

Attacking Samsung Galaxy A* Boot Chain, and Beyond

Maxime Rossi Bellom
Damiano Melotti
Raphaël Neveu
Gabrielle Viala



Quarkslab

Who we are

- Maxime Rossi Bellom [@max_r_b](#)
 - Security researcher and R&D leader @ Quarkslab
 - Working on mobile and embedded software security
- Gabrielle Viala [@pwissenlit](#)
 - Security researcher and R&D leader @ Quarkslab
 - Playing with low-level stuff
- Damiano Melotti [@DamianoMelotti](#)
 - Ex security researcher @ Quarkslab
 - Interested in low-level mobile security and fuzzing
- Raphaël Neveu
 - Security researcher @ Quarkslab
 - Working on low-level mobile security

Dissecting the Modern Android Data Encryption Scheme

Maxime Rossi Bellom
Damiano Melotti



Quarkslab

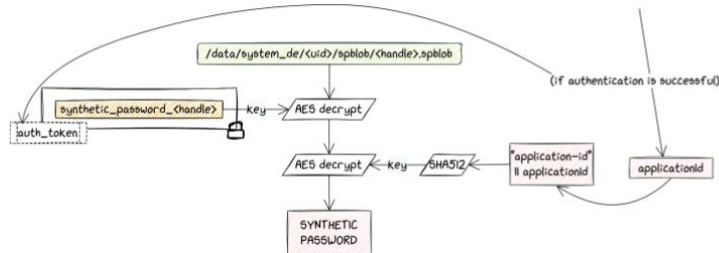
Bruteforce of the password

1. pwd = generate new password
2. token = **scrypt**(pwd, R, N, P, Salt)
3. Application_id = token || Prehashed value
4. Key = **SHA512**("application_id" || application_id)
5. **AES_Decrypt**(value_from_keymaster, key)

```
$ python3 bruteforce-tee.py
workers will cycle through the last 5 chars
Found it: 1234
the plaintext is '1234'
Done in 18.031058311462402s
Throughput: 1478.448992816657 tries/s
```

Attacking SP derivation

- Need to target the TEE
- Two alternatives
 - Keymaster TA (accessing the first AES key)
 - **Gatekeeper TA** (validating credentials and minting auth tokens)



22

```

Preloader - HW version: 8x0
Preloader - MDT: 0x19807800
Preloader - User: 0x11801800
Preloader - Boot payload addr: 0x388200
Preloader - DA payload addr: 0x201009
Preloader - CQ DMA addr: 0x10212800
Preloader - Ver: 0x25
Preloader - Disabling Watchdog...
Preloader - HW code: 0x707
Preloader - Target config: 0x5
Preloader - SEC enabled: True
Preloader - S14 enabled: False
Preloader - S14 enabled: True
Preloader - SMTAC enabled: True
Preloader - EPP PARAN at 0x600 after EMWC BOOT/SDWC BOOT: False
Preloader - Root cert required: False
Preloader - Mem read auth: True
Preloader - Mem write auth: True
Preloader - Onl GxG blocked: True
Preloader - Get Target info
Preloader - BRON mode detected.
Preloader - HW subcode: 0x8a80
Preloader - HW Ver: 0xc80
Preloader - SM Ver: 0x0
Preloader - RE ID: 34C0B89C3AC601793F878155391927F9
Preloader - SOC ID: BED0E12C1C71F2C48C1DC3D079F3FC0D2348ACUCBCEB3C0F336568D3F18D
Preloader - Loading payload from *16768_payload.bin, 0x264 bytes
Preloader - Kanakiri / DA Run
Preloader - Trying kanakiri...
Preloader - Done sending payload.
Preloader - Successfully sent payload: /home/maxime/tools/ntkclient/ntkclient/payloads/nt6768_payload.bin
Port - Device detected: 1
Main - Connected to device, loading
Main - Using custom preloader: preloader.k69v1.64.titan.buffalo.bin
Main - Valid preloader detected.
Main - Patched 'sec.ing.auth' in preloader
Main - Patched 'get_vfy_policy' in preloader
Main - Sent preloader to 0x201006, length 0x3FF24
Preloader - Jumping to 0x201000
Preloader - Jumping to 0x201000, ok.
Main - PL Jumped to 0x201000.
Main - Keep pressed power button to boot.
Main - Waiting for device to boot
    
```



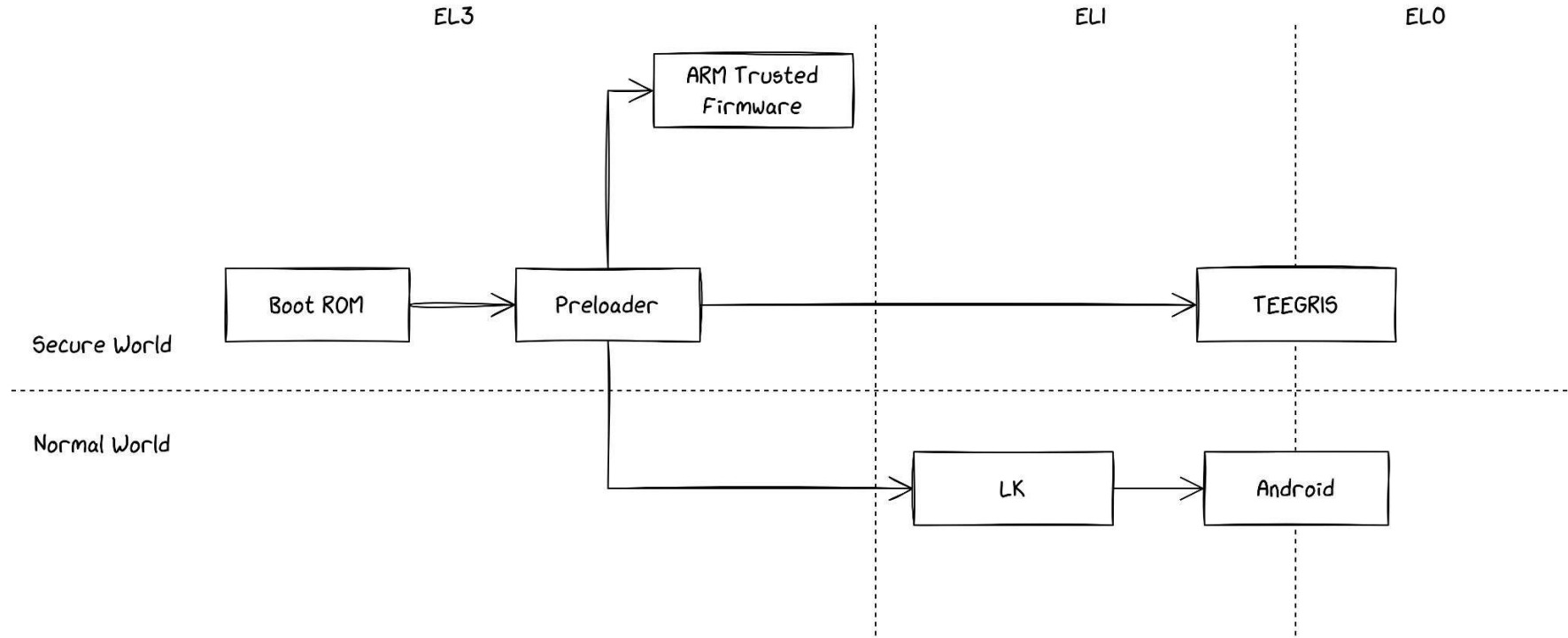
Our Device

- Samsung Galaxy A225F
 - Cheap (~300€)
 - Mediatek SoC MT6769V
 - Main OS: Android
 - Mix of Mediatek and Samsung code
 - Trustzone OS: TEEGRIS
 - Secure Boot Bypass using MTKClient¹
 - ➔ making debugging easier

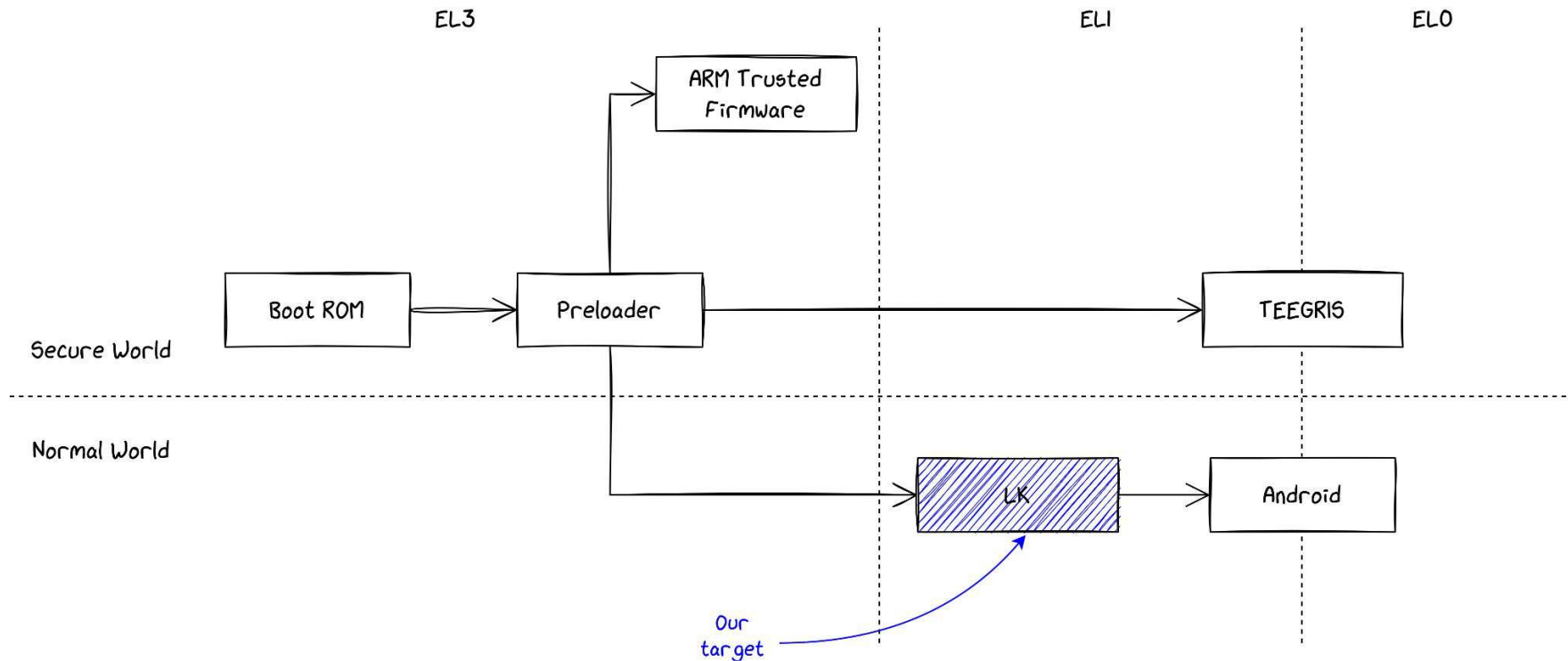


[1]: <https://github.com/bkerler/mtkclient>

Mediatek Secure Boot Process



Mediatek Secure Boot Process



Little Kernel (LK)

- Open-source OS²
- Common as bootloader in the Android world
- Allows to boot Android or other modes (Recovery)
- Implements **A**ndroid **V**erified **B**oot v2
 - Verification of Android images
 - Involving boot and vbmeta partitions
 - Anti-rollback



[2]: <https://github.com/littlekernel/lk>

Little Kernel by Samsung

- Samsung modified LK to include:
 - The Odin recovery protocol
 - Knox Security Bit
 - Etc...
 - And a JPEG parser/renderer
- This version is closed source



Security Error 系统错误

This phone has been flashed with unauthorized software & is locked. Call your mobile operator for additional support. Please note that repair/return for this issue may have additional cost.

本机由于安装了未授权的软件而被锁定,请前往就近的售后服务中心寻求帮助,届时所发生的维修费用有可能需要自行承担,请知悉

Why Targeting the JPEG Loader/Parser

- JPEGs are placed in a TAR archive in the *up_param* partition
- The archive is signed... but the signature is not checked at boot
 - ! Anyone able to write the flash can modify these JPEGs
- Parsing JPEG is known to be hard (cf. LogoFail³)

[3]: <https://www.binarly.io/blog/inside-the-logofail-poc-from-integer-overflow-to-arbitrary-code-execution>

Why Targeting the JPEG Loader/Parser

- JPEGs are placed in a TAR archive in the *up_param* partition
- The archive is signed... but the signature is not checked at boot
 - ! Anyone able to write the flash can modify these JPEGs
- Parsing JPEG is known to be hard (cf. LogoFail³)

How are these JPEGs loaded by LK?

[3]: <https://www.binarly.io/blog/inside-the-logofail-poc-from-integer-overflow-to-arbitrary-code-execution>

Heap Overflow in JPEG Loading

```
_JPEG_BUF = alloc(0x100000);
if (_JPEG_BUF == 0) {
    log("%s: img buf alloc fail\n", "drawing");
    uVar2 = 0xffffffff;
}
else {
    memset(_JPEG_BUF, 0, 0x100000);
    iVar1 = read_jpeg_file(file_name, _JPEG_BUF, 0);
    if (iVar1 == 0) {
        log("%s: read %s from up_param as 0, size\n", "drawing", file_name);
        uVar2 = 0xffffffff;
    }
}
// ...

pimage(*(undefined4 *)(&DAT_4c5107fc + param_1 * 0x3c),
        *(undefined4 *)(&DAT_4c510800 + param_1 * 0x3c),
        0x2d0, 0x640, 1, _JPEG_BUF, iVar1);
```

Heap Overflow in JPEG Loading

Heap allocation of
constant size for the
buffer

```
_JPEG_BUF = alloc(0x100000);  
if (_JPEG_BUF == 0) {  
    log("%s: img buf alloc fail\n", "drawing");  
    uVar2 = 0xffffffff;  
}  
else {  
    memset(_JPEG_BUF, 0, 0x100000);  
    iVar1 = read_jpeg_file(file_name, _JPEG_BUF, 0);  
    if (iVar1 == 0) {  
        log("%s: read %s from up_param as 0, size\n", "drawing", file_name);  
        uVar2 = 0xffffffff;  
    }  
    // ...  
  
    pimage(*(undefined4 *)(&DAT_4c5107fc + param_1 * 0x3c),  
            *(undefined4 *)(&DAT_4c510800 + param_1 * 0x3c),  
            0x2d0, 0x640, 1, _JPEG_BUF, iVar1);
```

Heap Overflow in JPEG Loading

Read the JPEG in
the buffer

```
_JPEG_BUF = alloc(0x100000);
if (_JPEG_BUF == 0) {
    log("%s: img buf alloc fail\n", "drawing");
    uVar2 = 0xffffffff;
}
else {
    memset(_JPEG_BUF, 0, 0x100000);
    iVar1 = read_jpeg_file(file_name, _JPEG_BUF, 0);
    if (iVar1 == 0) {
        log("%s: read %s from up_param as 0, size\n", "drawing", file_name);
        uVar2 = 0xffffffff;
    }
}
// ...

pimage(*(undefined4 *)(&DAT_4c5107fc + param_1 * 0x3c),
        *(undefined4 *)(&DAT_4c510800 + param_1 * 0x3c),
        0x2d0, 0x640, 1, _JPEG_BUF, iVar1);
```

Heap Overflow in JPEG Loading

```
_JPEG_BUF = alloc(0x100000);  
if (_JPEG_BUF == 0) {  
    log("%s: img buf alloc fail\n", "drawing");  
    uVar2 = 0xffffffff;  
}  
else {  
    memset(_JPEG_BUF, 0, 0x100000);  
    iVar1 = read_jpeg_file(file_name, _JPEG_BUF, 0);  
    if (iVar1 == 0) {  
        log("%s: read %s from up_param as 0, size\n", "drawing", file_name);  
        uVar2 = 0xffffffff;  
    }  
    // ...
```

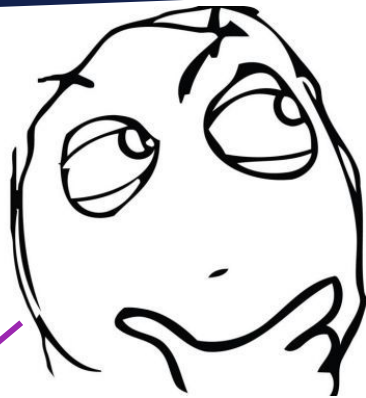
Parse and render
the JPEG



```
pimage(*(undefined4 *)(&DAT_4c5107fc + param_1 * 0x3c),  
        *(undefined4 *)(&DAT_4c510800 + param_1 * 0x3c),  
        0x2d0, 0x640, 1, _JPEG_BUF, iVar1);
```

Heap Overflow in JPEG Loading

```
_JPEG_BUF = alloc(0x100000);  
if (_JPEG_BUF == 0) {  
    log("%s: img buf alloc fail\n", "drawing");  
    uVar2 = 0xffffffff;  
}  
else {  
    memset(_JPEG_BUF, 0, 0x100000);  
    iVar1 = read_jpeg_file(file_name, _JPEG_BUF 0);  
    if (iVar1 == 0) {  
        log("%s: read %s from up_param as 0, size\n", "drawing", file_name);  
        uVar2 = 0xffffffff;  
    }  
    // ...  
  
    pimage(*(undefined4 *)(&DAT_4c5107fc + param_1 * 0x3c),  
            *(undefined4 *)(&DAT_4c510800 + param_1 * 0x3c),  
            0x2d0, 0x640, 1, _JPEG_BUF, iVar1);
```



Heap Overflow in JPEG Loading

- `read_jpeg_file` takes a size as 3rd argument
- It triggers an error if the file does not fit the size provided

```
file_size = string_to_int(tar_header_file.size, 0, 8);  
if (size != 0 && size < file_size) {  
    file_size = print("read fail! (%d < %d)\n", size, file_size, size);  
    return file_size;  
}  
iVar1 = read(data_addr, index + 1, file_size, outbuf);
```


Heap Overflow in JPEG Loading

- `read_jpeg_file` takes a size as 3rd argument
- It triggers an error if the file does not fit the size provided
 - 👉 Unless the size provided is 0...

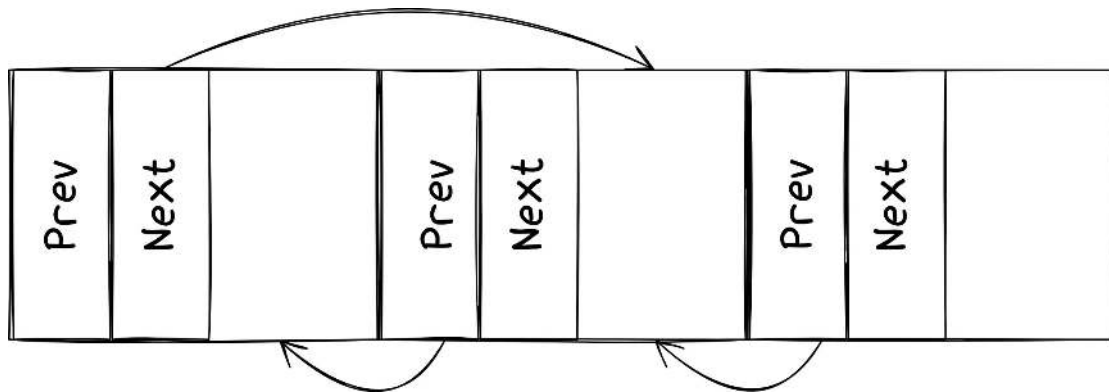
```
file_size = string_to_int(tar_header_file.size, 0, 8);  
if (size != 0 && size < file_size) {  
    file_size = print("read fail! (%d < %d)\n", size, file_size, size);  
    return file_size;  
}  
iVar1 = read(data_addr, index + 1, file_size, outbuf);
```

Is it exploitable?

Exploiting a Heap Overflow in Little Kernel

- The heap algorithm is *miniheap*
 - It relies on a doubly linked list
- Chunks are in a unique memory pool
 - An overflow may overwrite the metadata of next chunk

```
struct free_chunk_head {  
    struct free_chunk_head *prev;  
    struct free_chunk_head *next;  
    size_t len;  
}
```



From Heap Overflow to Arbitrary Write

- After allocation, a chunk is removed from the free list
- `next` and `prev` are dereferenced to change the corresponding nodes
 - ⇒ Controlling a free chunk leads to a write-what-where

```
node->next->prev = node->prev;  
node->prev->next = node->next;  
node->prev = node->next = 0;
```

From Heap Overflow to Arbitrary Write

- After allocation, a chunk is removed from the free list
- `next` and `prev` are dereferenced to change the corresponding nodes
 - ⇒ Controlling a free chunk leads to a write-what-where
 - ! Both values must be writable addresses

```
node->next->prev = node->prev;  
node->prev->next = node->next;  
node->prev = node->next = 0;
```

From Arbitrary Write to Code Execution

Important details about LK

- ✗ No ASLR
- ✗ No canaries
- ✗ No bounds checks in the heap algorithm
- ✗ Heap is executable!

From Arbitrary Write to Code Execution

Important details about LK

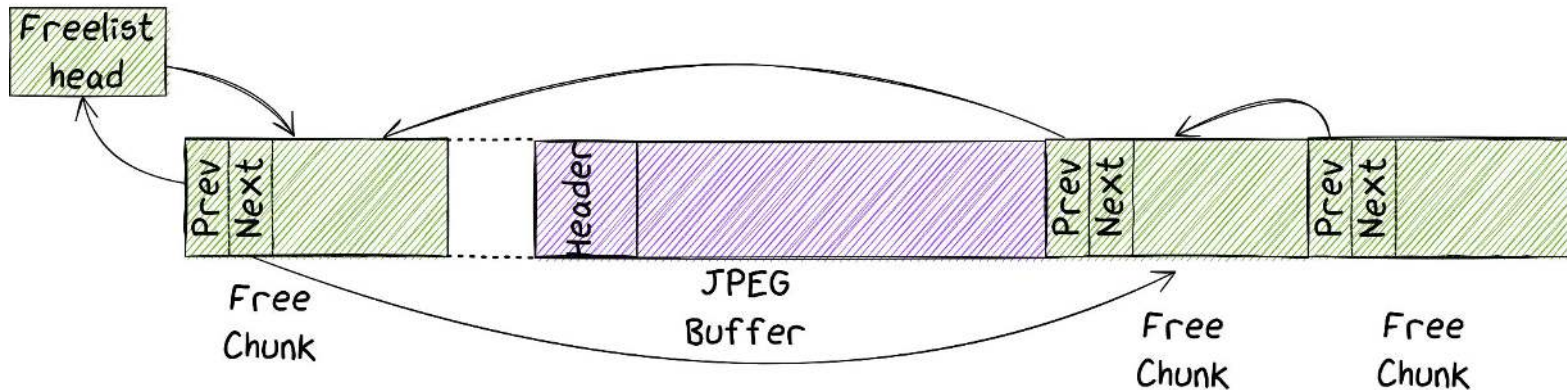
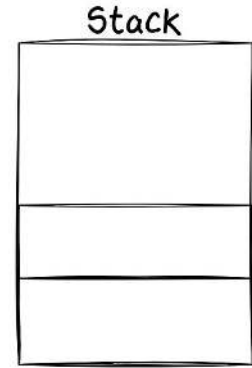
- ✗ No ASLR
- ✗ No canaries
- ✗ No bounds checks in the heap algorithm
- ✗ Heap is executable!

Exploit strategy becomes simple:

1. Overwrite a pointer that the code will jump to
👉 the return address in the stack
2. Make it point to a shellcode in our JPEG buffer

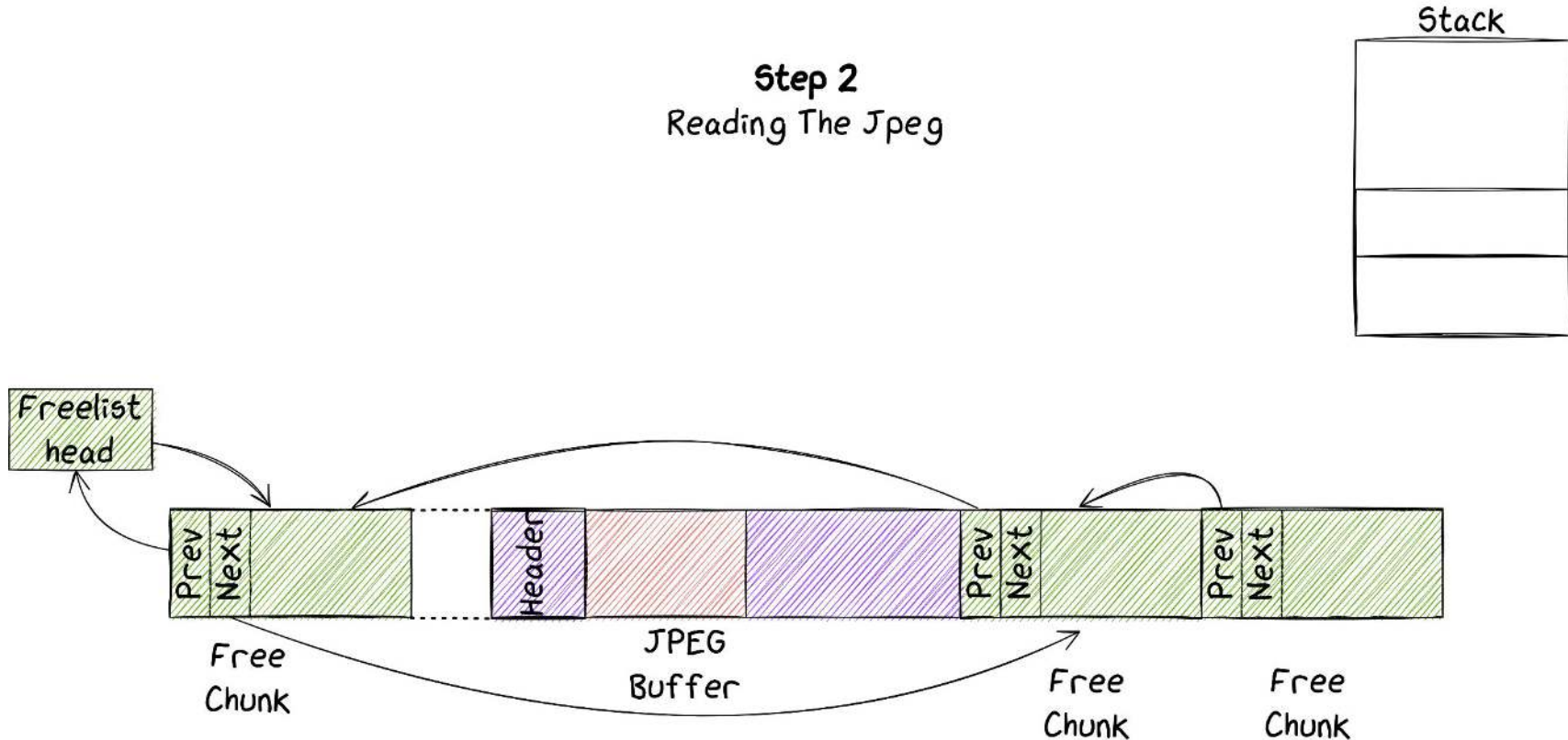
Exploiting a Heap Overflow in Little Kernel

Step I
JPEG Buffer
Allocation



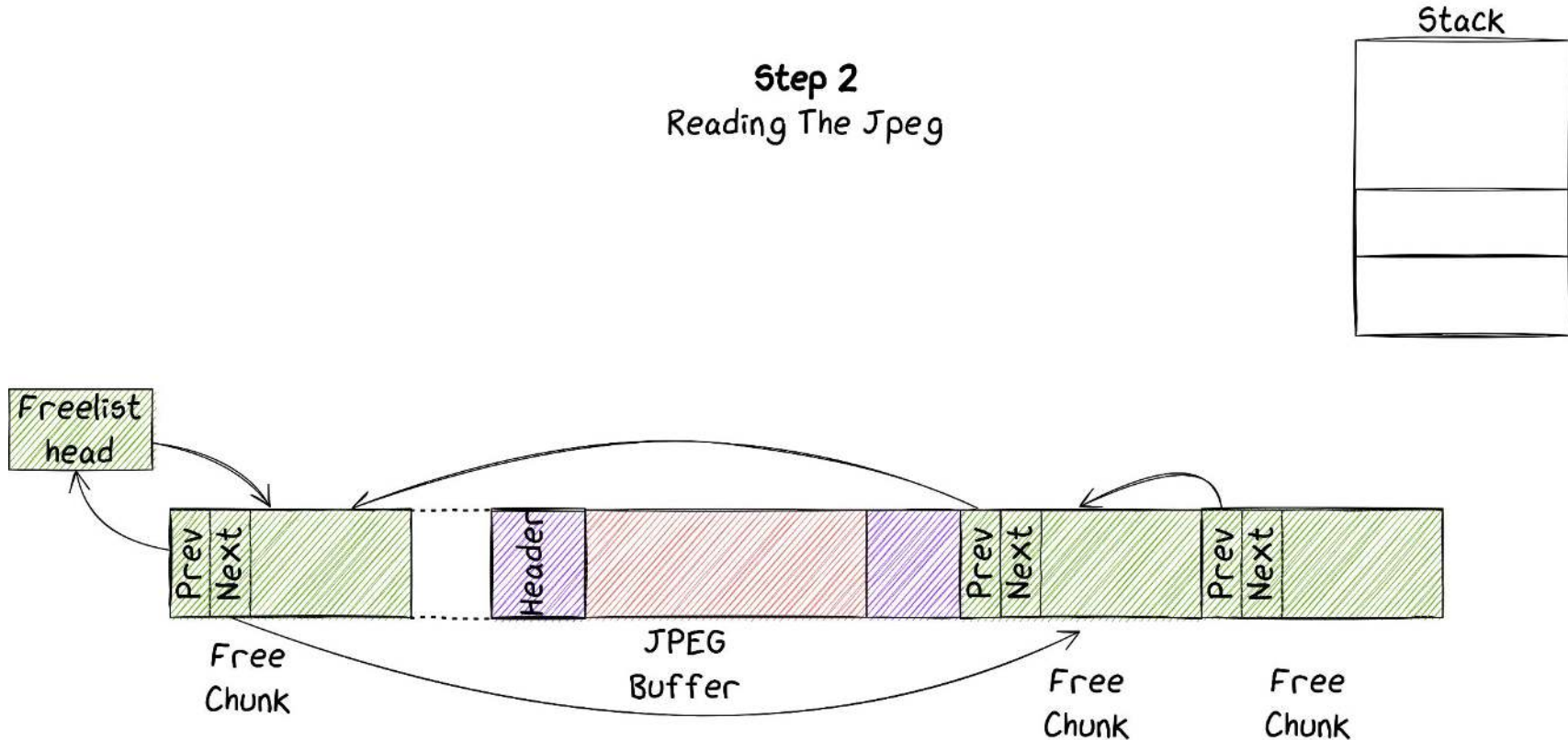
Exploiting a Heap Overflow in Little Kernel

Step 2
Reading The Jpeg



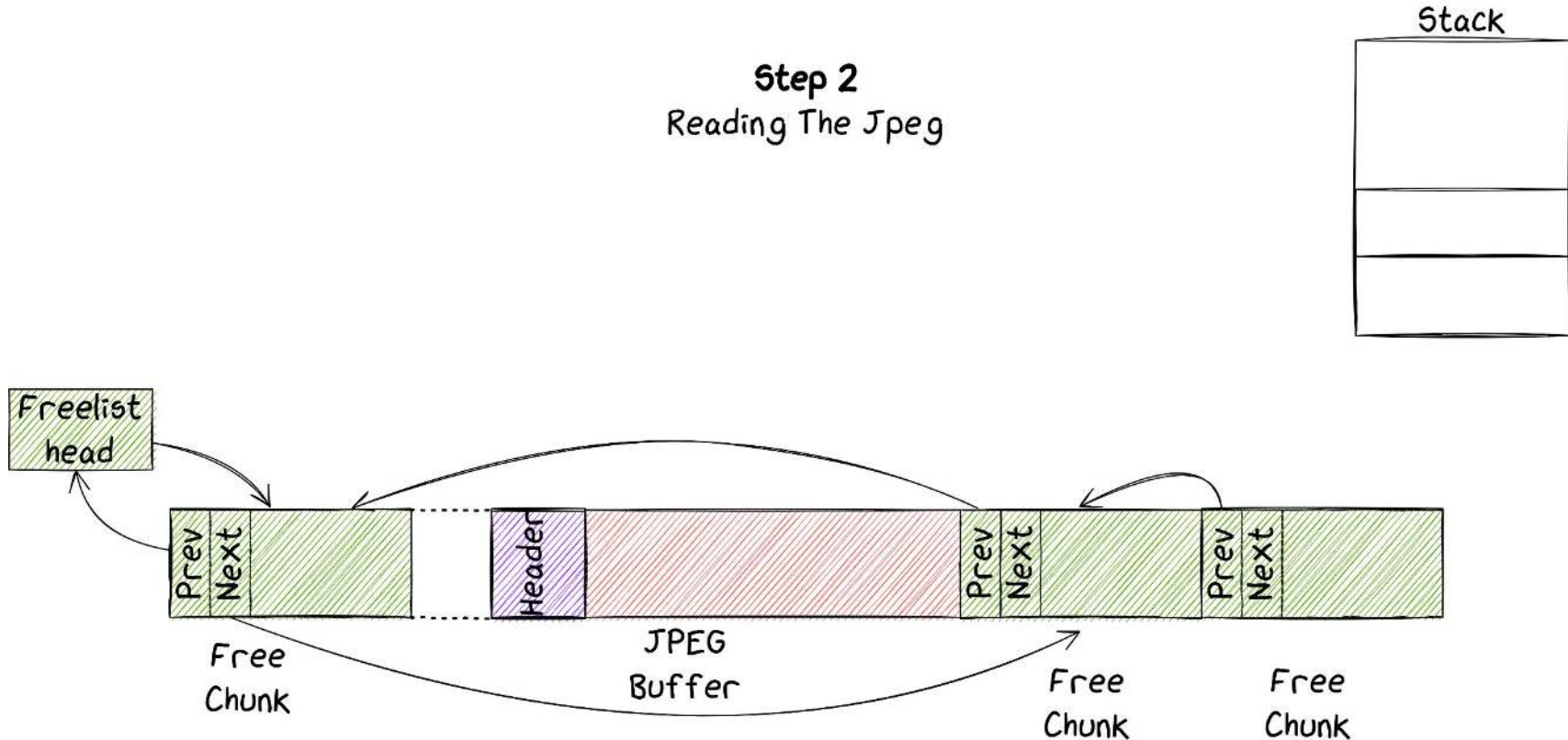
Exploiting a Heap Overflow in Little Kernel

Step 2
Reading The Jpeg



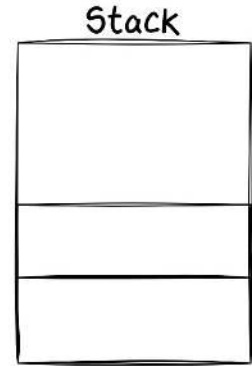
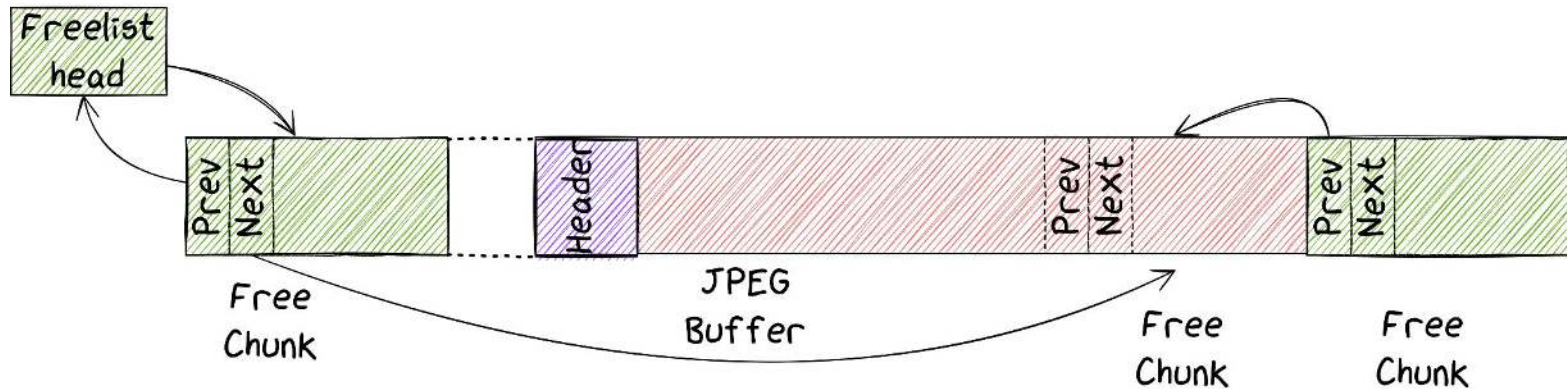
Exploiting a Heap Overflow in Little Kernel

Step 2
Reading The Jpeg

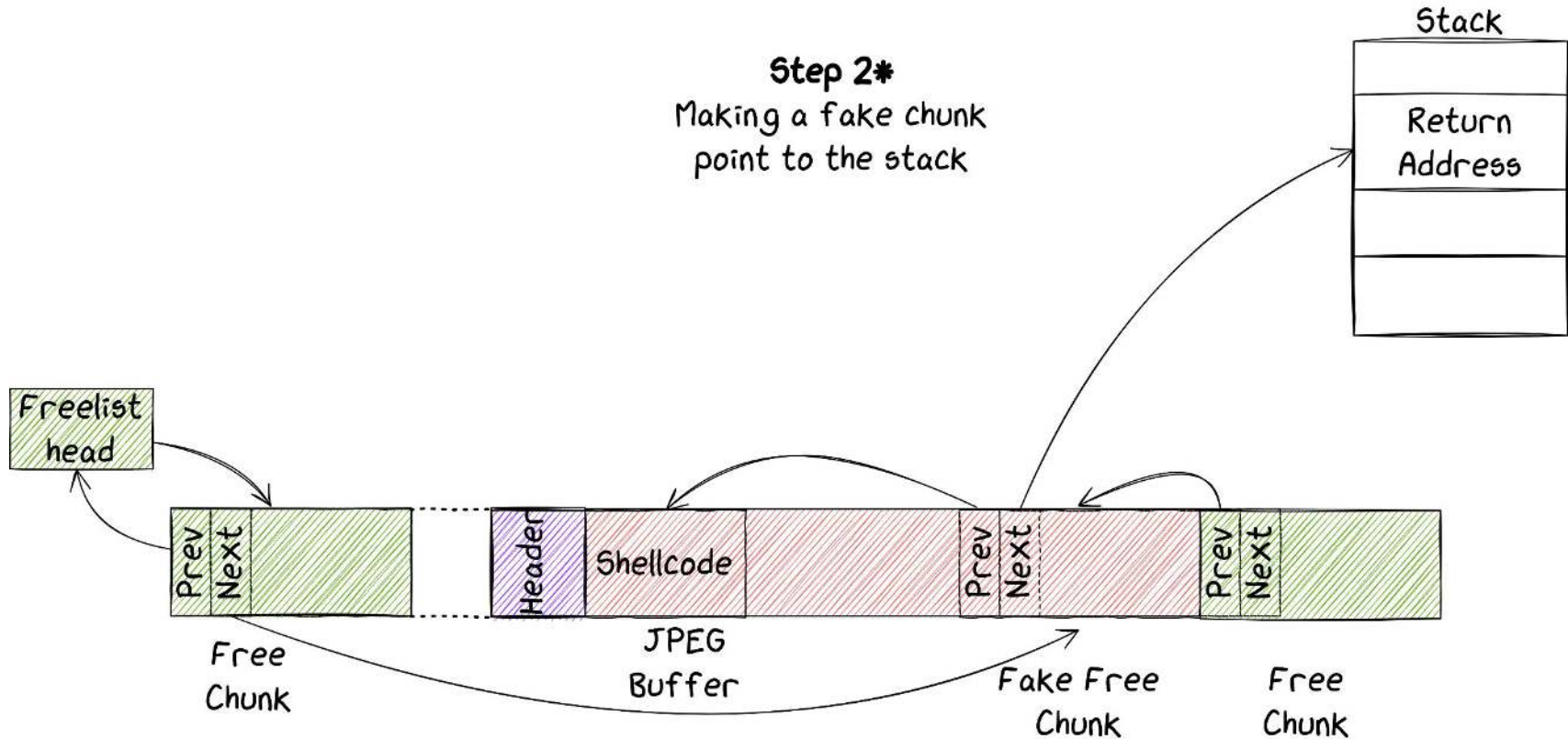


Exploiting a Heap Overflow in Little Kernel

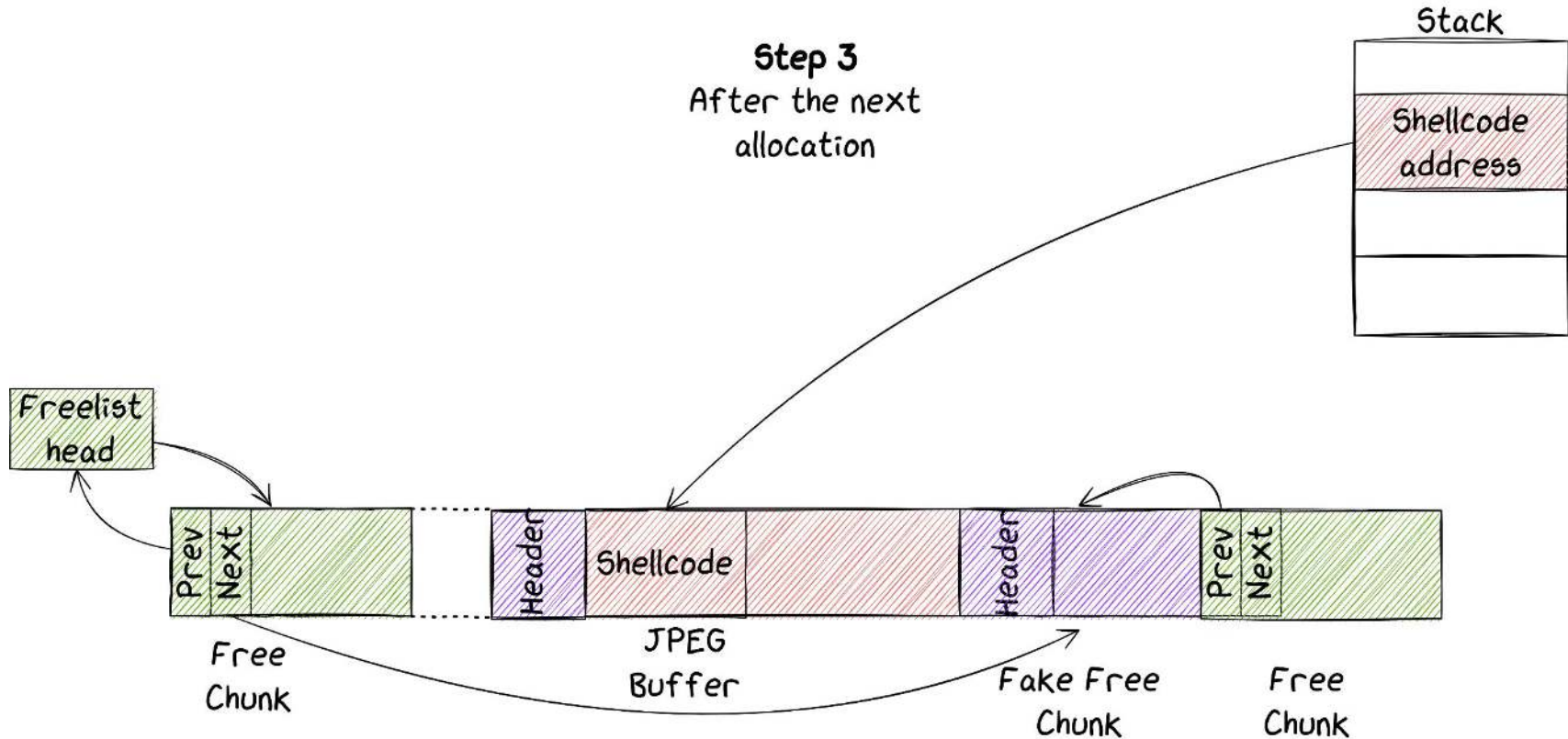
Step 2
Reading The Jpeg
And overwriting next chunk



Exploiting a Heap Overflow in Little Kernel



Exploiting a Heap Overflow in Little Kernel



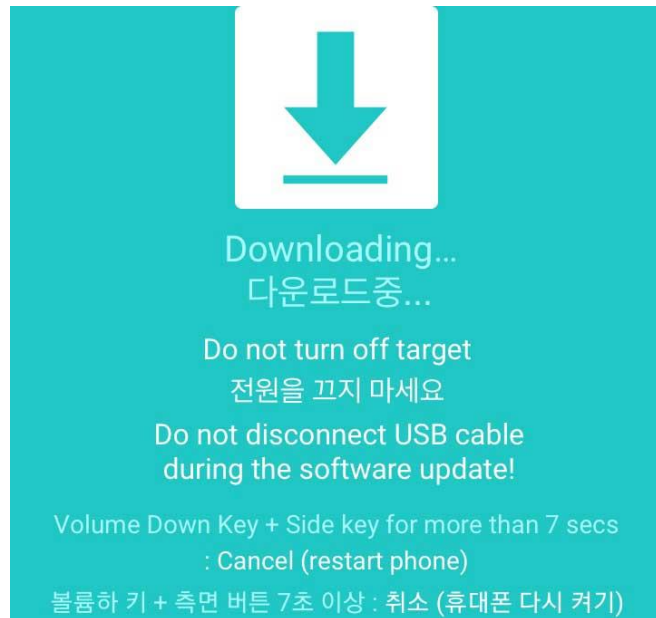
To sum-up

- SVE-2023-2079/CVE-2024-20832
 - ✓ Leads to code execution
 - ✓ Persistent (it survives reboots and factory reset)
 - ✓ Gives full control over Normal World EL1/0
 - ✓ Impacts Samsung devices based on Mediatek SoCs
 - Including those for which MTKClient does not work
 - ✗ Requires to flash the *up_param* partition

*How to write our JPEGs in the
up_param partition?*

Odin: Samsung's recovery protocol

- Odin is implemented in LK
- It is available through the *Download Mode*
 - It allows to flash partitions over USB
- The Odin official client is closed source
- There is an open-source client: Heimdall⁴



[4]: <https://github.com/Benjamin-Dobell/Heimdall>

Odin: Samsung's recovery protocol

- Images are authenticated and contain a footer signature
- Two internal structures indicate which partitions to flash
 - The *Partition Information Table* (PIT)
 - A global structure indicating which partitions to authenticate

53	69	67	6E	65	72	56	65	72	30	32	00	00	00	00	00	SignerVer02.....
36	35	37	33	31	38	36	36	52	00	00	00	00	00	00	00	65731866R.....
41	32	32	35	46	58	58	55	36	44	57	45	33	00	00	00	A225FXXU6DWE3...
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
32	30	32	33	30	35	32	34	31	32	34	37	30	30	00	00	20230524124700..
53	4D	2D	41	32	32	35	46	5F	43	49	53	5F	53	45	52	SM-A225F_CIS_SER
5F	4D	4B	45	59	30	00	00	00	00	00	00	00	00	00	00	_MKEY0.....
53	52	50	55	42	31	35	42	30	30	36	00	00	00	00	00	SRPUB15B006.....

Odin: Partition Information Table

- PIT is retrieved statically from the eMMC
- It indicates where partitions are stored
 - Memory type, block count, etc
- A partition not present in PIT can't be flashed
- PIT can be updated, but requires a signed image

--- Entry #1 ---

Binary Type: 0 (AP)

Device Type: 2 (MMC)

Identifier: 70

Attributes: Read/Write

Update Attributes: 1

Block Size/Offset: 0

Block Count: 34

Partition Name: pgpt

...

Odin: Image Authentication

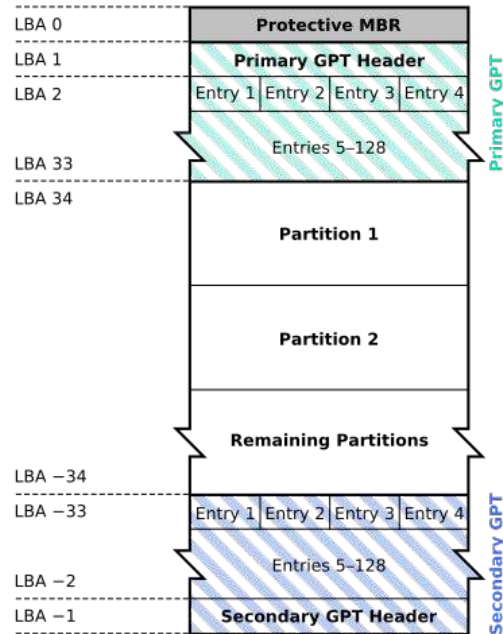
- A global array indicates how an image should be authenticated
- An image not present in this array will not be authenticated
 - (Except for some specific images)
- Comparing this array with PIT gives a set of images flashable without authentication

md5hdr, md_udc, pgpt, sgpt, and vbmeta_vendor

GPT: GUID Partition Table

- **pgpt** points to the Primary GPT Header
- **sgpt** points to the Secondary GPT Header
- Similarly to the PIT, it describes the partitions
 - (Names, sizes, addresses, etc)
- Any GPT can be flashed through Odin
 - ! No authentication required

GUID Partition Table Scheme



GPT vs PIT


- **PIT** and **GPT** are used for the same thing: to describe partitions
 - **PIT** is mainly used for Samsung features in LK
 - Odin, JPEGs loading, etc
 - And **GPT** is used the rest of the time
- ! We can't just rename a partition to *up_param* to flash our JPEGs

PIT Loading

```
pit_address = 0x4400;  
exist = get_part_table("pit");  
if (exist == 0) {  
    pit_address = get_partition_offset("pit");  
}  
type = storage(3);  
iVar1 = storage_read(type, 0x4000, (int)pit_address,  
                     (int)((ulonglong)pit_address >> 0x20),  
                     &ODIN_TEMP_BUF_PIT, 0x4000);
```

PIT Loading

PIT default address



```
pit_address = 0x4400;
exist = get_part_table("pit");
if (exist == 0) {
    pit_address = get_partition_offset("pit");
}
type = storage(3);
iVar1 = storage_read(type, 0x4000, (int)pit_address,
                    (int)((ulonglong)pit_address >> 0x20),
                    &ODIN_TEMP_BUF_PIT, 0x4000);
```


PIT Loading

PIT default address

```
pit_address = 0x4400;
```

```
exist = get_part_table("pit");
```

```
if (exist == 0) {
```

```
    pit_address = get_partition_offset("pit");
```

```
}
```

```
type = storage(3);
```

```
iVar1 = storage_read(type, 0x4000, (int)pit_address,  
                    (int)((ulonglong)pit_address >> 0x20),  
                    &ODIN_TEMP_BUF_PIT, 0x4000);
```

Check for pit partition
And use it if it exists

PIT Loading

PIT default address

```
pit_address = 0x4400;
```

```
exist = get_part_table("pit");
```

```
if (exist == 0) {
```

```
    pit_address = get_partition_offset("pit");
```

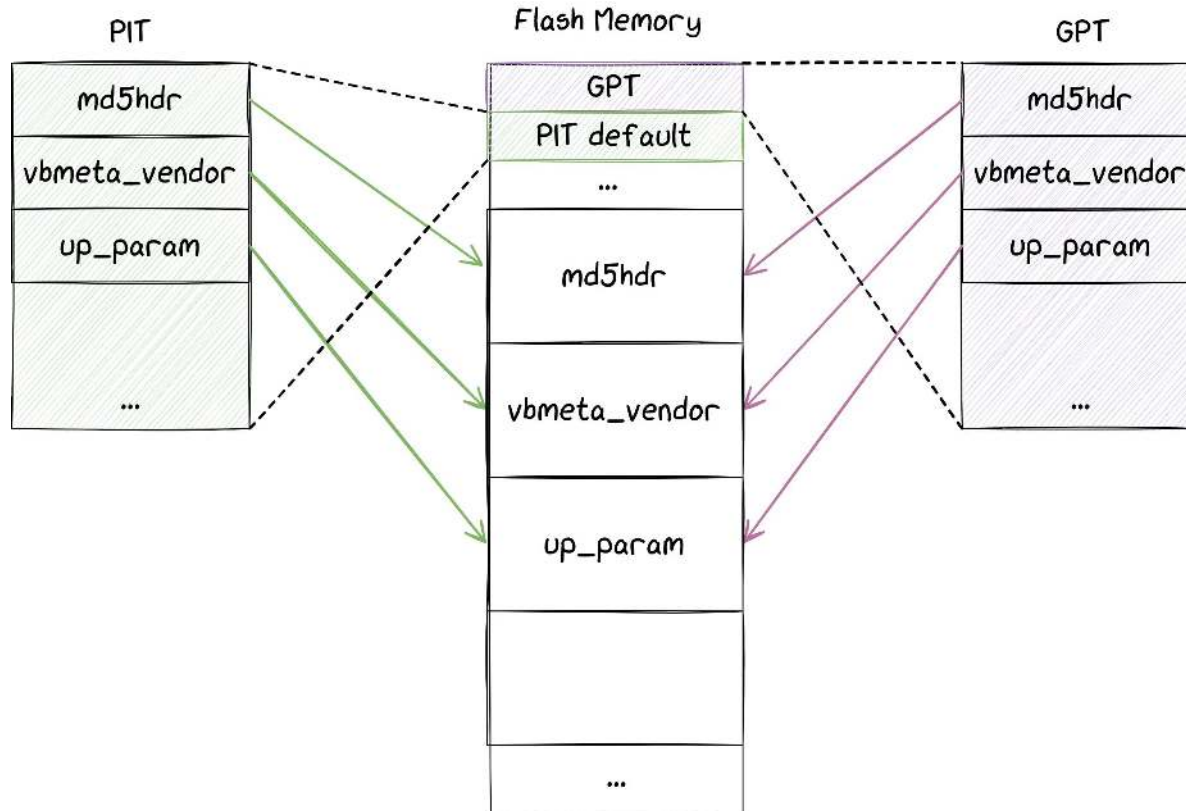
```
}
```

```
type = storage(3);
```

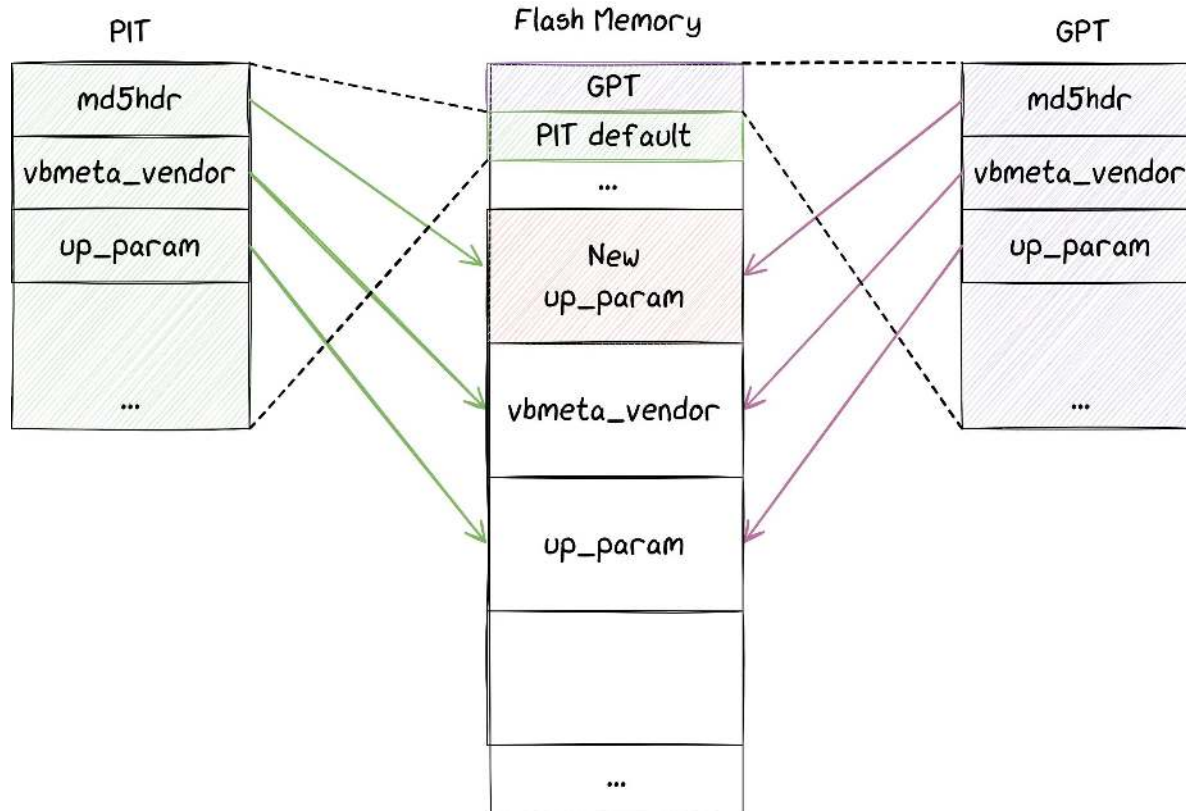
```
iVar1 = storage_read(type, 0x4000, (int)pit_address,  
                     (int)((ulonglong)pit_address >> 0x20),  
                     &ODIN_TEMP_BUF_PIT, 0x4000);
```

Uses GPT table 😈

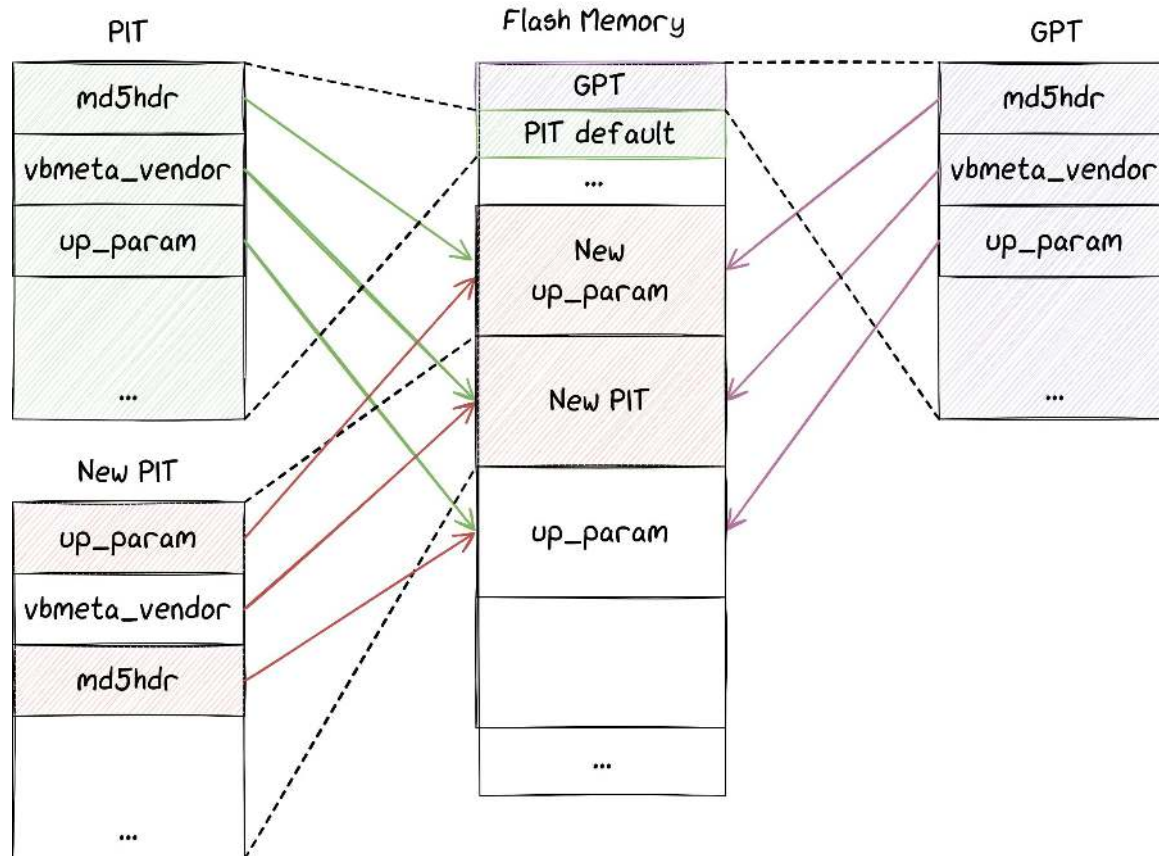
Strategy to Bypass Odin Authentication



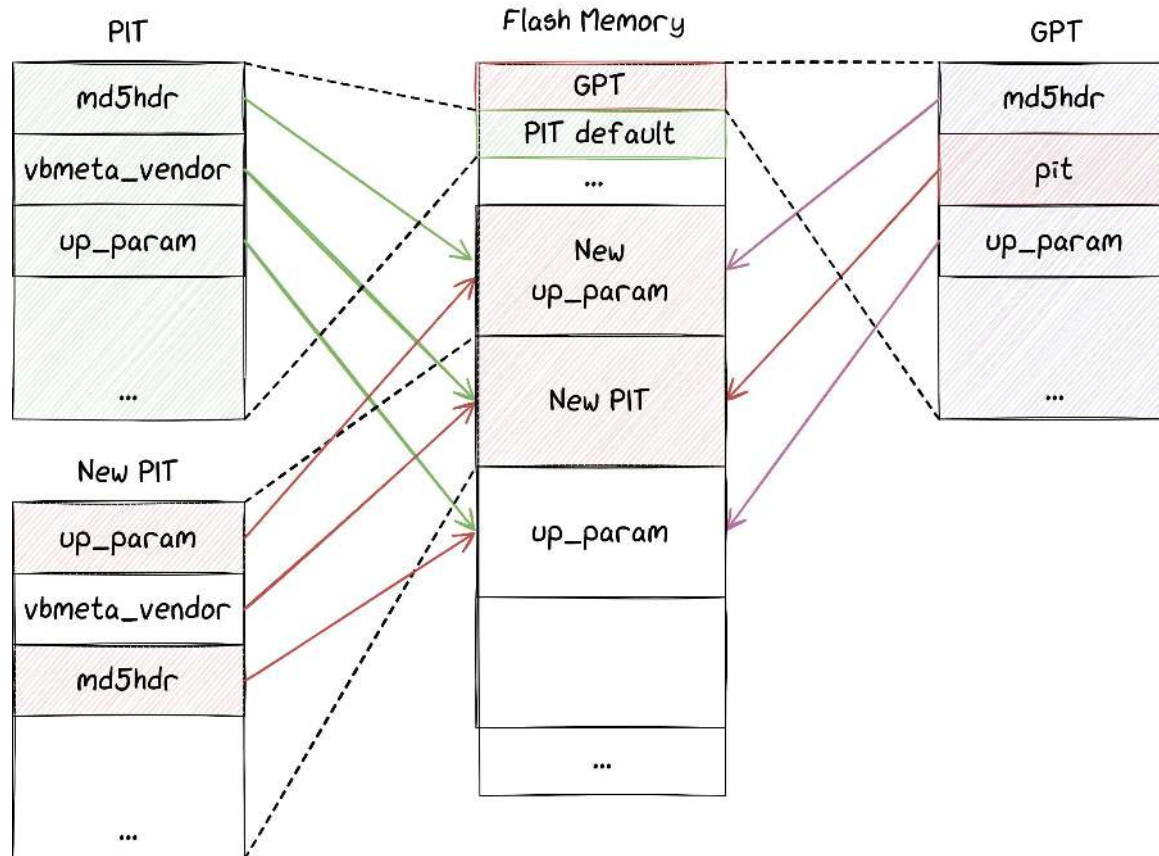
Strategy to Bypass Odin Authentication



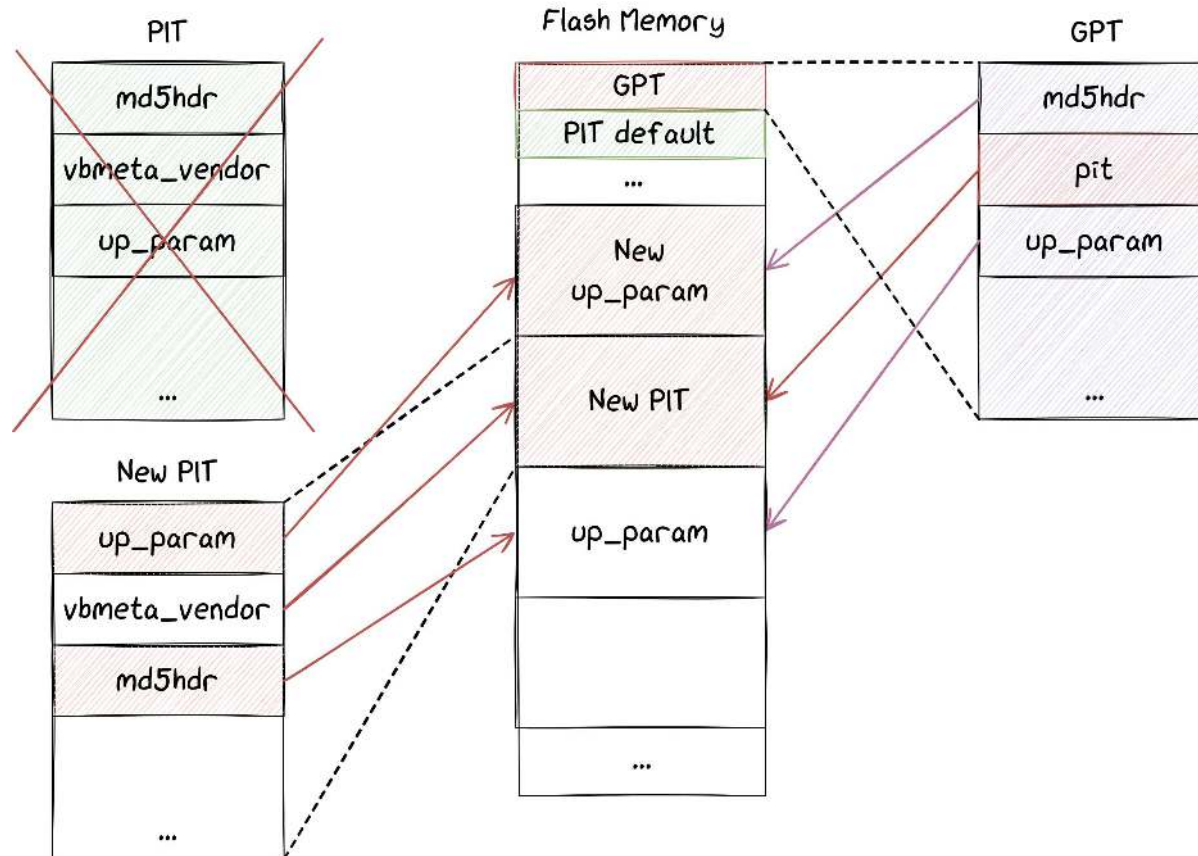
Strategy to Bypass Odin Authentication



Strategy to Bypass Odin Authentication



Strategy to Bypass Odin Authentication

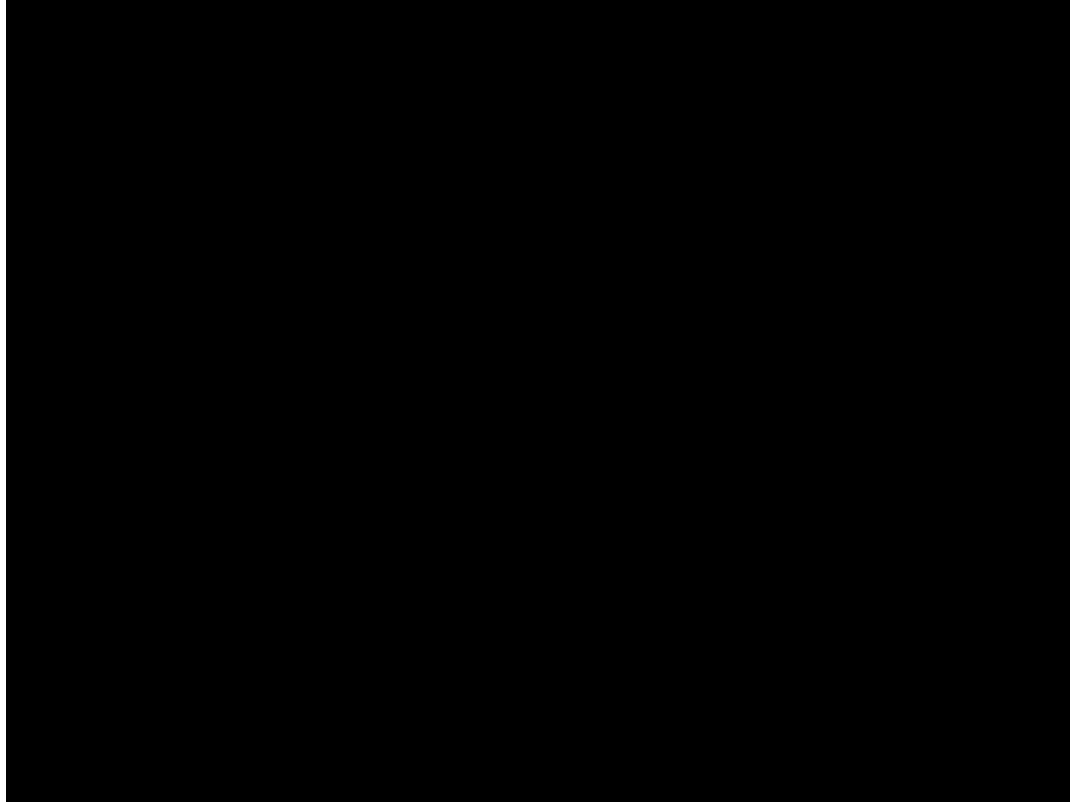


To sum up

- SVE-2024-0234/CVE-2024-20865
 - ✓ Can bypass authentication in Odin
 - ✓ We can flash anything in the eMMC
 - ✓ Including our *up_param* partition
 - ✓ Seems to impact most Samsung using Mediatek SoCs



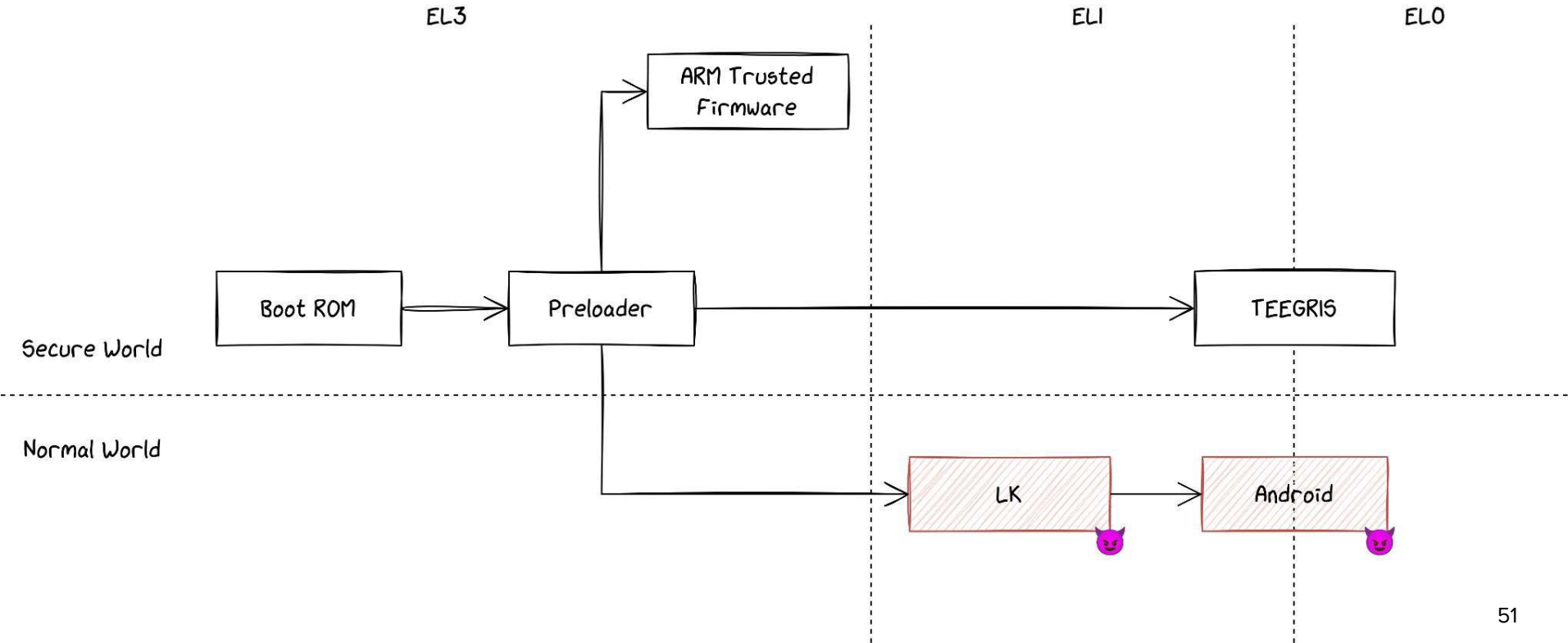
Chaining Everything Together



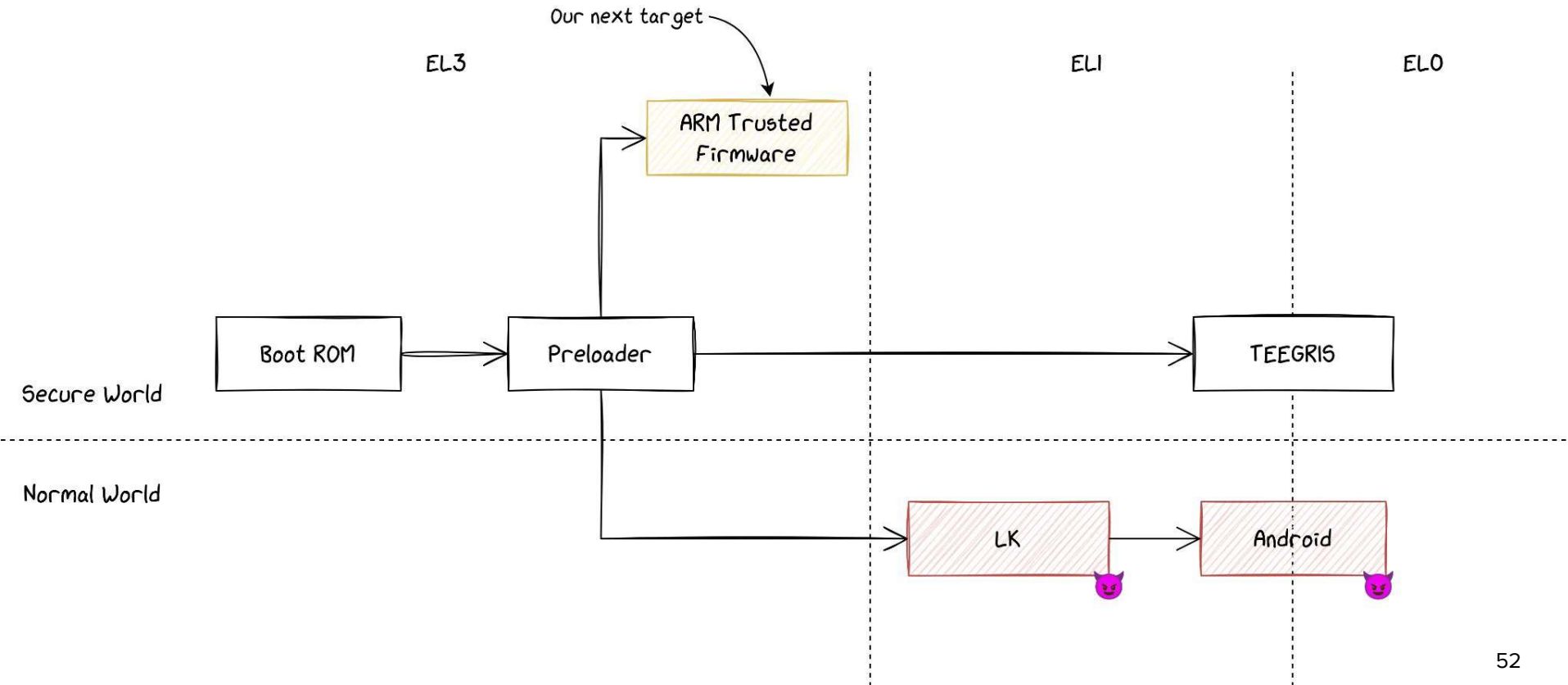
To Conclude

- Chain based on 2 vulnerabilities
 - ✓ Leads to code execution in LK
 - ✓ Persistent (it survives reboots and factory reset)
 - ✓ Impacts Samsung devices based on Mediatek SoCs
 - Including those for which MTKClient does not work
 - ✓ Can be triggered over USB thanks to Odin authentication bypass
 - ✓ Gives full control over Normal World EL1/0
 - ✗ Still no access to secrets stored in Secure World

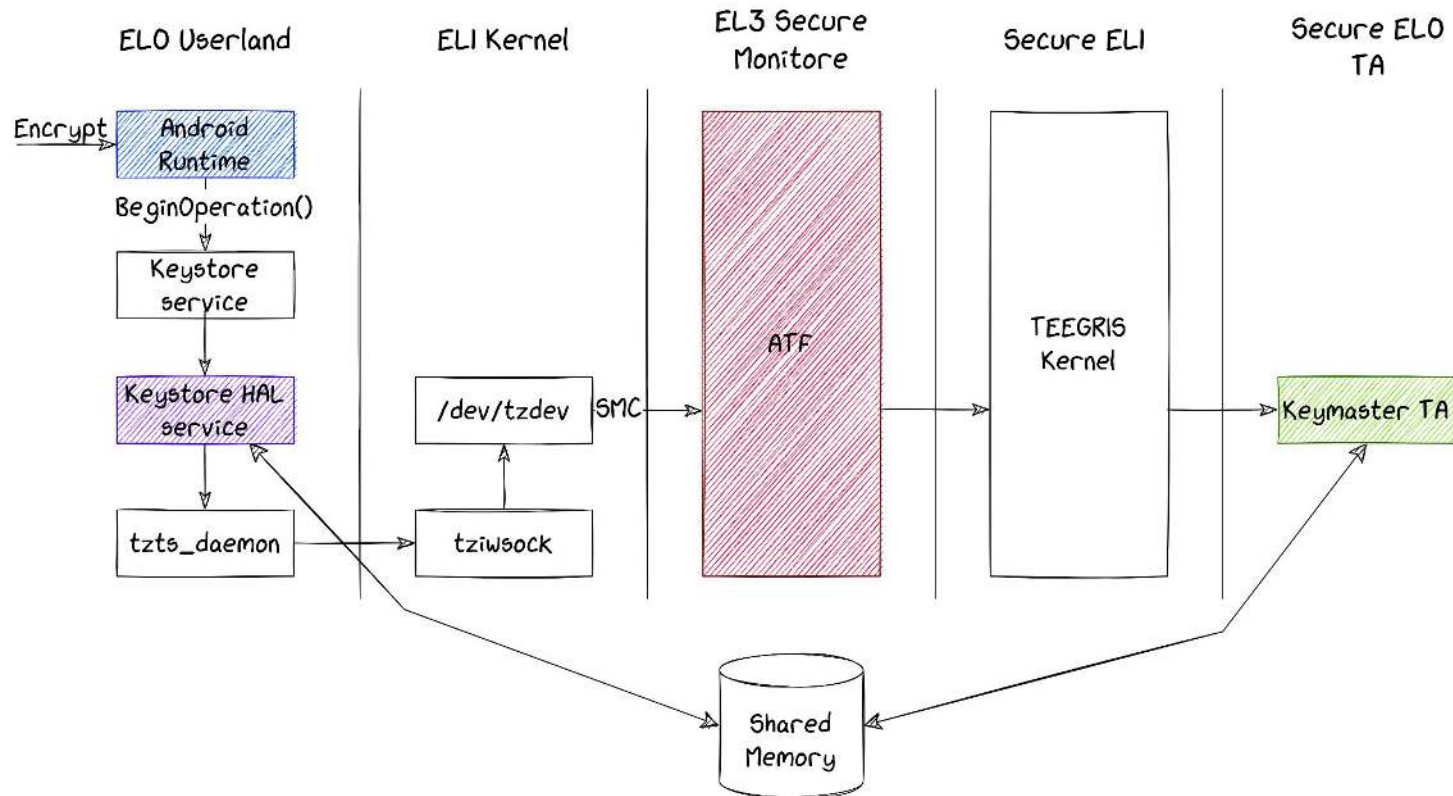
Targeting ARM Trusted Firmware



Targeting ARM Trusted Firmware



Communication between NSW and SW



Vulnerability Research on ATF

- Motivation:
 - Highest privilege level → A bug here can be devastating
 - Reachable from Normal World through SMCs
- Code is simple
- Interacts a lot with HW through unknown registers
 - Fuzzing not particularly interesting in this case
- Our approach: focus on static analysis

Extracting ATF

- Inside an Android ROM Image:
 - tee-verified.img: **ATF**, TEEGRIS kernel, userboot.so...

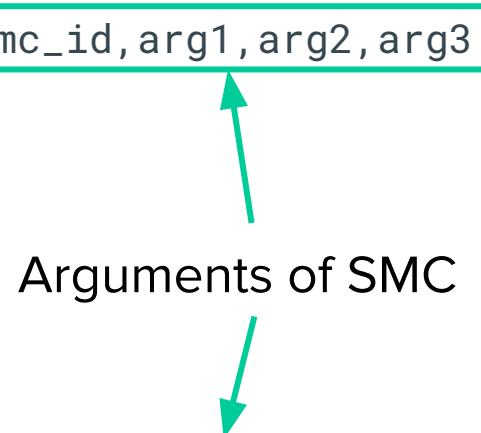
```
00000000  88 16 88 58 00 6c 02 00 61 74 66 00 00 00 00 00 |...X.l..atf....|
00000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
00000020  00 00 00 00 00 00 00 00 ff ff ff ff 00 00 00 00 |.....|
00000030  89 16 89 58 00 02 00 00 01 00 00 00 00 00 00 00 |...X.....|
00000040  00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 |.....|
00000050  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff |.....|
...
000001f0  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff |.....|
00000200  45 45 54 20 4b 54 4d 20 40 02 00 00 01 00 01 30 |EET KTM @.....0|
00000210  01 00 00 00 c0 69 02 00 c0 69 02 00 00 00 00 00 |....i...i.....|
00000220  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
...
00026df0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
00026e00  88 16 88 58 ad 06 00 00 63 65 72 74 31 00 00 00 |...X....cert1...|
```

SMC Handlers

```
if ((is_secure & 1) == 0) {  
    puVar1 = mediatek_plat_sip_handler_secure(smc_id, arg1, arg2, arg3  
                                              , arg4, arg5, output);  
    return puVar1;  
}  
[...]  
if ((origin < 2) && (IN_BOOTLOADER == 0)) {  
    puVar1 = mediatek_plat_sip_handler_kernel(smc_id, arg1, arg2, arg3  
                                              , arg4, arg5, output);  
    return puVar1;  
}
```


SMC Handlers

```
if ((is_secure & 1) == 0) {  
    puVar1 = mediatek_plat_sip_handler_secure(smc_id, arg1, arg2, arg3  
                                              , arg4, arg5, output);  
    return puVar1;  
}  
[...]  
if ((origin < 2) && (IN_BOOTLOADER == 0)) {  
    puVar1 = mediatek_plat_sip_handler_kernel(smc_id, arg1, arg2, arg3  
                                              , arg4, arg5, output);  
    return puVar1;  
}
```



Arguments of SMC

The diagram illustrates the arguments of the SMC (Secure Monitor Call) function. It features two green rectangular boxes, one around the arguments of the `mediatek_plat_sip_handler_secure` function and another around the arguments of the `mediatek_plat_sip_handler_kernel` function. A green arrow points from the text "Arguments of SMC" to the first box, and another green arrow points from the same text to the second box.

Leaking from Virtual Address Space

```
uint* global_array = (uint *)0x4ce2f578;
[...]  
if (smcid == 0x82000526) {  
    out_value = global_array[arg1 * 4];  
    goto exit;  
}  
[...]  
    output[2] = out_value;  
    output[1] = arg1;  
    *output = 0;  
    return output;
```

Leaking from Virtual Address Space

```
uint* global_array = (uint *)0x4ce2f578;
```

```
[...]
```

```
if (smcid == 0x82000526) {
```

```
    out_value = global_array[arg1 * 4];
```

```
    goto exit;
```

```
}
```

```
[...]
```

```
    output[2] = out_value;
```

```
    output[1] = arg1;
```

```
    *output = 0;
```

```
    return output;
```



Fully controlled by
attacker

Leaking from Virtual Address Space

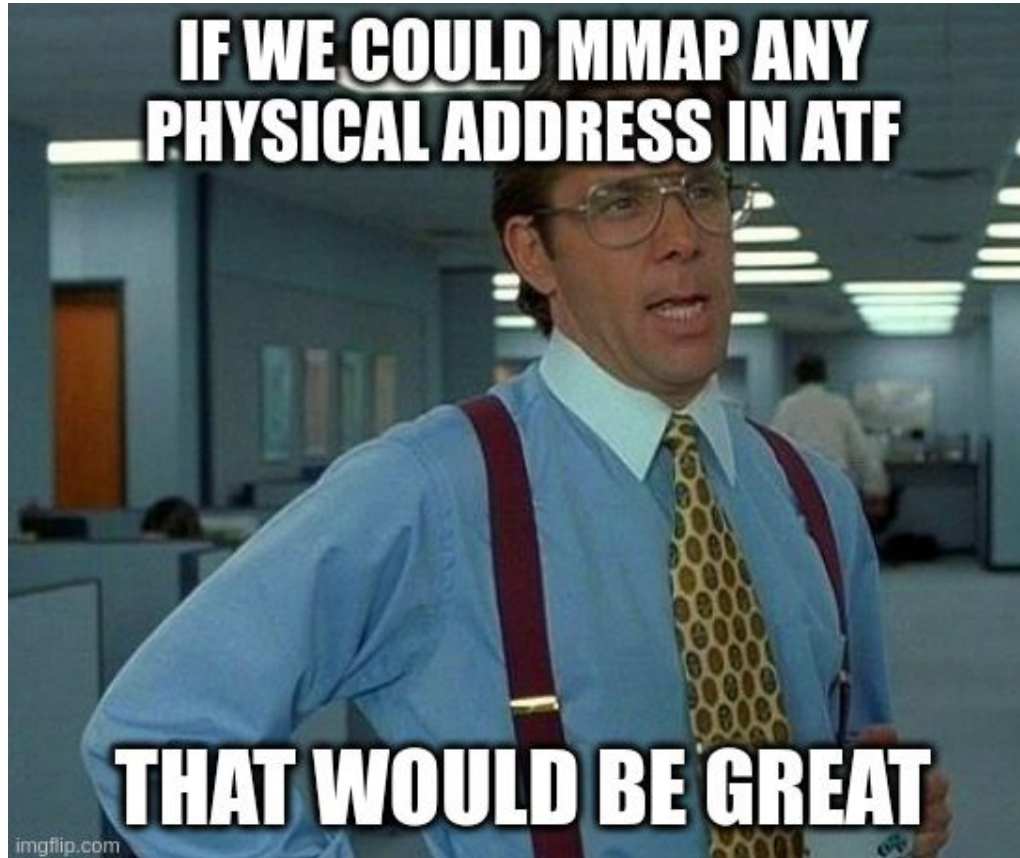
```
uint* global_array = (uint *)0x4ce2f578;  
[...]  
if (smcid == 0x82000526) {  
    out_value = global_array[arg1] * 4;  
    goto exit;  
}  
[...]  
output[2] = out_value;  
output[1] = arg1;  
*output = 0;  
return output;
```

Fully controlled by
attacker... And never
checked



SVE-2023-2215 (CVE-2024-20820)

- In `mediatek_plat_sip_handler_kernel`, reachable from Linux Kernel
- To exploit it, send the SMC `0x82000526` with
 - `(arbitrary_address - 0x4ce2f578) / 4`
- Bug introduced by Samsung only in some devices (including A225F)
- It leaks 4 bytes from ATF virtual address space
 - We can read all the internal data of ATF
 - But we can't leak anything from other SW components



Mapping Any Physical Address in ATF

SMC 0x8200022A calls function `spm_actions`

```
if (smc_id == 0x8200022a) {  
    spm_actions(arg1, arg2, arg3);  
}
```

Mapping Any Physical Address in ATF

SMC 0x8200022A calls function `spm_actions`

```
undefined * spm_actions(ulong cmdid, undefined *addr, ulong size) {  
    switch(cmdid & 0xffffffff) {  
[...]  
        case 1:  
            if (size < 0x100001) {  
                mmap_wrap(addr, size);  
[...]  
            }  
    }
```


Mapping Any Physical Address in ATF

SMC 0x8200022A calls function `spm_actions`

```
undefined * spm_actions(ulong cmdid, undefined *addr, ulong size) {  
    switch(cmdid & 0xffffffff) {  
[...]  
        case 1:  
            if (size < 0x100001) {  
                mmap_wrap(addr, size);  
[...]  
            }  
    }
```

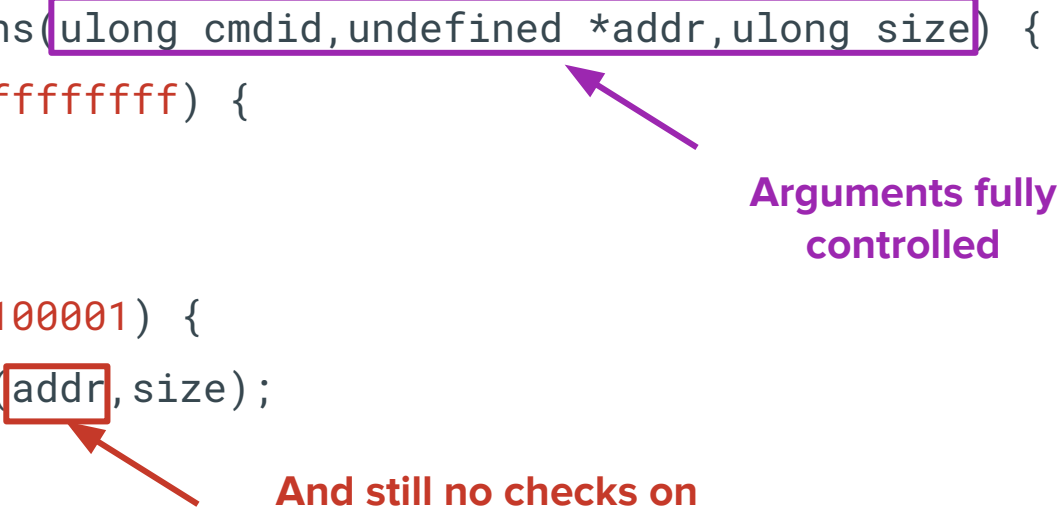


**Arguments fully
controlled**

Mapping Any Physical Address in ATF

SMC 0x8200022A calls function `spm_actions`

```
undefined * spm_actions(ulong cmdid, undefined *addr, ulong size) {  
    switch(cmdid & 0xffffffff) {  
[...]  
        case 1:  
            if (size < 0x100001) {  
                mmap_wrap(addr, size);  
[...]  
            }  
    }
```



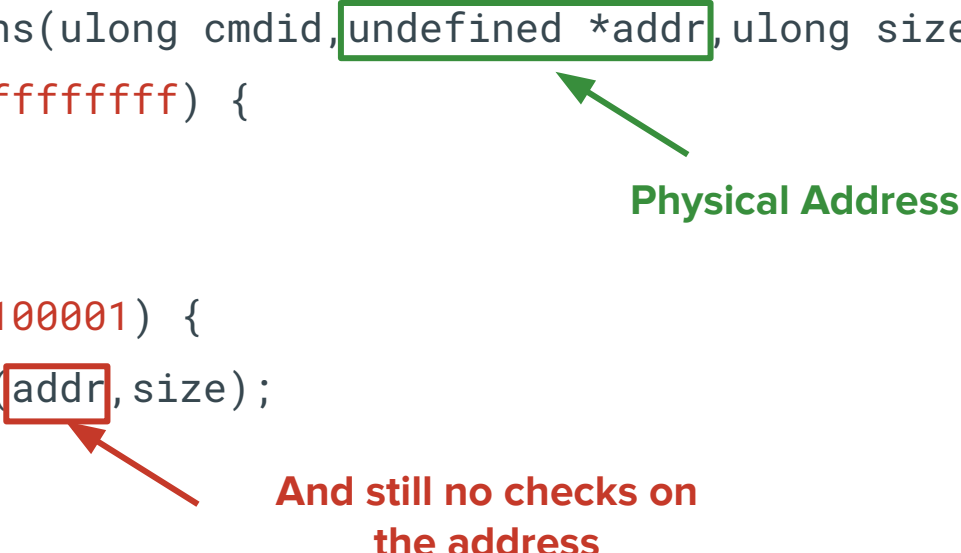
Arguments fully controlled

And still no checks on the address

Mapping Any Physical Address in ATF

SMC 0x8200022A calls function `spm_actions`

```
undefined * spm_actions(ulong cmdid, undefined *addr, ulong size) {  
    switch(cmdid & 0xffffffff) {  
[...]  
        case 1:  
            if (size < 0x100001) {  
                mmap_wrap(addr, size);  
[...]  
            }  
    }
```



Physical Address

And still no checks on the address

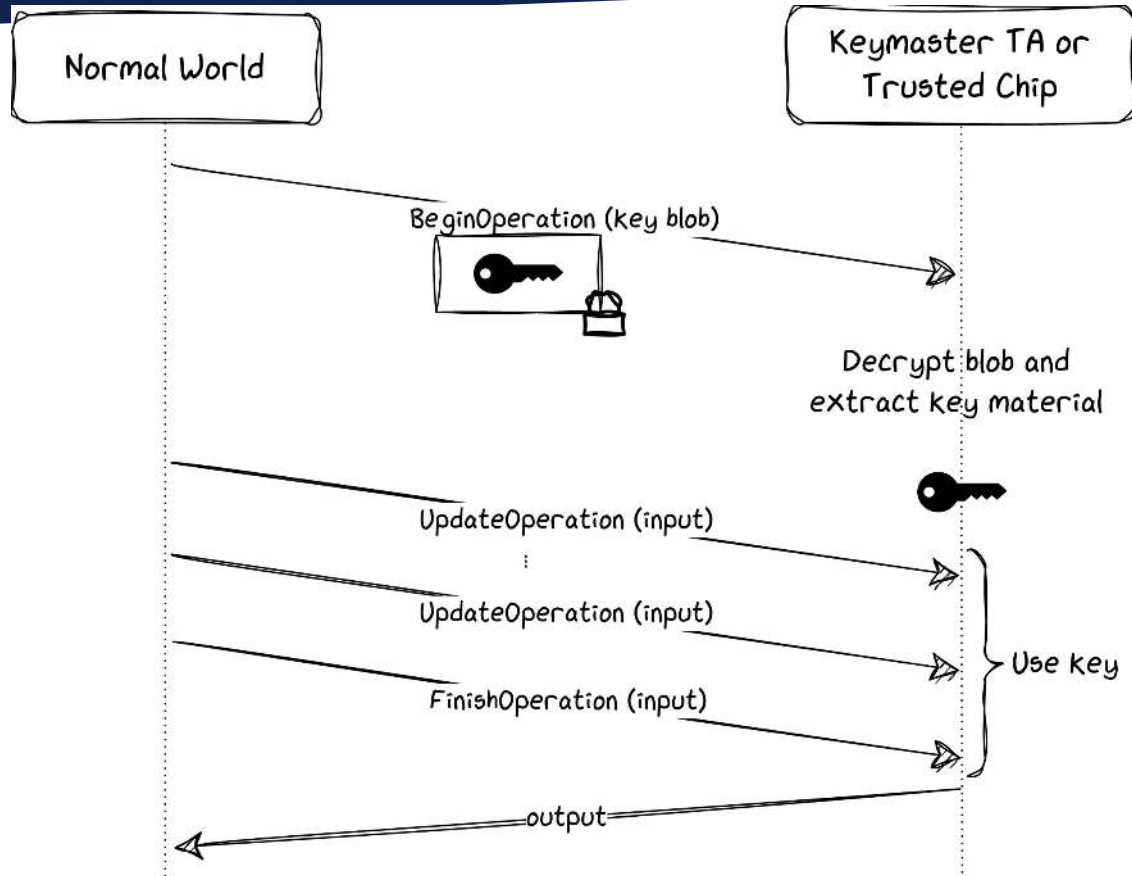
- Also in mediatek_plat_sip_handler_kernel
- Will mmap with physical base address to the same virtual address
 - ... however we can't munmap
 - So we are limited to 8 consecutive mmaps
 - Meaning we can leak up to **8MB** of data
- Introduced by Mediatek (impacts plenty of Mediatek SoCs)
- Chained to our leak, we can read everything in Secure World
 - Including TEEGRIS

*Can we use this vulnerability to leak
Keystore keys?*

Android Keystore system

- Key storage and crypto services
- Keys are stored as encrypted *key blobs*
- Three protection levels:
 - Software only
 - TEE (default)
 - Hardware-backed (StrongBox)
- Raw key should never leave protected environment

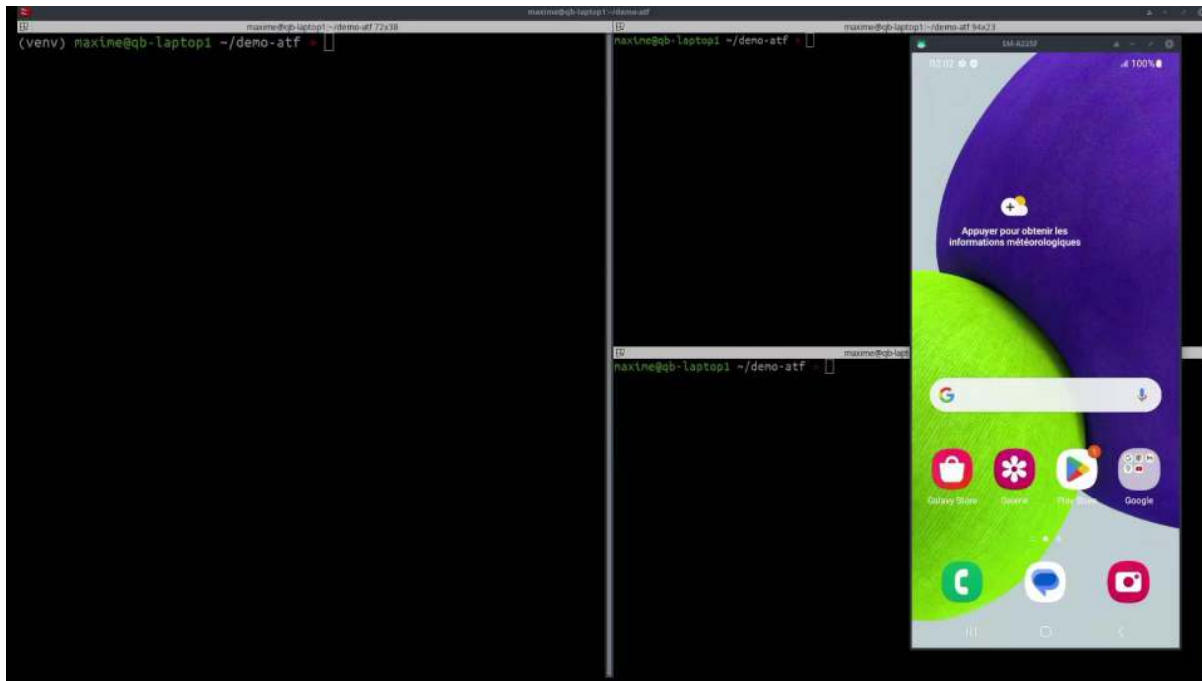
Android Keystore system



Our PoC

1. **Import** a key into the Android Keystore
2. **Encrypt** using that key
3. **Stop the execution** after BeginOperation is called
 - To make sure the key stays in memory
4. **Leak** the identified region of memory
5. Try all possible keys from leak to decrypt ciphertext

Demo



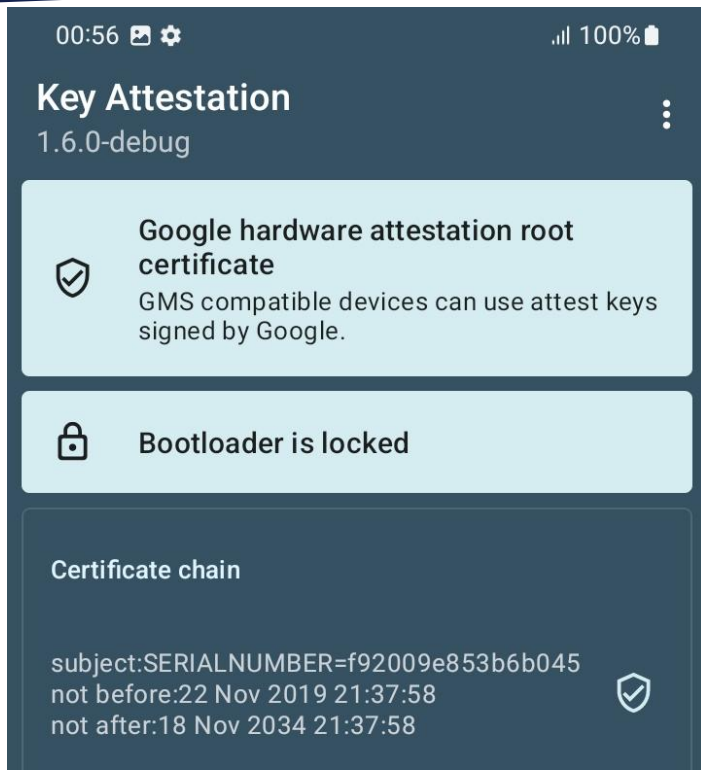
What's next?

Key Attestation

- Proves that a key pair is stored in the secure hardware
 - Trustzone or Security Chip
- Contains information about the device state
 - Such as bootloader locked status and verified boot state
- Used by SafetyNet⁵ to tell if a device has been compromised

[5]: https://www.sstic.org/media/SSTIC2022/SSTIC-actes/droidguard_a_deep_dive_into_safetynet/SSTIC2022-Article-droidguard_a_deep_dive_into_safetynet-thomas.pdf

First Key Attestation test



Attestation generated with a demo app⁶



Our exploit seems not detected!

[6]: <https://github.com/vvb2060/KeyAttestation>

What about SafetyNet?

Device

Model

SM-A225F (a22)

Android version

13 (API 33)

Security patch level

2023-06-01

Google Play Services version

22.46.17 (190400-491726958)

Result

Basic integrity

Fail 

CTS profile match

Fail 

Evaluation type

BASIC

Timestamp

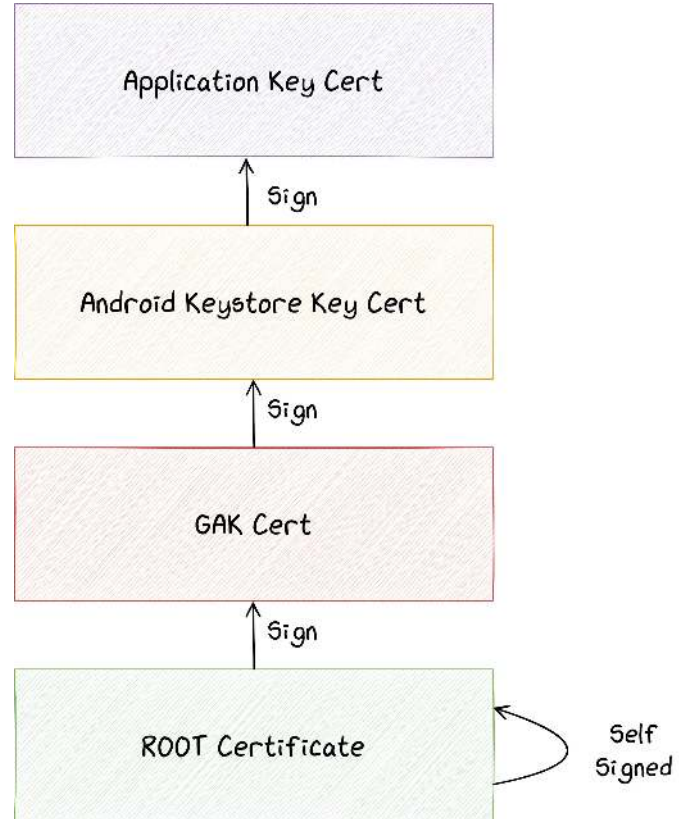
6 Aug 2024 03:12:29



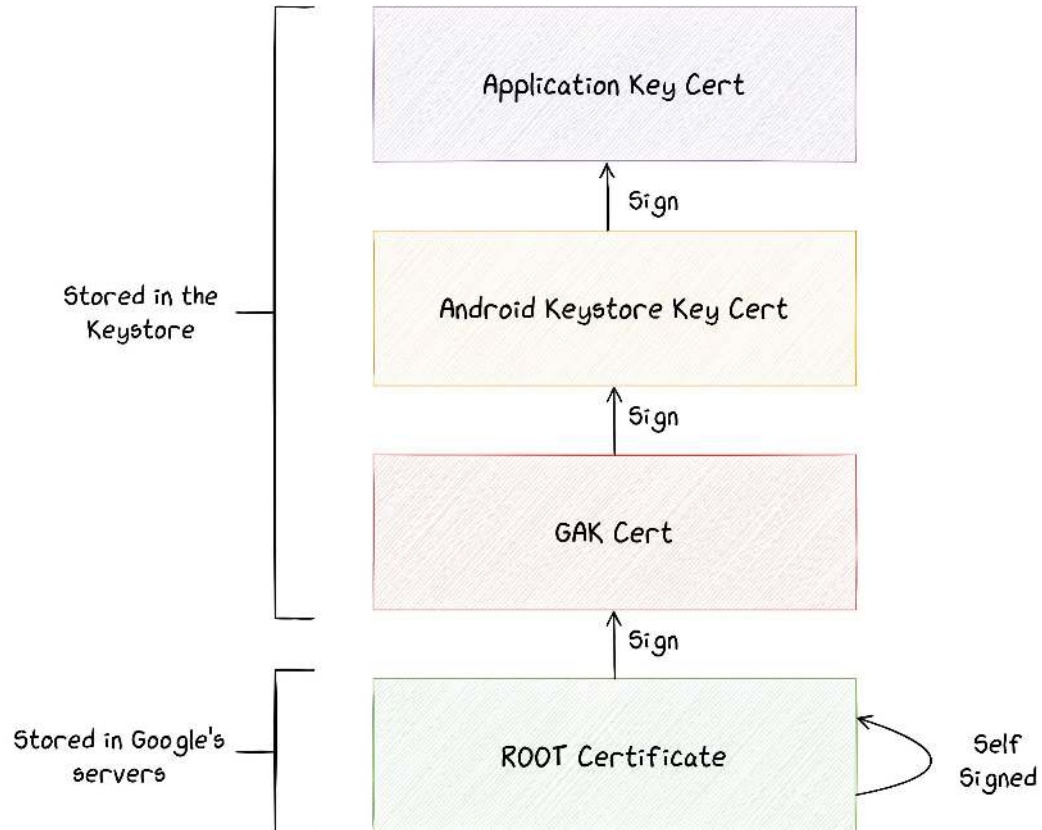
SafetyNet detects the exploit

- Possibly through heuristics to detect Magisk

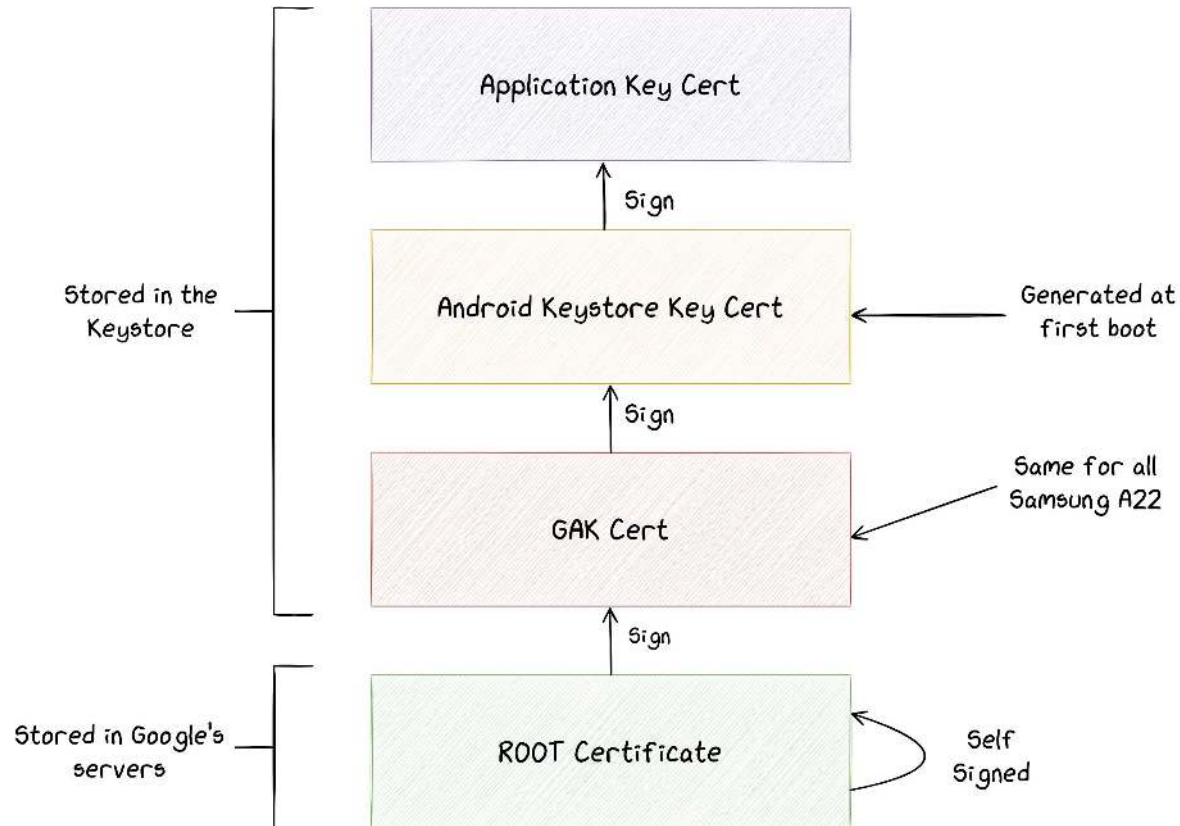
Certificate Chain



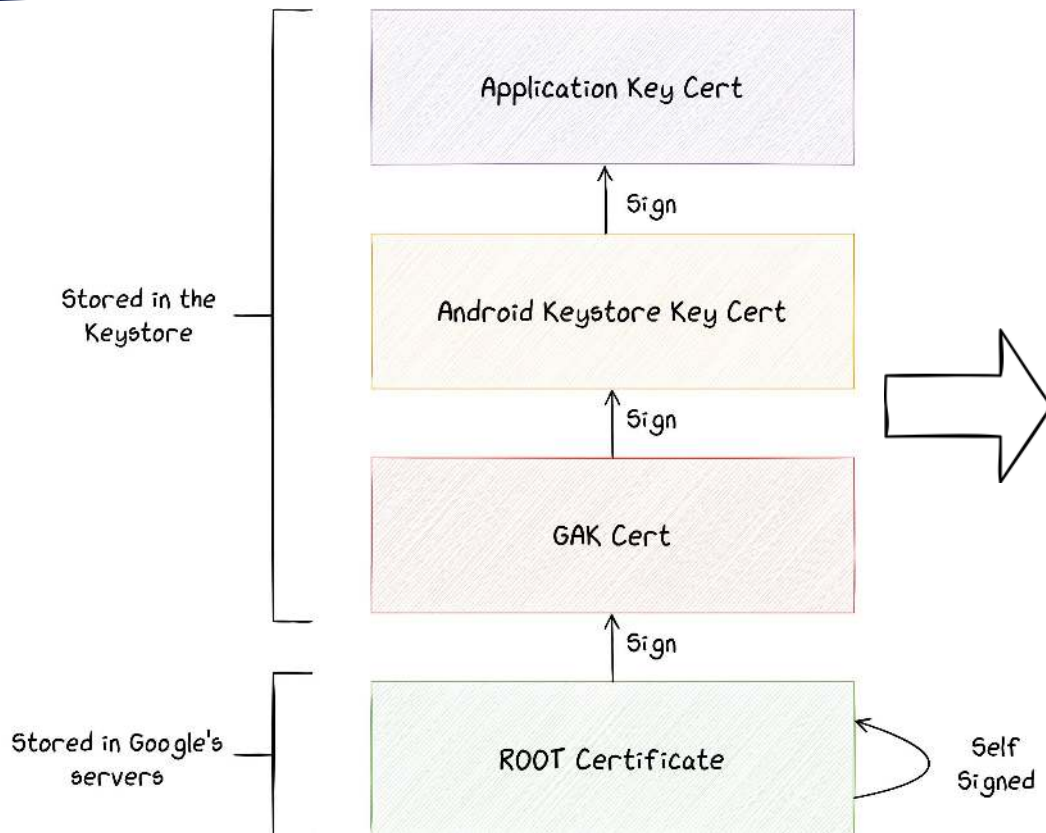
Certificate Chain



Certificate Chain



Certificate Chain



Root of trust

verifiedBootKey:
07EFE057CEBF085AAA79D1FB2090CCCB59
9D6A51B027FEABF123FD78E84D3D63
deviceLocked: true
verifiedBootState: Verified
verifiedBootHash:
E07ADF185A36CE4A2FED204CF9C3103D54
1B7A90305F89785154A460688553E0

OS version

130000

OS patch level

202306

Attestation application ID

Package info 1/1:
io.github.vvb2060.keyattestation (version
code 165)

Certificate sha256 digest 1/1:
D8B53693490D6E7467F985165001A547F72
2E75735B34E6AA13BF949F5216F1E

Strategy to leak GAK

- Stored as EKEY in Android Filesystem
 - `/mnt/vendor/efs/DAK/GAK_EC.private`

Strategy to leak GAK

- Stored as EKEY in Android Filesystem
 - `/mnt/vendor/efs/DAK/GAK_EC.private`
- 1. **Forge a valid** Begin request with GAK keyblob
- 2. **Stop the execution** after BeginOperation is called
- 3. **Leak** memory (as in previous PoC)
- 4. Try to every possible private keys in the dump
 - By generating the public key out of it

Strategy to leak GAK

- Stored as EKEY in Android Filesystem
 - `/mnt/vendor/efs/DAK/GAK_EC.private`
 - 1. **Forge a valid** Begin request with GAK keyblob
 - 2. **Stop the execution** after BeginOperation is called
 - 3. **Leak** memory (as in previous PoC)
 - 4. Try to every possible private keys in the dump
 - By generating the public key out of it
- Still WIP

Strategy to leak GAK

- Stored as EKEY in
 - /mnt/vendor/...
- 1. **Forge a valid** Beg
- 2. **Stop the executio**
- 3. **Leak** memory (as
- 4. Try to every possi
 - By generating

→ Still WIP



Conclusion

- We presented 4 vulnerabilities leading to
 - Authentication bypass in Odin
 - Code execution with persistence in LK
 - Leak of SW memory, including Keystore keys
 - Still unclear if we can leak Attestation Keys
- Impact low/middle end Samsung devices
 - Vulnerabilities are simple, and yet super impactful
 - No mitigations in LK nor ATF
- All the vulnerabilities are now fixed

Thank you!

contact@quarkslab.com

Quarkslab



@max_r_b

@DamianoMelotti

@pwissenlit