsi3((x + y - 1), y))
2 char *scratch_address = 0x11000000;
3 uint update_kernel_in_booting(struct partition_entry *pentry, uint *
4     data_len)
5 {
6     boot_img_hdr *part_dl = scratch_address; //downloaded kernel binary
7     if(part_dl->magic != 0xE1A00000 || part_dl->tags_addr != 0xEA000002)
8     {
9         return -1;
10    }
11    boot_img_hdr *part_read = scratch_address + 0x6900000;
12    long long offset = (long long)pentry->first_lba * 512;
13    uint res = mmc_read(offset, part_read, pentry->size * 512);
14    if(res){
15        printf("ODIN: flash read failure\n");
16        return -1;
17    }else if(memcmp(part_read, "ANDROID!", 8)){
18        return res; // 0
19    }else{
20        boot_img_hdr *part_dl_copy = scratch_address + 0x7300000;
21        memcpy(part_dl_copy, part_dl, *data_len);
22        memcpy(part_dl, part_read, 0x260); // copy image header
23        part_dl->kernel_size = *data_len; // so part_dl->kernel_size !=
24            part_read->kernel_size despite previous memcpy
25        memcpy((char*)part_dl + 0x800, part_dl_copy, *data_len); //update
26            read image with downloaded kernel binary
27
28        uint part_read_page_cnt = ROUND_TO_PAGE(part_read->kernel_size,
29            part_read->page_size); // page count for kernel size
30        uint kernel_size = part_read_page_cnt * part_read->page_size + 0
31            x800;
32        part_read_page_cnt += ROUND_TO_PAGE(part_read->ramdisk_size,
33            part_read->page_size); // page count for ramdisk size
```

```

27     part_read_page_cnt += ROUND_TO_PAGE(part_read->second_size,
28         part_read->page_size); // page count for second size
29     part_read_page_cnt += ROUND_TO_PAGE(part_read->dt_size, part_read
30         ->page_size); // page count for dt size
31     uint part_read_size = (part_read_page_cnt * part_read->page_size)
32         + 0x800;
33     uint kernel_dl_page_cnt = ROUND_TO_PAGE(part_dl->kernel_size,
34         part_dl->page_size); // page count for kernel size from
35         part_dl
36     memcpy((char*)part_dl + (part_dl->page_size * kernel_dl_page_cnt)
37         + 0x800, (char*)part_read + kernel_size, part_read_size -
38         kernel_size); // append original ramdisk, second & dt images
39     *data_len = part_read_size;
40     return 0;
41 }
42 }

```

Listing 4: Reconstruction of flawed function `update_kernel_in_bootimg`

After analyzing this function, the following issues were discovered:

- line 11: Android boot image is read from flash into `part_read` buffer, but important header fields are not verified (sizes, offsets).
- line 22: Size `data_len` of downloaded kernel binary is not checked, so output buffer can overflow if `data_len` is oversized. Therefore `part_read` (located right after `part_dl`) would be overwritten with arbitrary data.
- line 33: The address of the output buffer can overflow because the `part_dl->page_size` value can be set arbitrarily in `part_dl` header. Copy size is fully controlled because `part_read_size` is calculated from arbitrary values in `part_read` header.

Controllability introduced by these issues transforms the last `memcpy` operation (line 33) in a powerful exploitation primitive because:

- output buffer address and copy size can be controlled to some extent;
- input buffer contains arbitrary data (downloaded kernel binary).

### 3 Exploitation

Exploitation is achieved by booting target device in Download mode, enabling T-Flash mode and flashing a specifically crafted boot image. In addition, SD-card must be partitioned and must contain a second crafted boot image.

All these steps can be performed using existing open-source flashing software Heimdall [1], with few changes: adding T-Flash mode support [3], and increasing maximum download size to allow an oversized boot image.

So the main challenge is to craft two images with carefully chosen header values to exploit the memory corruption bug. The objective is to overwrite a part of *boot* code in memory to gain arbitrary code execution.

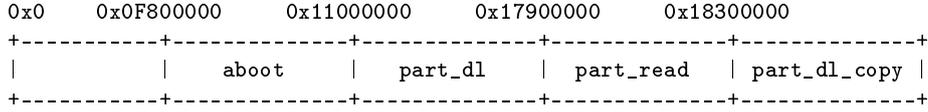


Figure 2: Memory layout during exploitation

To reach *boot* code area, an integer overflow is triggered in the output pointer calculation on line 33 by increasing `part_dl->page_size`. However, due to the header copy on line 20, `part_dl->page_size` is also used for copy size calculation, so it would be too huge for subtle exploitation.

Therefore, the buffer overflow on line 22 is used to overwrite the `part_read` header. Thus there are different page size values in each header, so the copy size is independent from the output pointer. With `part_read->page_size` set to 1, the copy size is equal to `part_read->ramdisk_size`.

But a large `data_len` value reduces the range of allowed values for `part_dl->page_size`, because an integer overflow could occur on line 31, which would invalidate the final output pointer.

Despite all these constraints, it is possible to figure out valid values for successful exploitation:

Variable	Value
<code>data_len</code>	<code>part_read + header_size - (part_dl + 0x800)</code>
<code>part_dl-&gt;page_size</code>	$< (2^{32} - \text{data\_len})$ to avoid overflow
<code>part_dl-&gt;magic</code>	0xE1A00000 (magic value checked on line 6)
<code>part_dl-&gt;tags_addr</code>	0xEA000002 (magic value checked on line 6)
<code>part_read-&gt;page_size</code>	0x1 (overwritten from <code>part_dl</code> buffer)
<code>part_read-&gt;ramdisk_size</code>	copy size (overwritten from <code>part_dl</code> buffer)

With this arbitrary copy operation, the exception vector table located at the beginning of *boot* area is overwritten in memory. In this table, the IRQ handler is replaced with a pointer to the shellcode (see appendix A) located right after the table. Thus, it is executed when an IRQ exception occurs.

It is worth noting exploitation does not void the Knox warranty bit [6].

## 4 Attacks

Successful exploitation allows following attacks:

### 4.1 NAND dump

Attackers can boot generic recovery image like TWRP [5]. This software has built-in features to get a shell and dump internal storage memory. User data partition can be extracted if it is not encrypted.

## 4.2 RAM dump

If target device was booted before the attack, user data in RAM can also be extracted using custom recovery image with additional kernel module [2]. However, original kernel memory could not be recovered, so potential encryption keys are not compromised.

## 4.3 Lock screen and reactivation lock bypass

Instead of booting a custom recovery image, attackers can also boot a modified Android kernel. So protections enforced by OS like lock screen and reactivation lock can be bypassed.

## 4.4 Rogue kernel

Attackers can also boot an Android kernel modified with a backdoor. If they have temporary physical access to an encrypted device, this attack could leak the encryption key when the victim enters passphrase.

# 5 Countermeasures

Vulnerable users still have ways to mitigate attacks:

- Full Disk Encryption prevents user data from being extracted.
- Download mode can be disabled using CC mode [7].
- TIMA Periodic Kernel Measurement (PKM) [4] performs continuous periodic monitoring of the Android kernel from TrustZone. Secure World is not compromised since the attack occurs in Normal World, after Secure World initialization. This proprietary security feature doesn't prevent from booting unsigned Android kernel, but Knox container is disabled.

# 6 Conclusion

This vulnerability allows to run unauthorized software and potentially extract user data from device memory. But disk encryption could not be defeated, even with RAM dump.

The issue has been reported to Samsung Mobile Security team on 2016-12-20 and is identified as SVE-2016-7930. Patch was released in Samsung Security Bulletin of March 2017. Code changes to enable the T-Flash feature have been submitted to the Heimdall project on GitHub [3].

## References

- [1] Heimdall. <http://glassechidna.com.au/heimdall/>.
- [2] LiME - Linux Memory Extractor. <https://github.com/504ensicsLabs/LiME>.
- [3] T-Flash mode: switch from internal MMC to SD-card. <https://github.com/Benjamin-Dobell/Heimdall/pull/389>.
- [4] TIMA Periodic Kernel Measurement (PKM). <http://developer.samsung.com/tech-insights/knox/platform-security>.
- [5] TWRP - Team Win Recovery Project. <https://twrp.me/>.
- [6] What is a Knox Warranty Bit and how is it triggered? <https://www.samsungknox.com/en/qa/what-knox-warranty-bit-and-how-it-triggered>.
- [7] André Moulu. How to lock the samsung download mode using an undocumented feature of aboot. <https://geOn0sis.github.io/posts/2016/05/how-to-lock-the-samsung-download-mode-using-an-undocumented-feature-of-aboot/>, 2016.
- [8] Ryan Nakamoto. Secure boot and image authentication. <https://www.qualcomm.com/news/onq/2017/01/17/secure-boot-and-image-authentication-mobile-tech>, 2017.

## A Shellcode to boot recovery partition from SD-card

```
.section .text
.globl _start
exception_vectors:
    b    _reset
    b    _undef_instr
    b    _swi
    b    _prefetch_abt
    b    _data_abt
    b    _reserved
    b    _start /* overwrite IRQ vector*/
    b    _fiq

_start:
/* save context */
    stmea    sp, {r4-r8}
    mov     r4, sp
/* disable SMC call for Knox warranty */
    ldr r5, Knox_bit_ptr
    ldr r6, nop_instr
    str r6, [r5]
/* replace call to image authentication with call to internal MMC init
*/
    ldr r6, bl_target_mmc_sdhci_init
    ldr r5, auth_img_ptr
    str r6, [r5]
/* set boot_into_recovery flag */
    ldr r5, boot_into_recovery_ptr
    mov r6, #1
    str r6, [r5]
/* inject call to boot_linux_from_mmc into T-FLASH handler */
    ldr r5, odin_error_ptr
    ldr r6, bl_boot_linux_from_mmc
    str r6, [r5]
/* restore context & jump to original IRQ handler */
    ldmia   r4, {r4-r8}
    b      _irq

nop_instr:
    mov     r0, #0 /* dummy instruction to replace function calls */
Knox_bit_ptr:
    .word   0x0F80FA28 /* offset of SMC call for burning Knox warranty
bit */
auth_img_ptr:
    .word   0x0F82164C /* offset of boot image authentication function
*/
bl_target_mmc_sdhci_init:
    .word   0xEBFF7B7E /* offset of MMC initialization function */
boot_into_recovery_ptr:
    .word   0x0F90BFE4 /* offset of recovery boot flag */
bl_boot_linux_from_mmc:
    .word   0xEA000F20 /* branch to boot_linux_from_mmc */
odin_error_ptr:
    .word   0x0F81D564 /* offset of ODIN error handler function */
```