Attacking Samsung Galaxy A* Boot Chain

Maxime Rossi Bellom
Damiano Melotti
Raphael 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

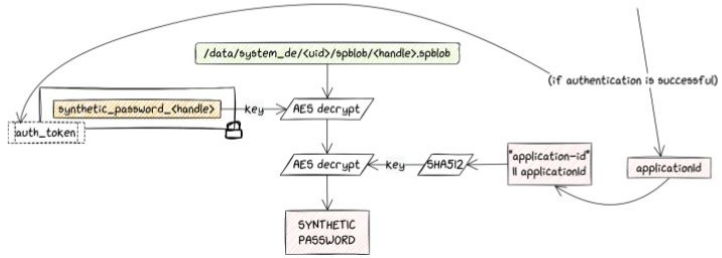
Bruteforce of the password

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)



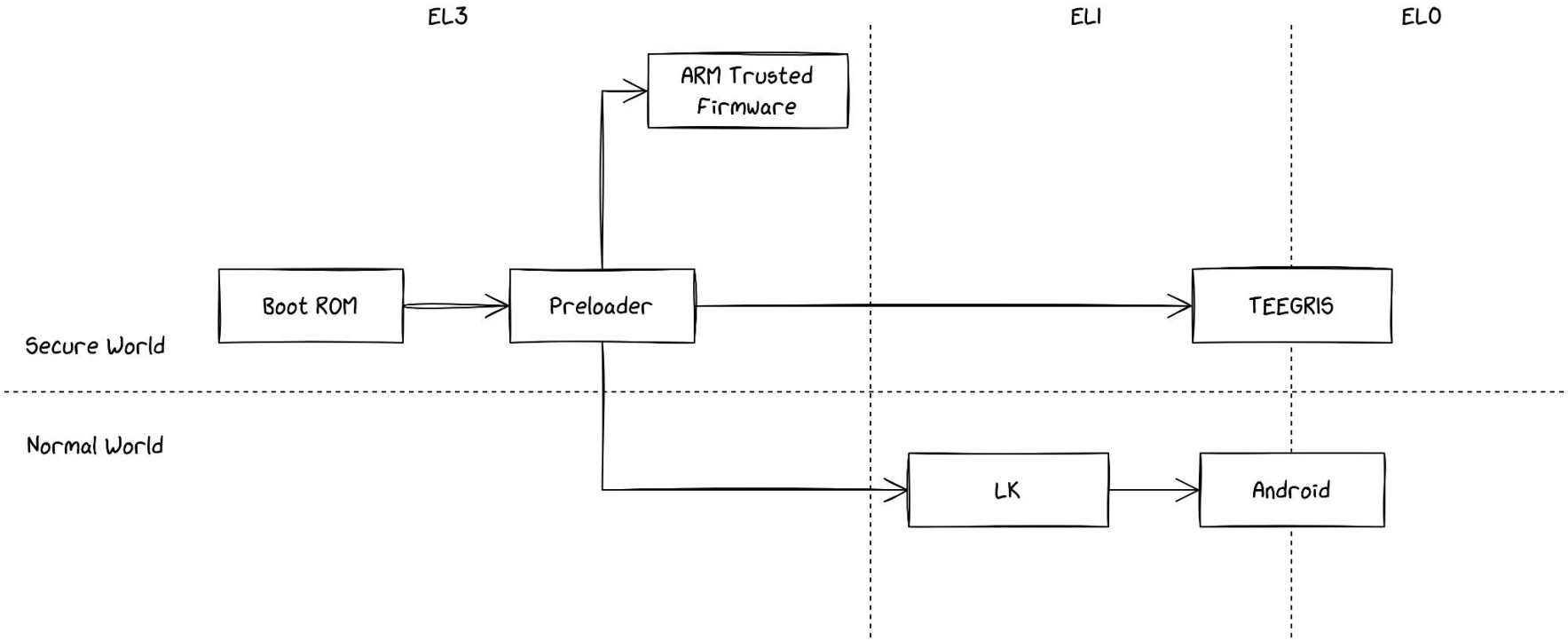

```
Preloader -     HW version:             0x0
Preloader -     WDT:                    0x10007000
Preloader -     Uart:                   0x11002000
Preloader -     Brom payload addr:      0x100a00
Preloader -     DA payload addr:        0x201000
Preloader -     CQ_DMA addr:            0x10212000
Preloader -     Var1:                   0x25
Preloader - Disabling Watchdog...
Preloader - HW code:                    0x707
Preloader - Target config:              0xe5
Preloader -     SBC enabled:            True
Preloader -     SLA enabled:            False
Preloader -     DAA enabled:            True
Preloader -     SWJTAG enabled:         True
Preloader -     EPP_PARAM at 0x600 after EMMC_BOOT/SDMMC_BOOT:  False
Preloader -     Root cert required:     False
Preloader -     Mem read auth:          True
Preloader -     Mem write auth:         True
Preloader -     Cmd 0xC8 blocked:       True
Preloader - Get Target info
Preloader - BROM mode detected.
Preloader -     HW subcode:             0x8a00
Preloader -     HW Ver:                 0xca00
Preloader -     SW Ver:                 0x0
Preloader - ME_ID:                      34C0B89C3AC60179BFB7015S591927F9
Preloader - SOC_ID:                     8EDADE25C1C71F2C4BC41DE3D879F3DC0D2348AC1C0CBFE8DCDF336568D3F18D
PLTools - Loading payload from mt6768_payload.bin, 0x264 bytes
PLTools - Kamakiri / DA Run
Kamakiri - Trying kamakiri2..
Kamakiri - Done sending payload...
PLTools - Successfully sent payload: /home/maxime/tools/mtkclient/mtkclient/payloads/mt6768_payload.bin
Port - Device detected :)
Main - Connected to device, loading
Main - Using custom preloader : preloader_k69v1_64_titan_buffalo.bin
Mtk - Valid preloader detected.
Mtk - Patched "seclib_sec_usbdl_enabled" in preloader
Mtk - Patched "sec_img_auth" in preloader
Mtk - Patched "get_vfy_policy" in preloader
Main - Sent preloader to 0x201000, length 0x3ff24
Preloader - Jumping to 0x201000
Preloader - Jumping to 0x201000: ok.
Main - PL Jumped to daaddr 0x201000.
Main - Keep pressed power button to boot.
[+] Waiting for device to boot
```

Our Device

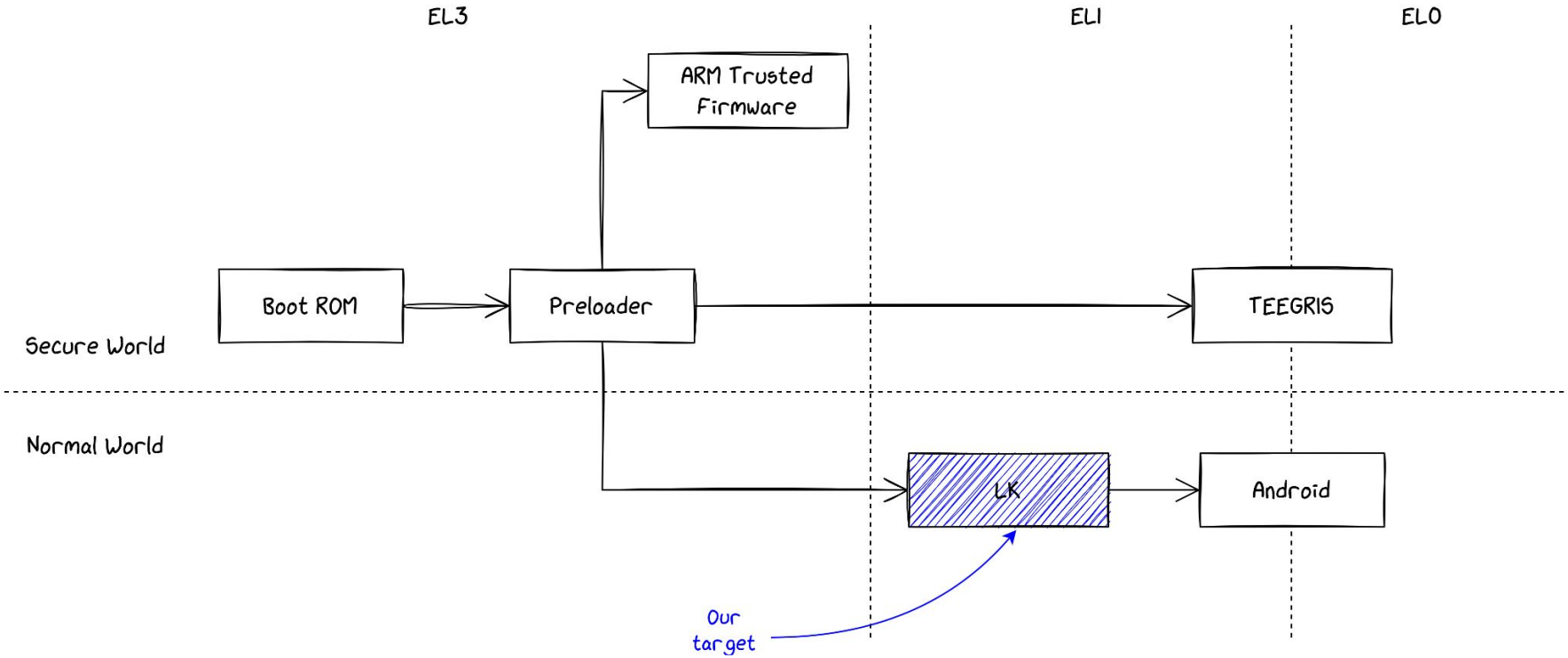
- Samsung Galaxy A225F
  - Cheap (~300€)
  - Mediatek SoC MT6769V
  - Main OS: Android
  - Mix of Mediatek and Samsung code
  - Trustzone OS: TEEGRIS
  - Secure Boot Bypass using MTKClient[1]
    - ➜ making debugging easier

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

Mediatek Secure Boot Process



Mediatek Secure Boot Process



Android partitions

- `boot.img`
  - Kernel and ramdisk
- vbmeta.img
  - For verified boot
- `super.img`
  - Dynamic partition combining `system`, `vendor`, and more
- … many more…

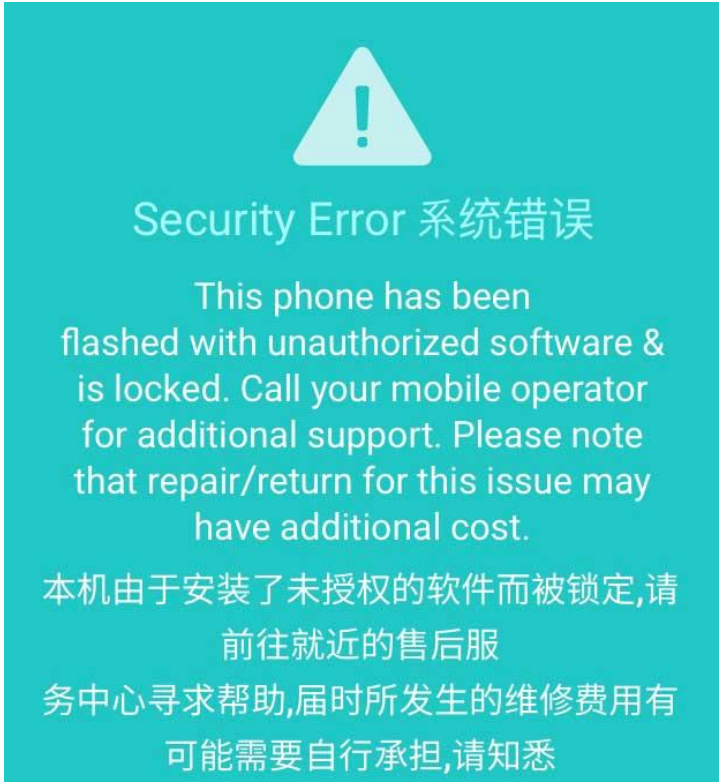
Little Kernel (LK)

- Open-source OS[2]
- 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



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. LogoFail[3])

[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. LogoFail[3])

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","drawimg");
  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","drawimg",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","drawimg");
  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","drawimg",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","drawimg");
  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","drawimg",file_name);
    uVar2 = 0xffffffff;
  }
// ...

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

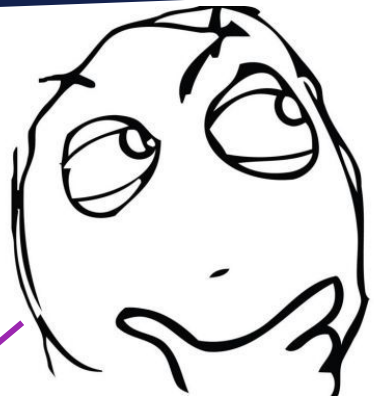
Heap Overflow in JPEG Loading

```
_JPEG_BUF = alloc(0x100000);
if (_JPEG_BUF == 0) {
  log("%s: img buf alloc fail\n","drawimg");
  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","drawimg",file_name);
    uVar2 = 0xffffffff;
  }
// ...

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

Parse and render the JPEG

Heap Overflow in JPEG Loading



```
_JPEG_BUF = alloc(0x100000);
if (_JPEG_BUF == 0) {
  log("%s: img buf alloc fail\n","drawimg");
  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","drawimg",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...

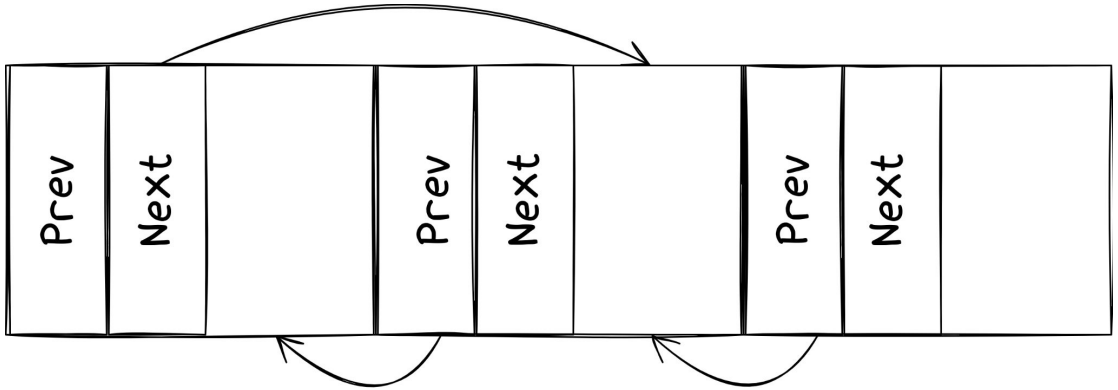
```
file_size = string_to_int(tar_header_file.size,0,8);
if (size != 0 && size < file_size) {
    file_size = print("read fail! (%d < %d)\n",size,file_size,size);
    return file_size;
}
iVar1 = read(data_addr,index + 1,file_size,outbuf);
```

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 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

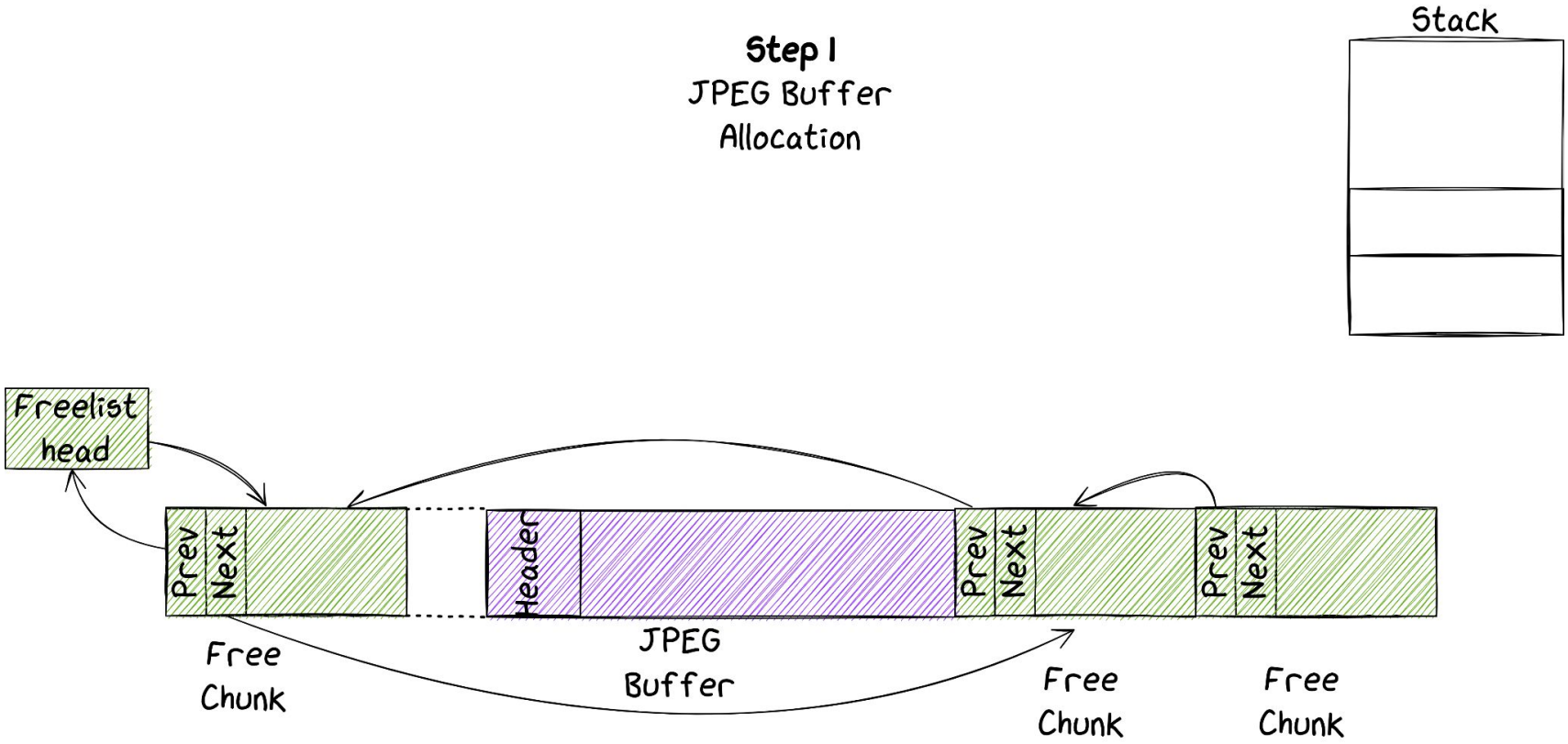
Important details about LK

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

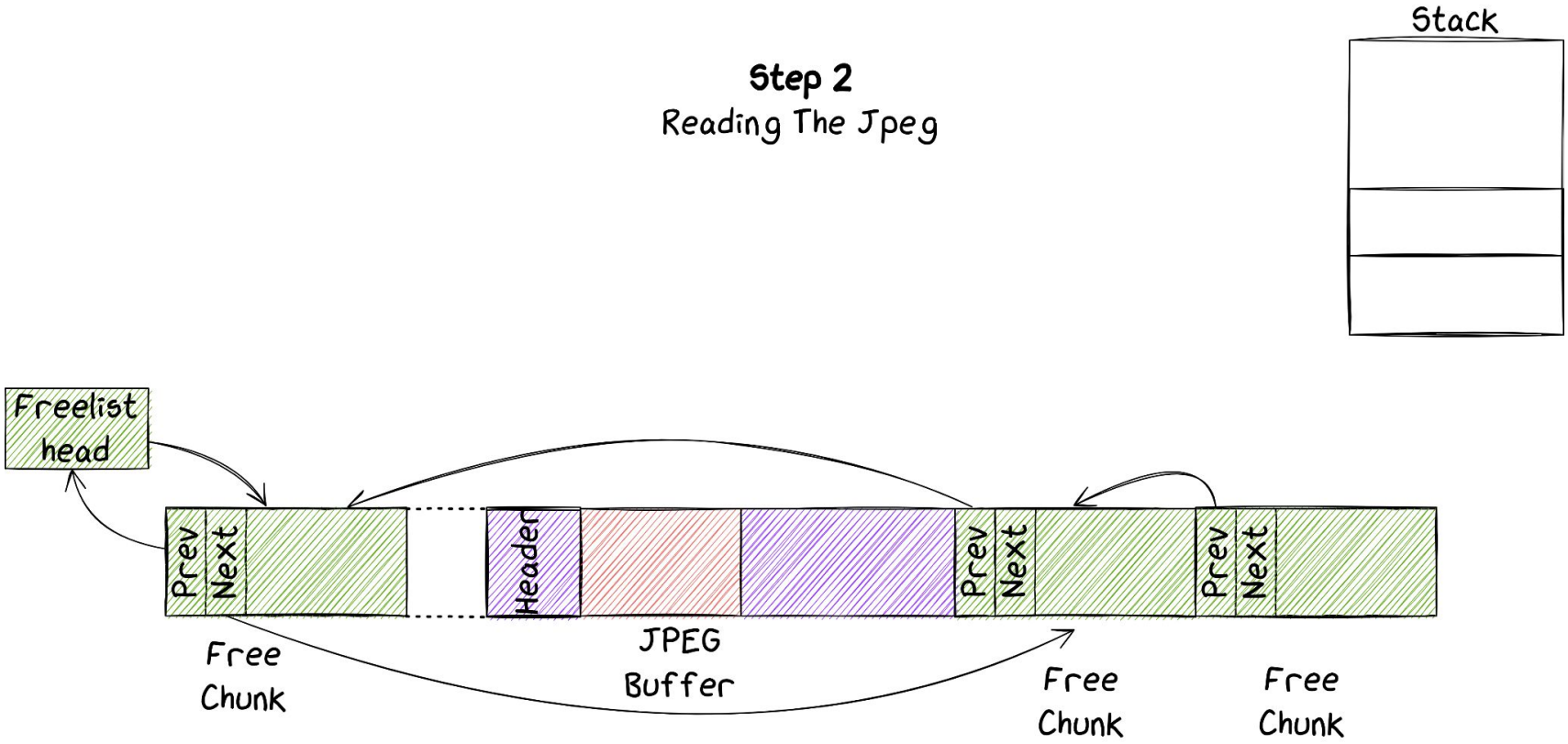
Exploit strategy becomes simple:

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

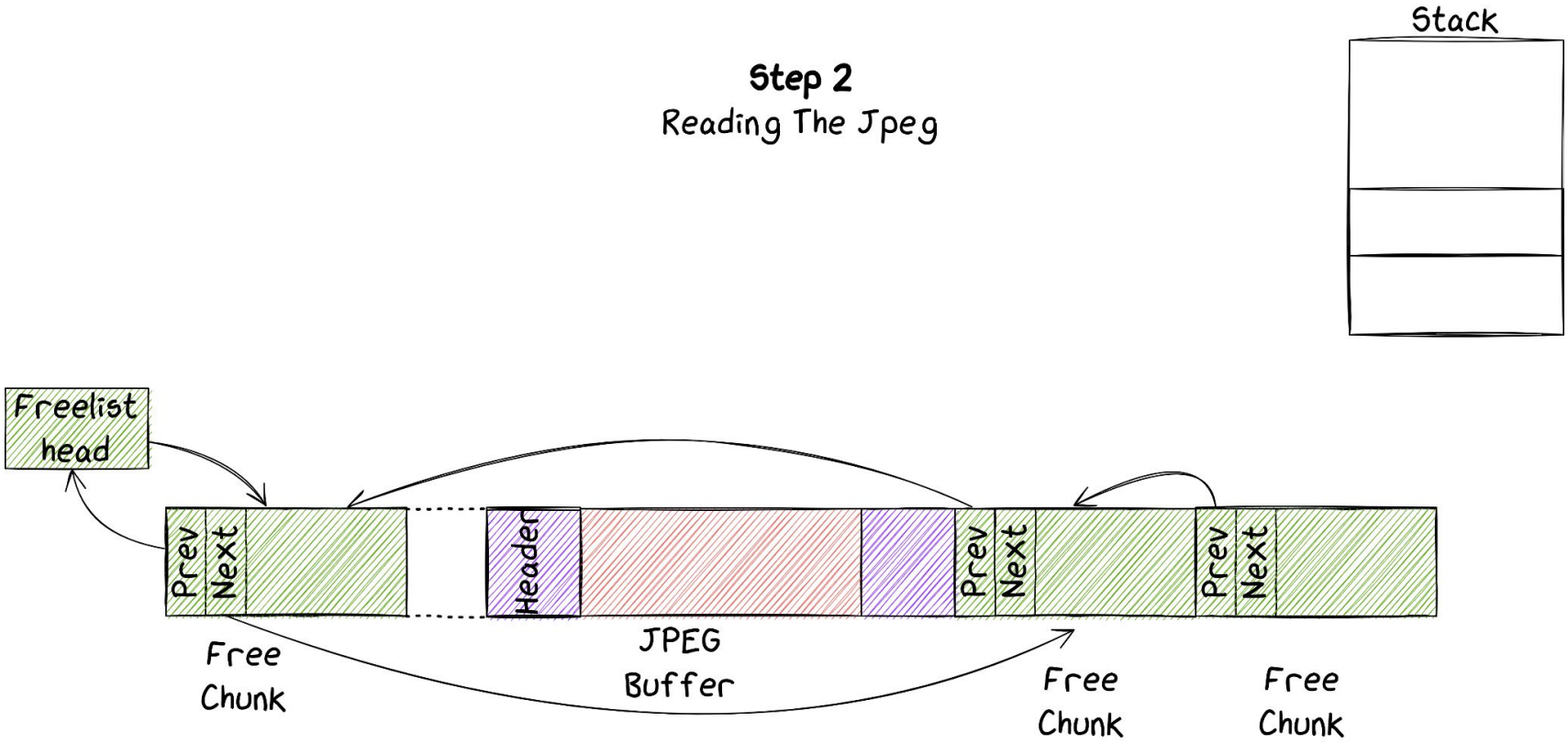
Exploiting a Heap Overflow in Little Kernel



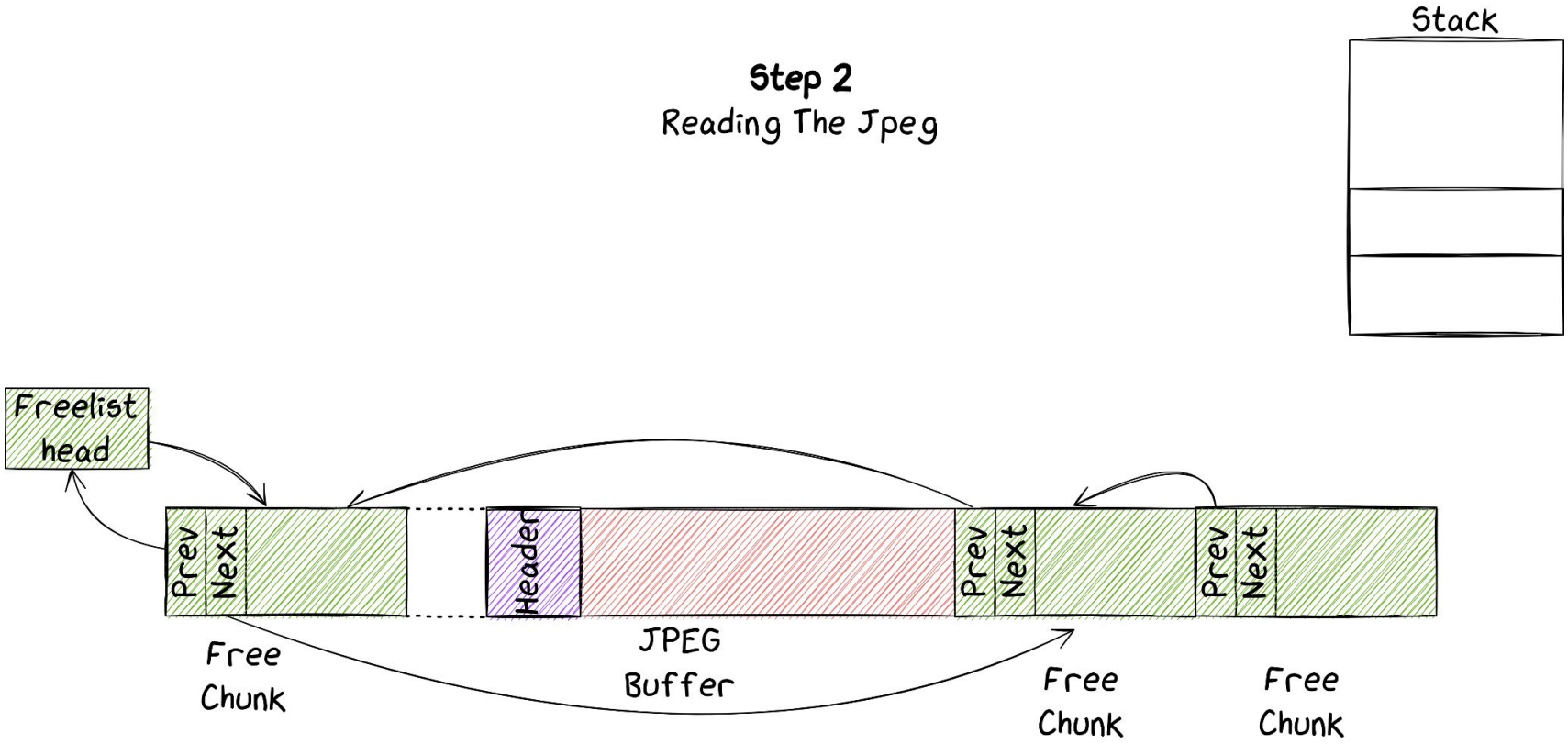
Exploiting a Heap Overflow in Little Kernel



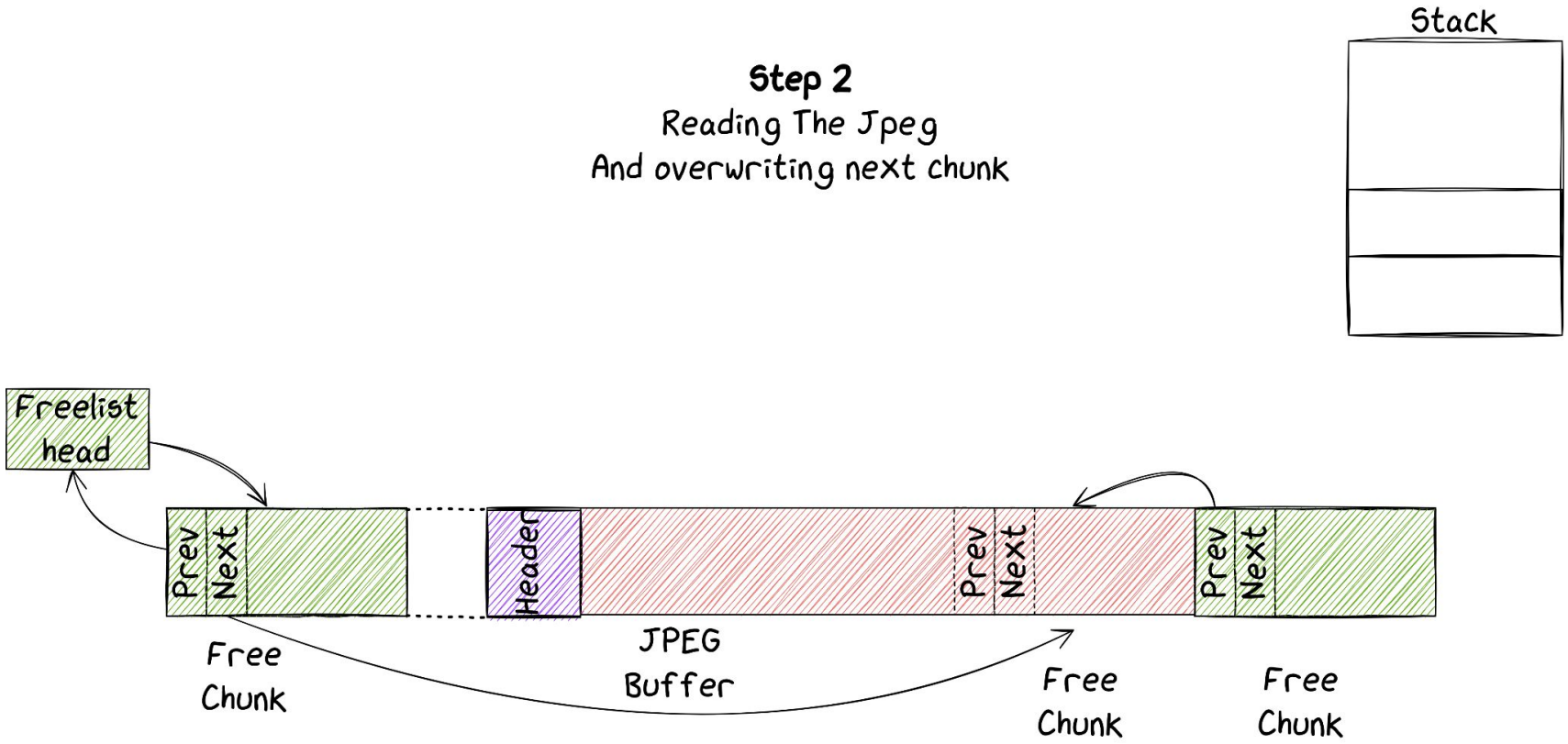
Exploiting a Heap Overflow in Little Kernel



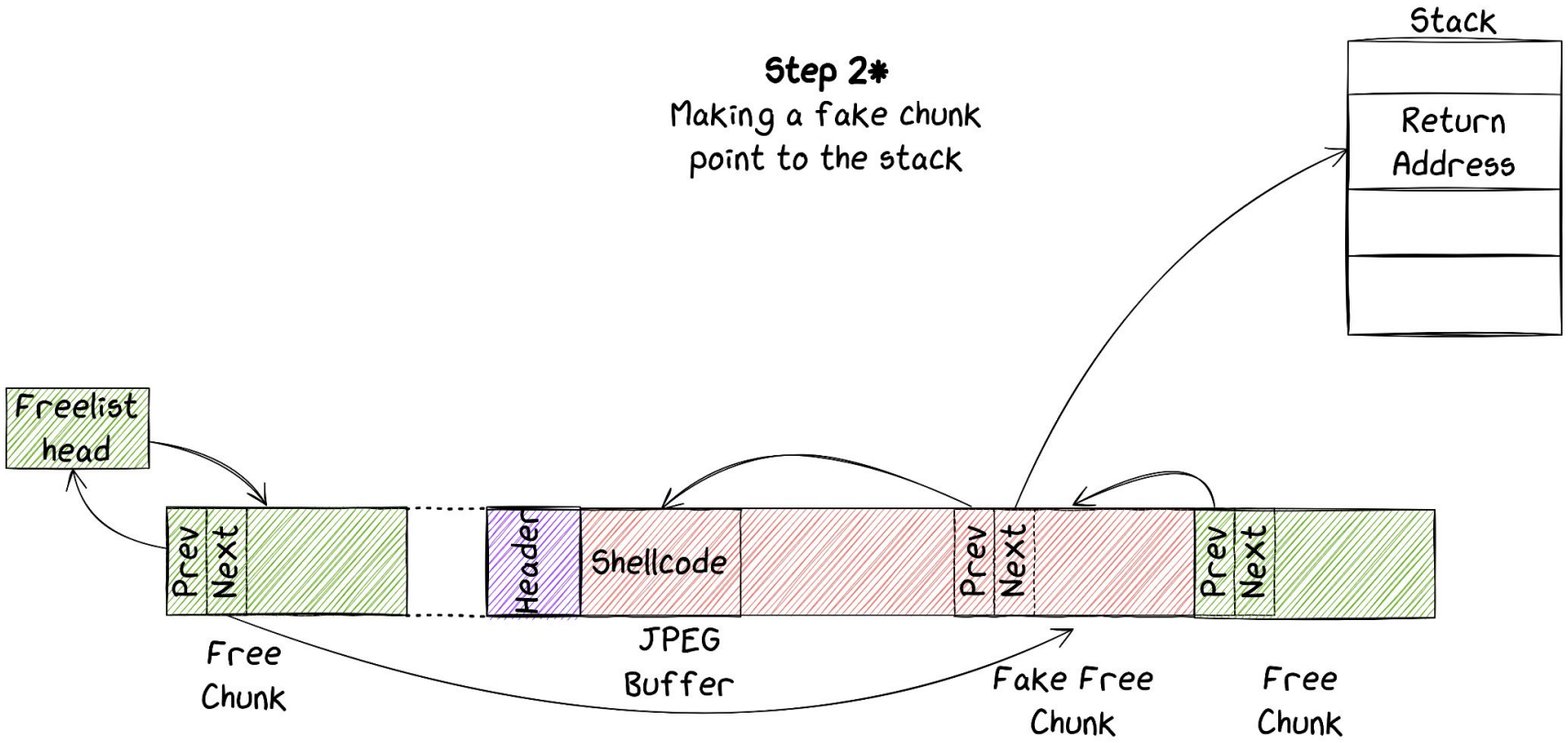
Exploiting a Heap Overflow in Little Kernel



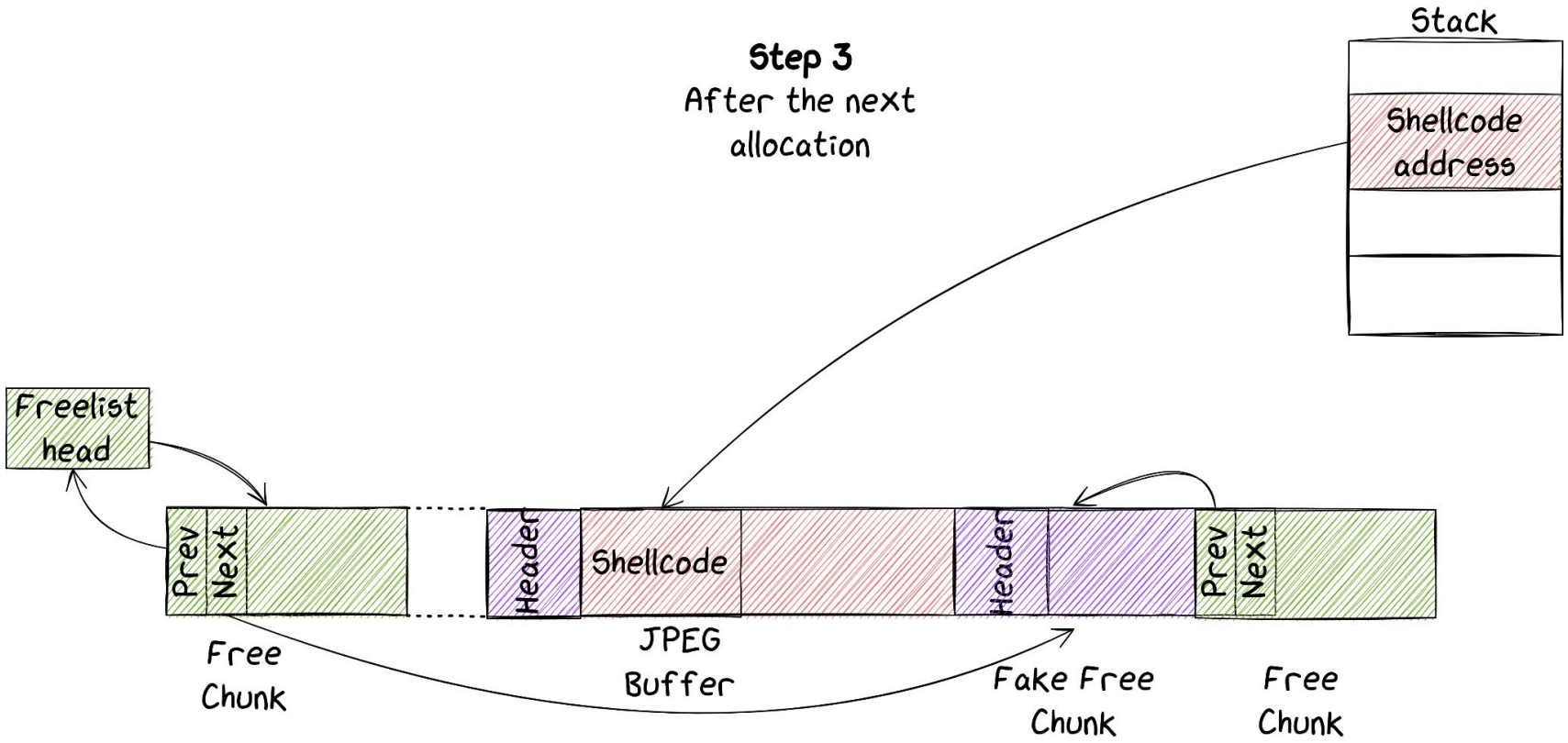
Exploiting a Heap Overflow in Little Kernel



Exploiting a Heap Overflow in Little Kernel



Exploiting a Heap Overflow in Little Kernel



Emulating and Debugging our Exploit

- We used Unicorn
  - Emulates CPU only
  - We do not care about full-system emulation
  - Easy to setup & tweak

✅ Exploiting the vulnerability is quite straightforward in the emulator

❌ But the same exploit does not work on the real device

Emulating and Debugging our Exploit

What can go wrong?

Emulating and Debugging our Exploit

What can go wrong?

❌ Heap and stack are different from the ones IRL

Emulating and Debugging our Exploit

What can go wrong?

❌ Heap and stack are different from the ones IRL

✅ Dump it from the device

Emulating and Debugging our Exploit

What can go wrong?

❌ Heap and stack are different from the ones IRL

✅ Dump it from the device

👉 Memory state (addresses, sizes) is the same every boots

Emulating and Debugging our Exploit

What can go wrong?

- ❌ Heap and stack are different from the ones IRL
  - ✅ Dump it from the device
- ❌ Heap algo writes something at the address `*(shellcode+4)`

Emulating and Debugging our Exploit

What can go wrong?

❌ Heap and stack are different from the ones IRL

✅ Dump it from the device

❌ Heap algo writes something at the address *(shellcode+4)

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

Emulating and Debugging our Exploit

What can go wrong?

❌ Heap and stack are different from the ones IRL

✅ Dump it from the device

❌ Heap algo writes something at the address *(shellcode+4)

✅ Skip the next instruction in the shellcode

```
add pc, pc, 0x4;
nop
nop
```

Emulating and Debugging our Exploit

What can go wrong?

- ❌ Heap and stack are different from the ones IRL
  - ✅ Dump it from the device
- ❌ Heap algo writes something at the address `*(shellcode+4)`
  - ✅ Skip the next instruction in the shellcode
- ❌ Heap after exploitation may crash during the next allocation

Emulating and Debugging our Exploit

What can go wrong?

- ❌ Heap and stack are different from the ones IRL
  - ✅ Dump it from the device
- ❌ Heap algo writes something at the address `*(shellcode+4)`
  - ✅ Skip the next instruction in the shellcode
- ❌ Heap after exploitation may crash during the next allocation
  - ✅ Restore the heap in a working state

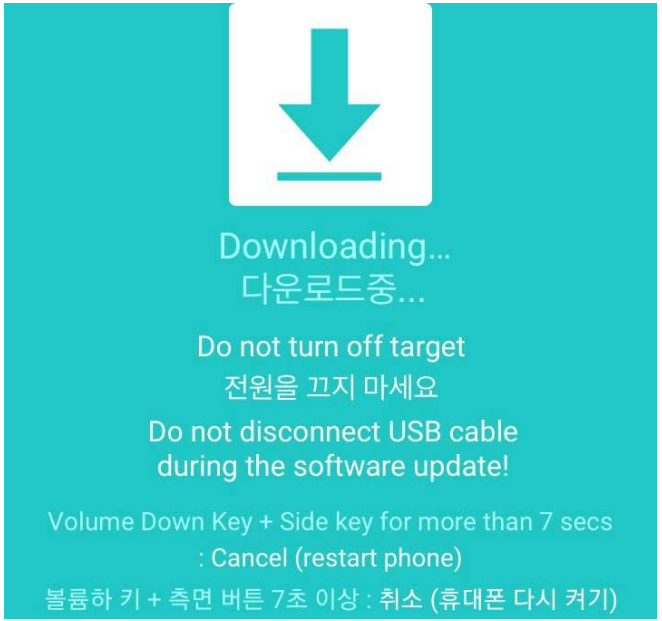
To sum-up

- 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

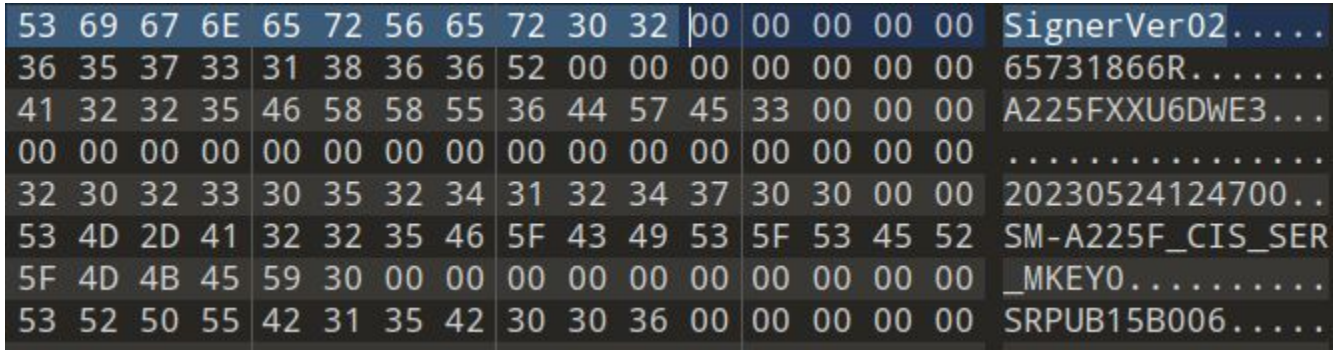
- Odin is implemented in LK
- It is available through the *Download Mode*
  - It allows to flash partitions over USB
- The Odin official client is closed source
- There is an open-source client: Heimdall[4]



[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



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

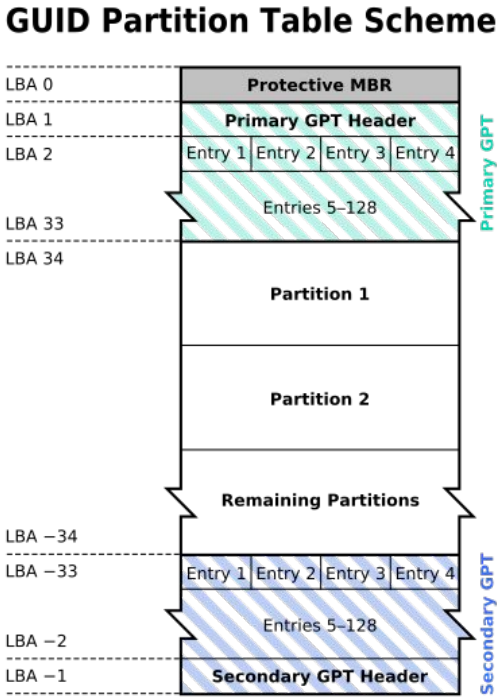
- A global array indicates how an image should be authenticated
- An image not present in this array will not be authenticated
  - (Except for some specific images)
- Comparing this array with PIT gives a set of images flashable without authentication

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



Source: https://en.wikipedia.org/wiki/GUID_Partition_Table

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);
```

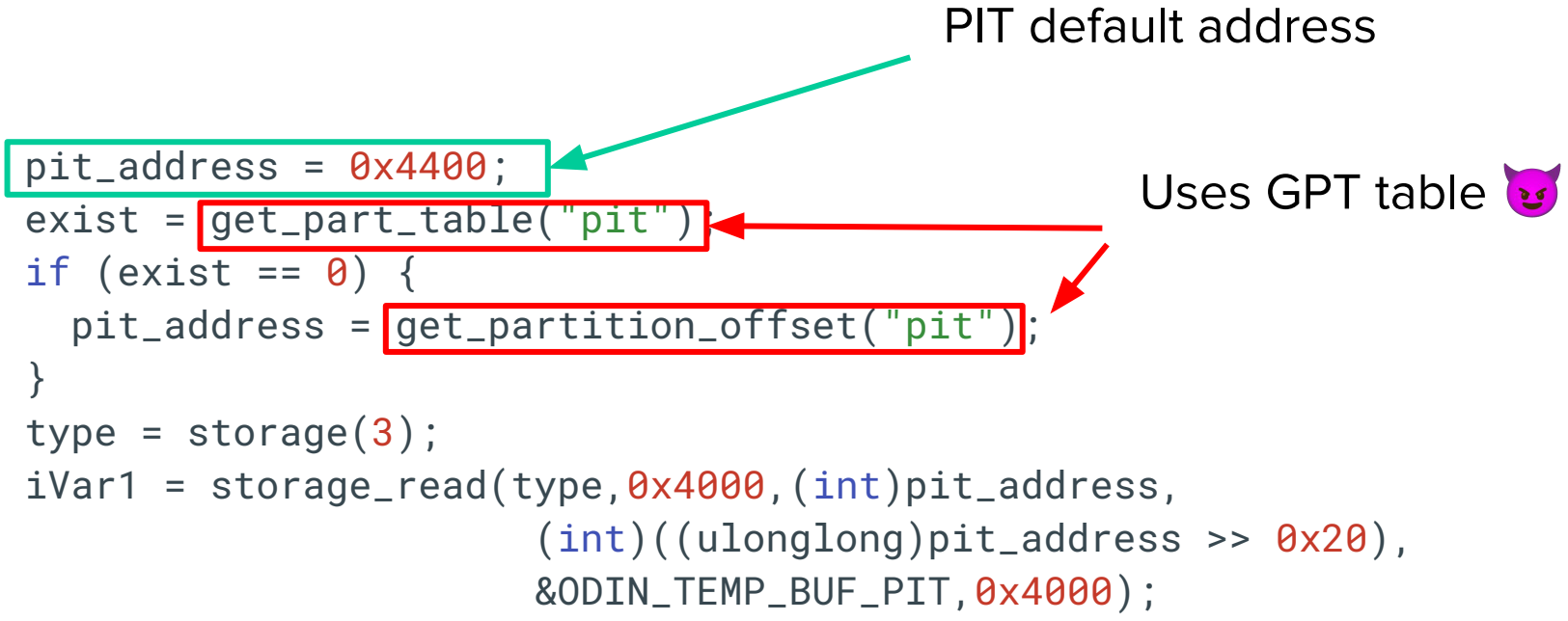
PIT Loading

PIT default address

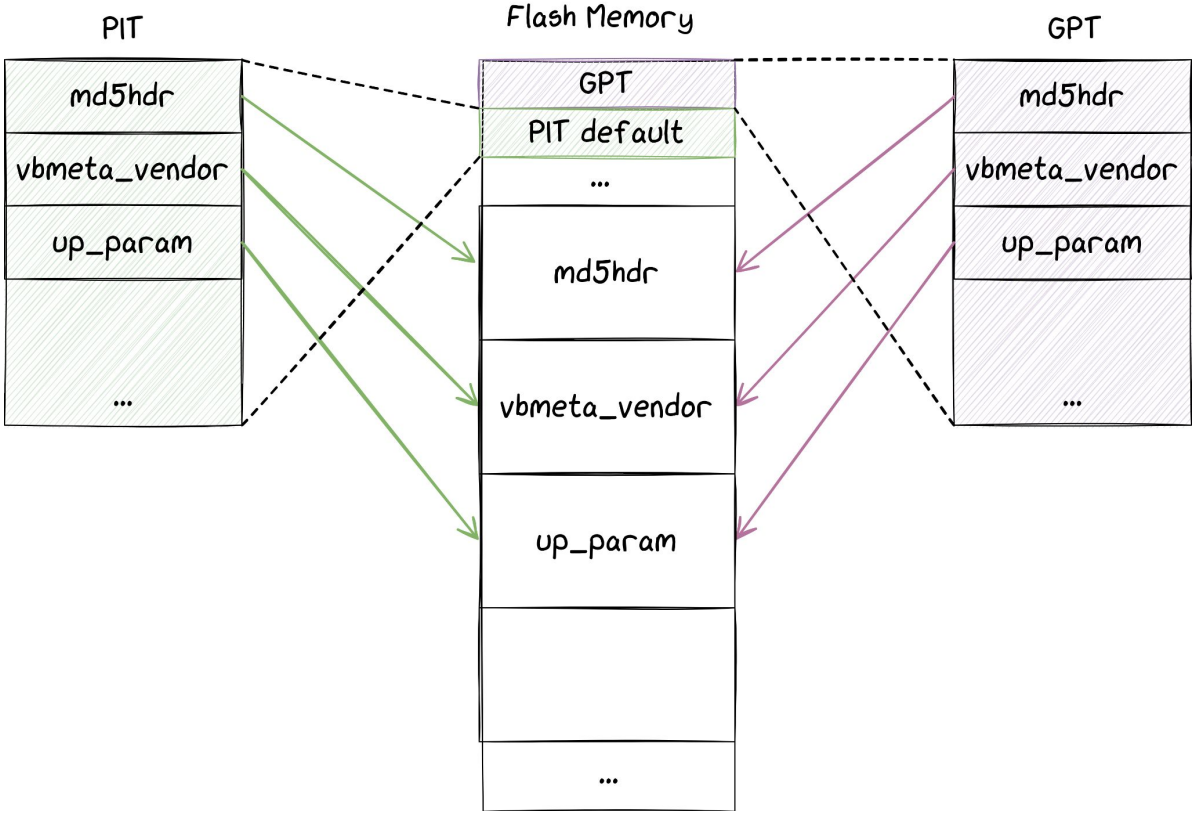
Check for pit partition
And use it if it exists

```
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);
```

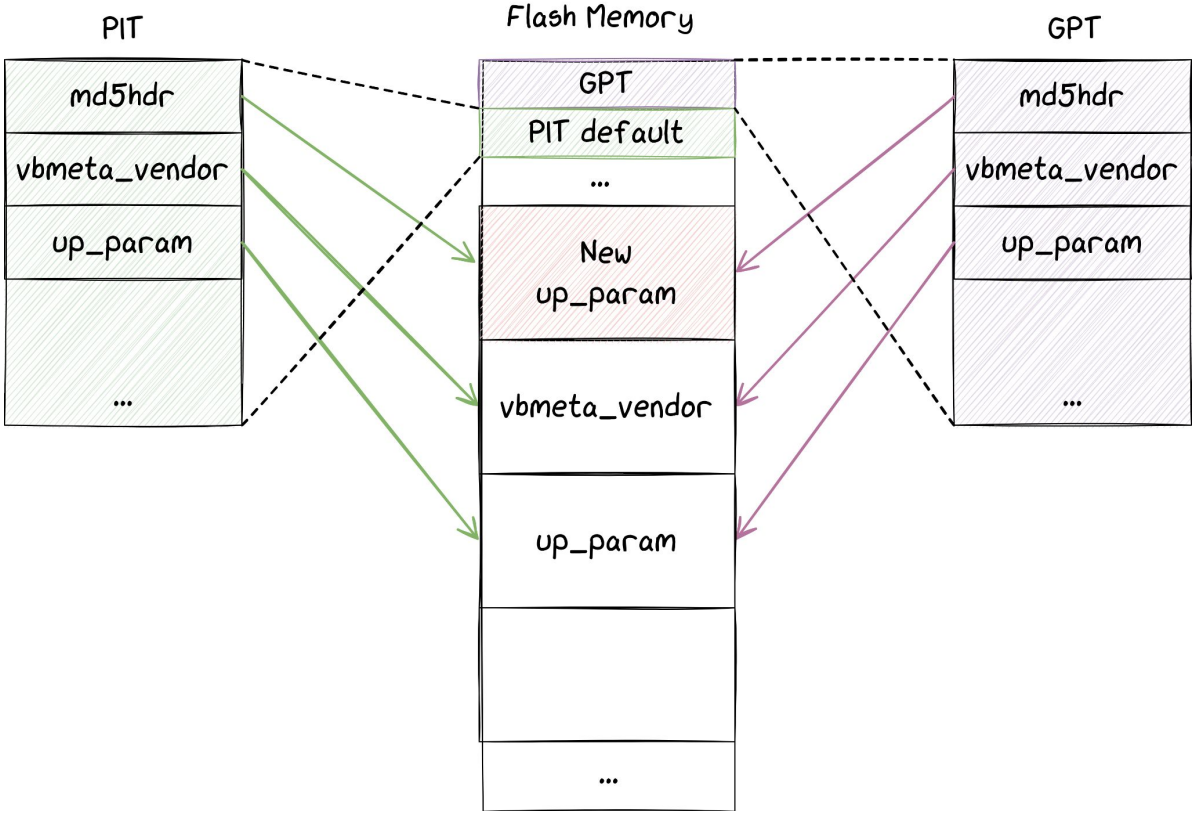
PIT Loading


PIT default address
Uses GPT table 😈
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);

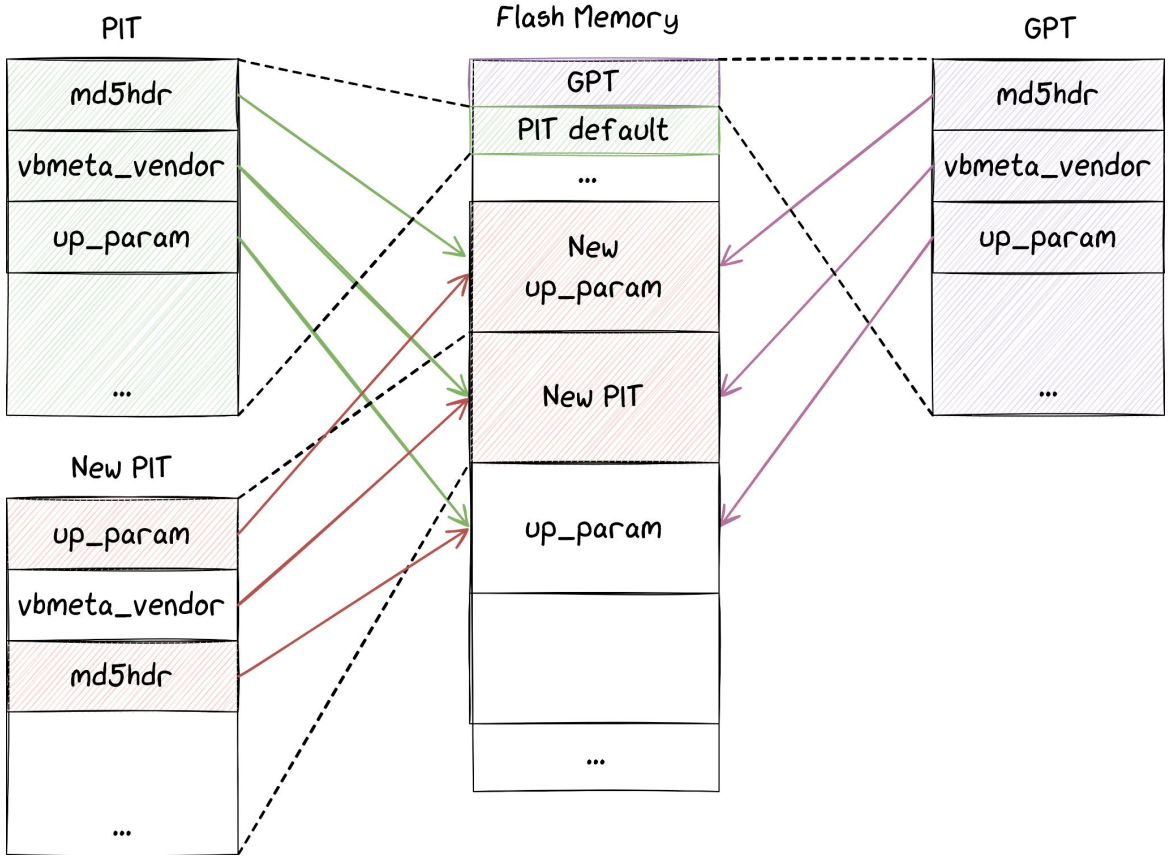
Strategy to Bypass Odin Authentication



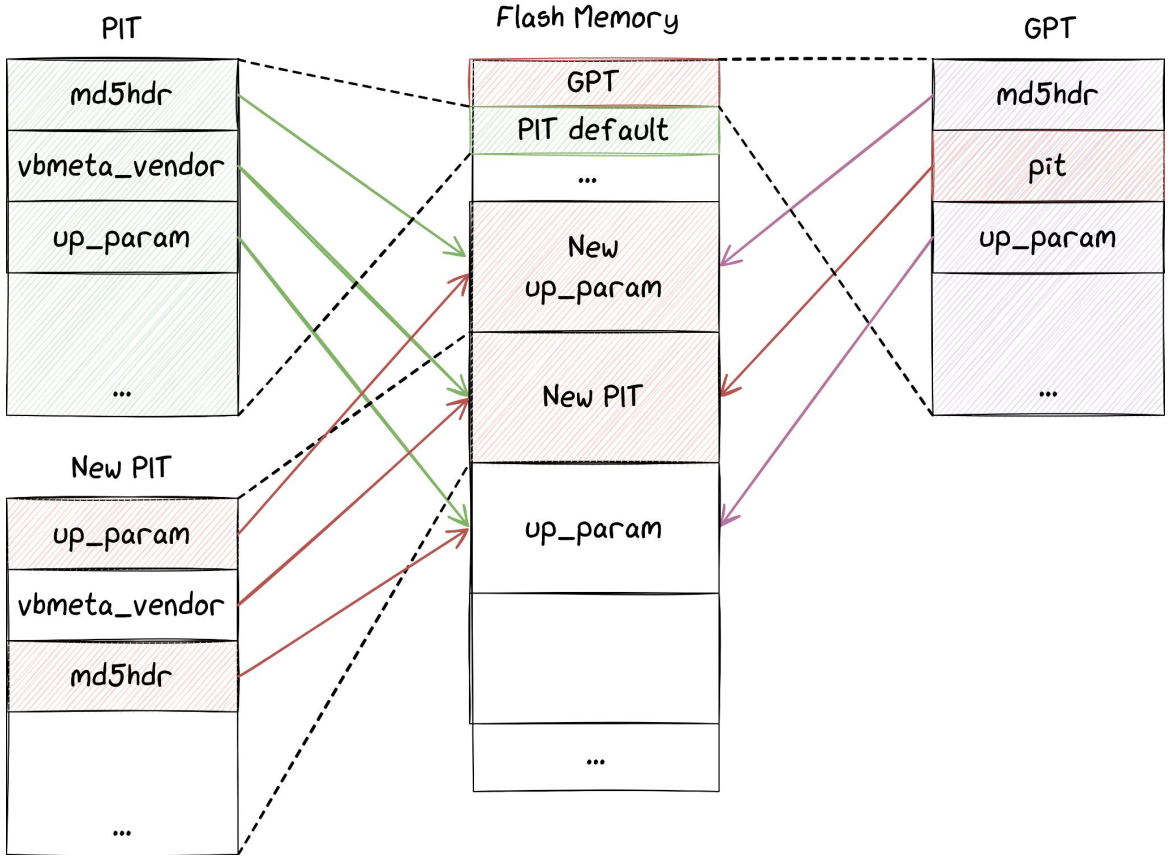
Strategy to Bypass Odin Authentication



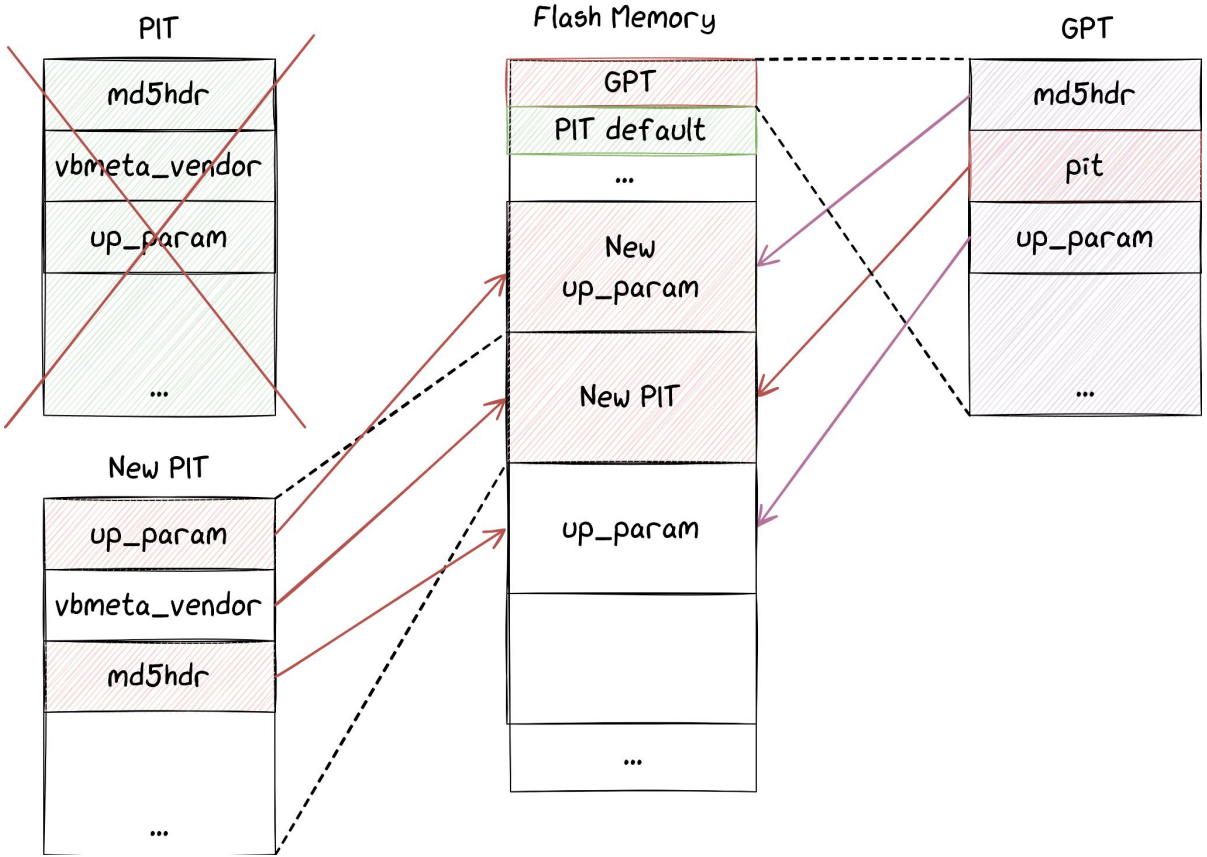
Strategy to Bypass Odin Authentication



Strategy to Bypass Odin Authentication



Strategy to Bypass Odin Authentication

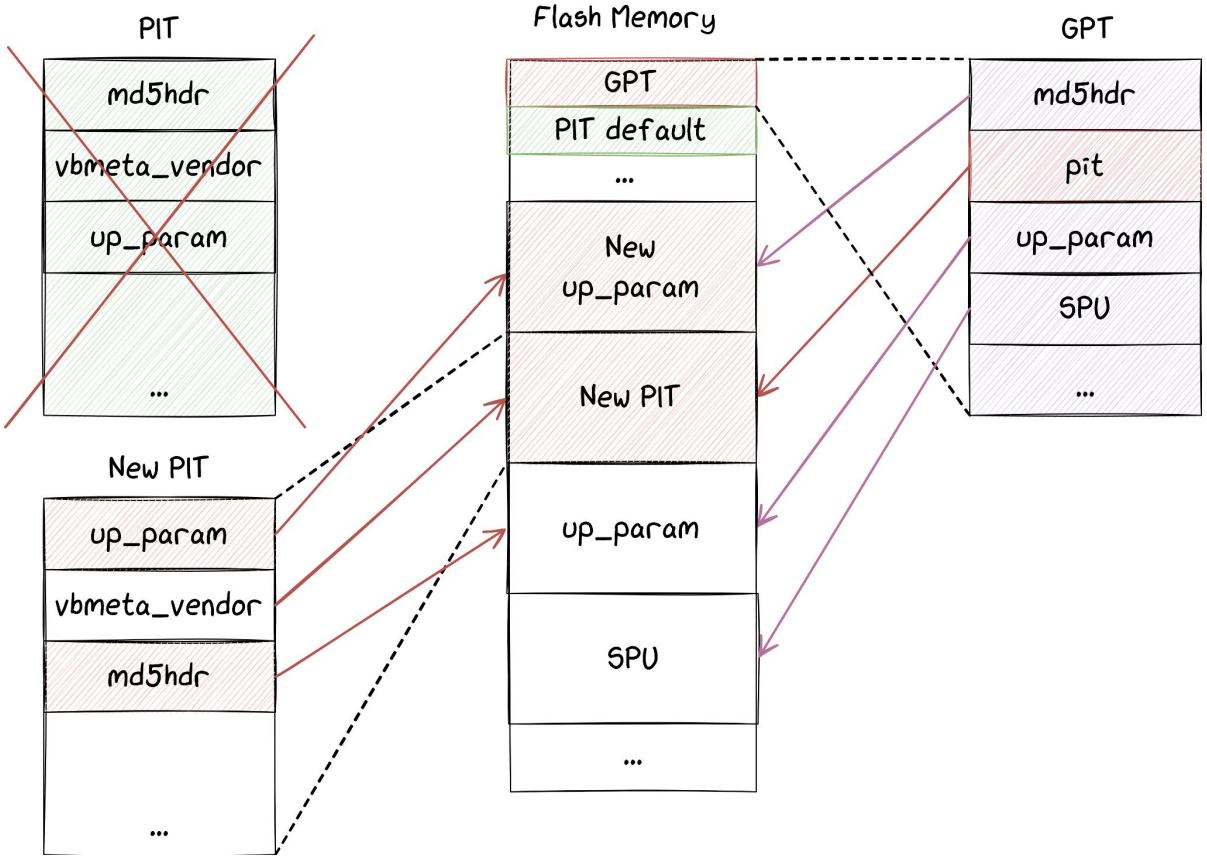


Strategy to Bypass Odin Authentication

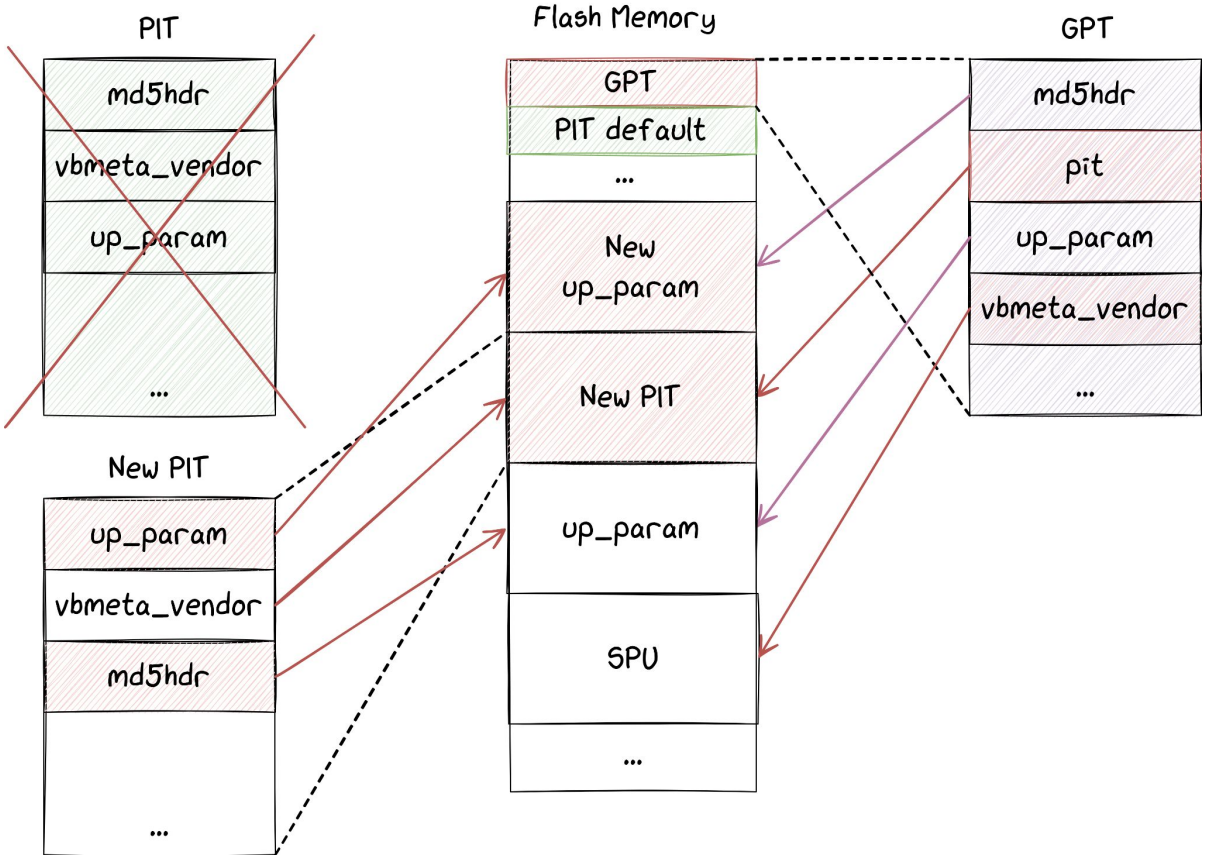
✅ It works (new jpegs are loaded)

❌ Android won't boot without the *vbmeta_vendor* partition

Strategy to Bypass Odin Authentication

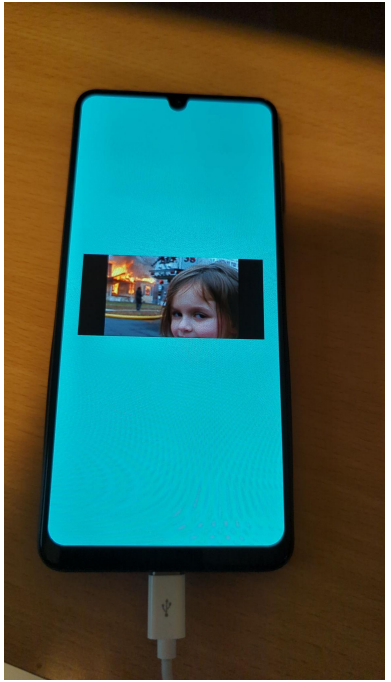


Strategy to Bypass Odin Authentication



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

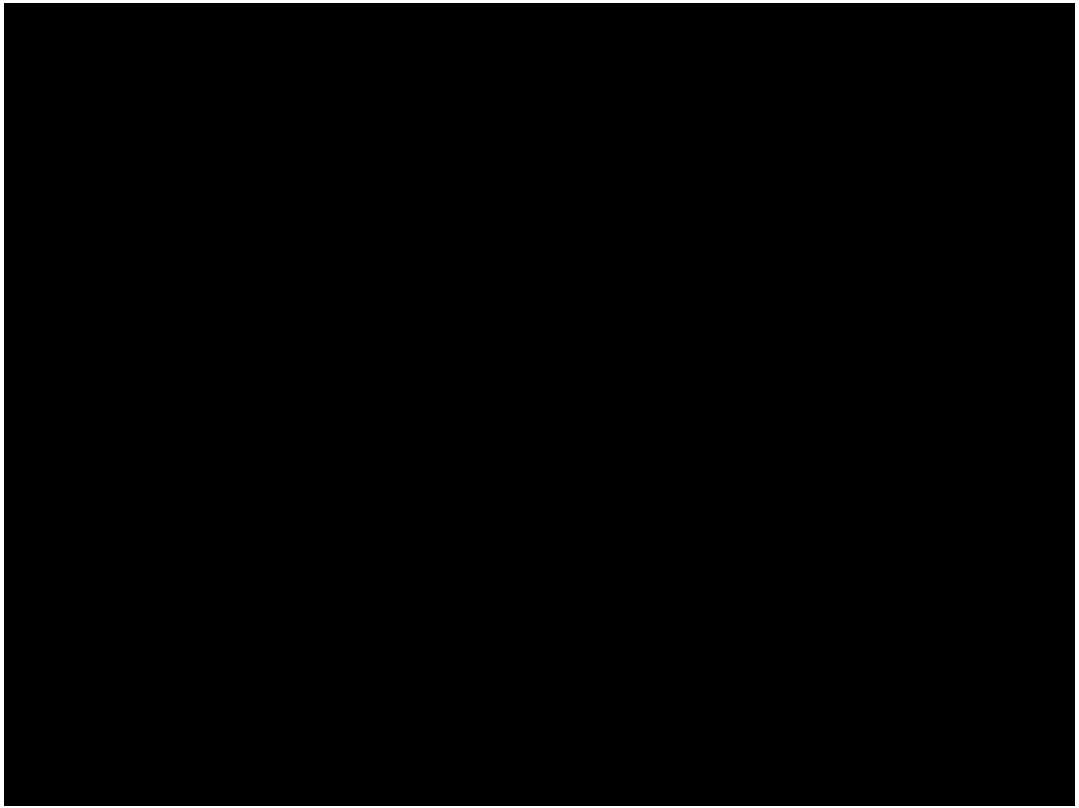


Post Exploitation: bypassing Android Verified Boot

- Our ultimate goal is to bypass AVB checks in LK to load a modified Android
  - We patch LK code from our shellcode
  - Memory already writable ➔ no need to play with MMU

```
26  iVar1 = do_hash(param_1,param_2,DAT_4c6463e0 - param_2,&hash,0x20);
27  if (iVar1 == 0) {
28    iVar2 = memcmp(&STORED_HASH,&hash,0x20);
29    if (iVar2 == 0) {
30      print("[%s][oem] img auth pass\n",&s_SBC_030151a8);
31      goto LAB_02ff82e0;
32    }
33    iVar1 = 0x7021;
34  }
35  print("[%s][oem] img auth fail (0x%x)\n",&s_SBC_030151a8,iVar1);
```

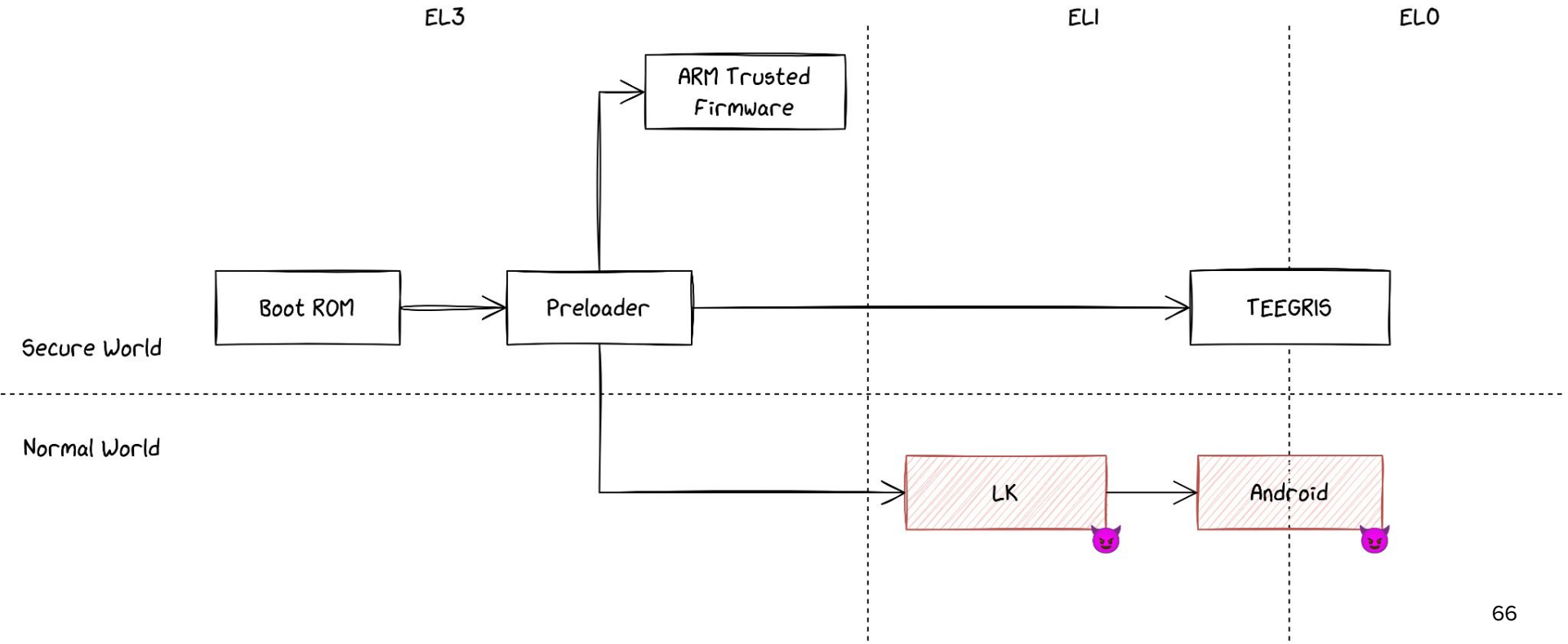
Chaining Everything Together



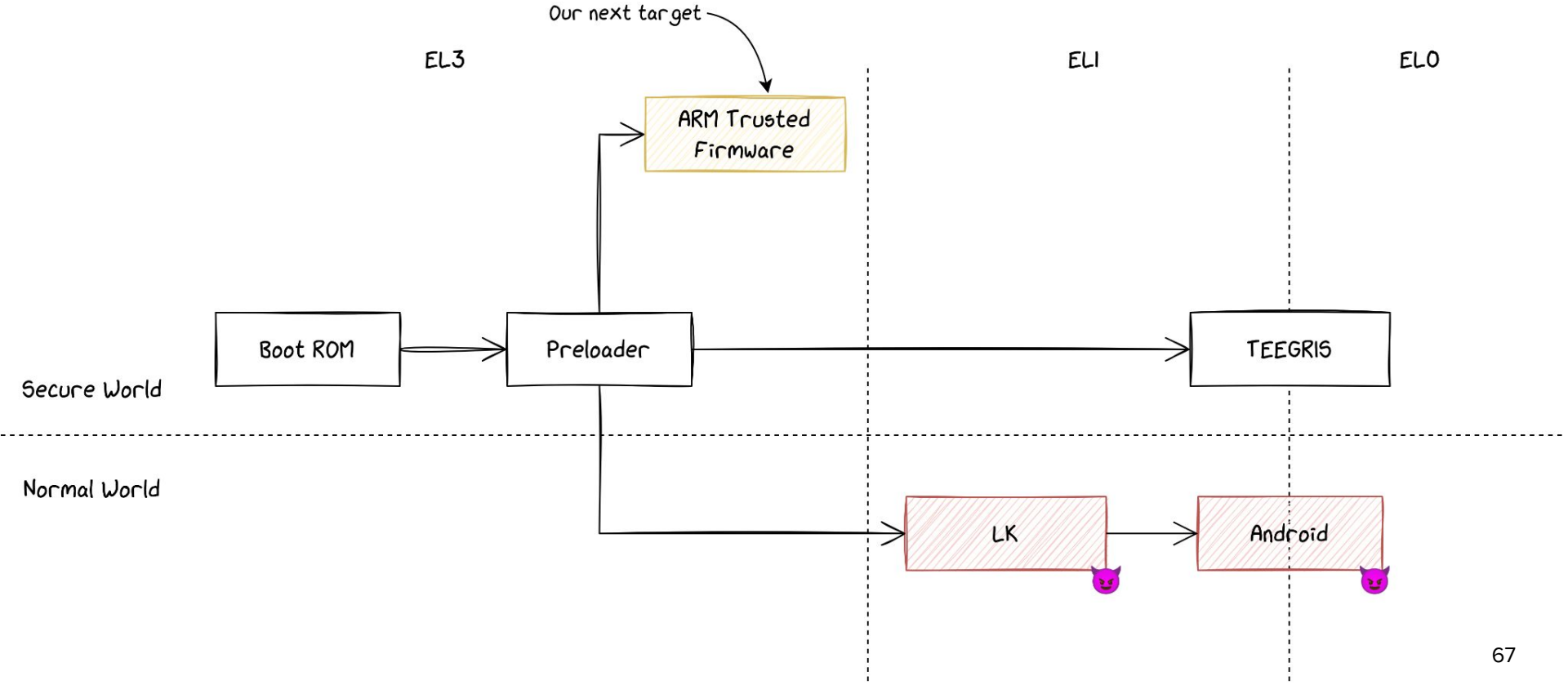
To Conclude

- 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

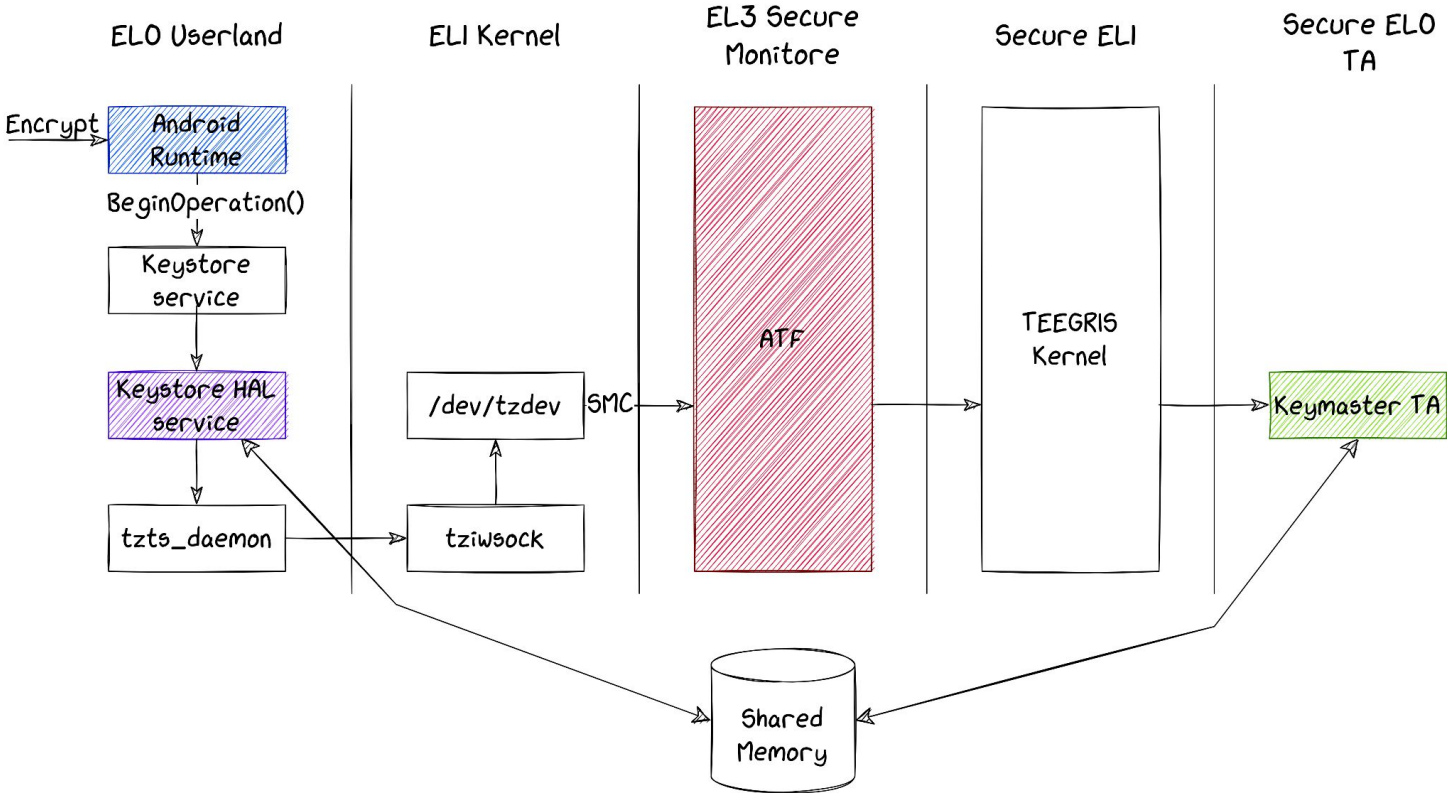
Targeting ARM Trusted Firmware



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

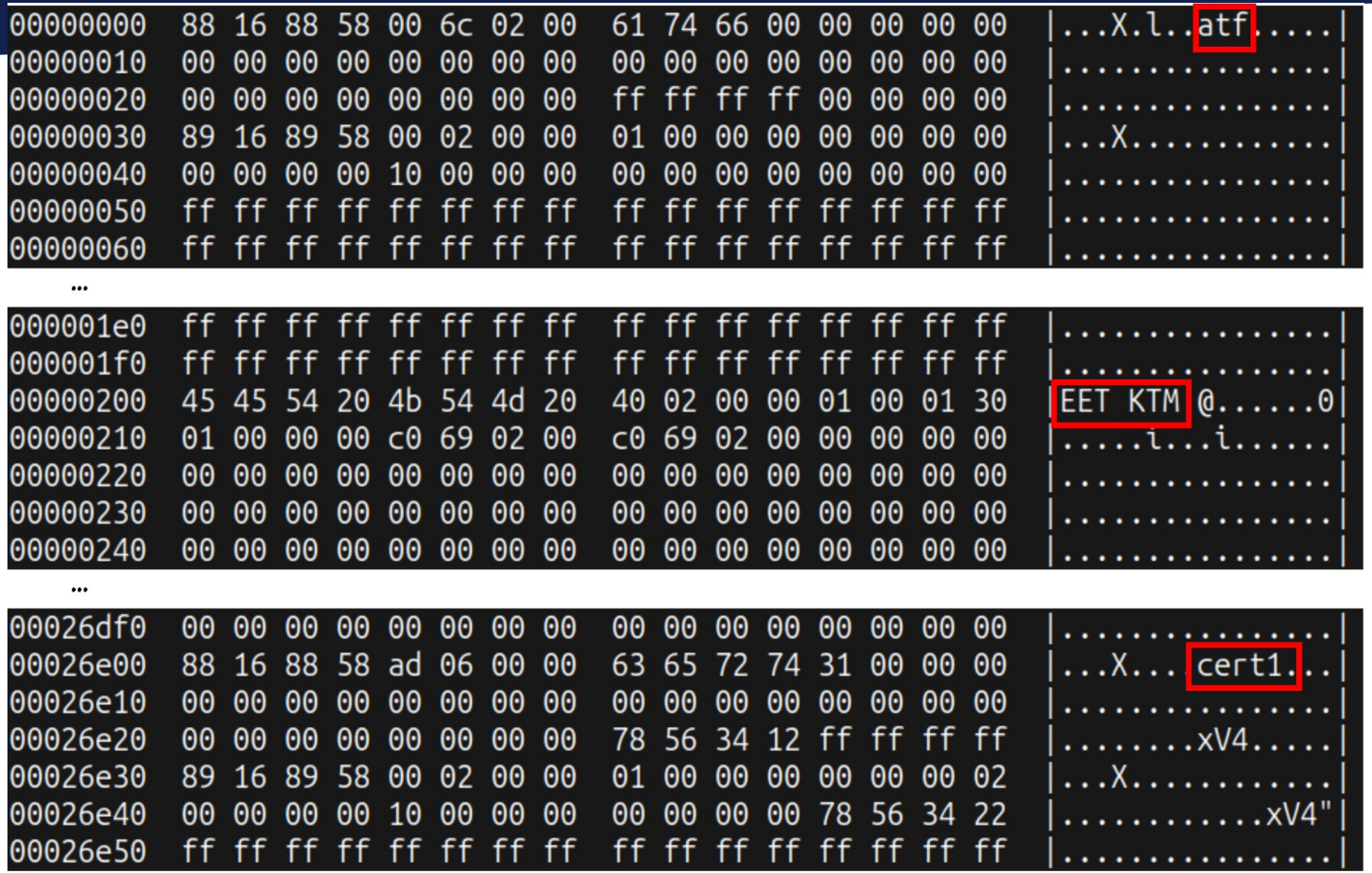
Few Words about TEEGRIS

TEEgris

- Trustzone OS designed by Samsung
- ROM images:
  - `tee-verified.img`: ATF, TEEGRIS kernel, `userboot.so`
  - `tzar.img`: TEE root filesystem
  - `super.img`: Android system, Trusted Applications and Drivers
- Excellent references online[7]

[7]: https://www.riscure.com/tee-security-samsung-teegris-part-1/

Extracting ATF

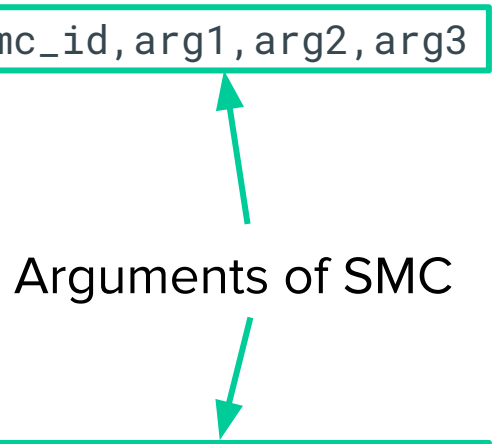


SMC Handlers

```
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;
}
```



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

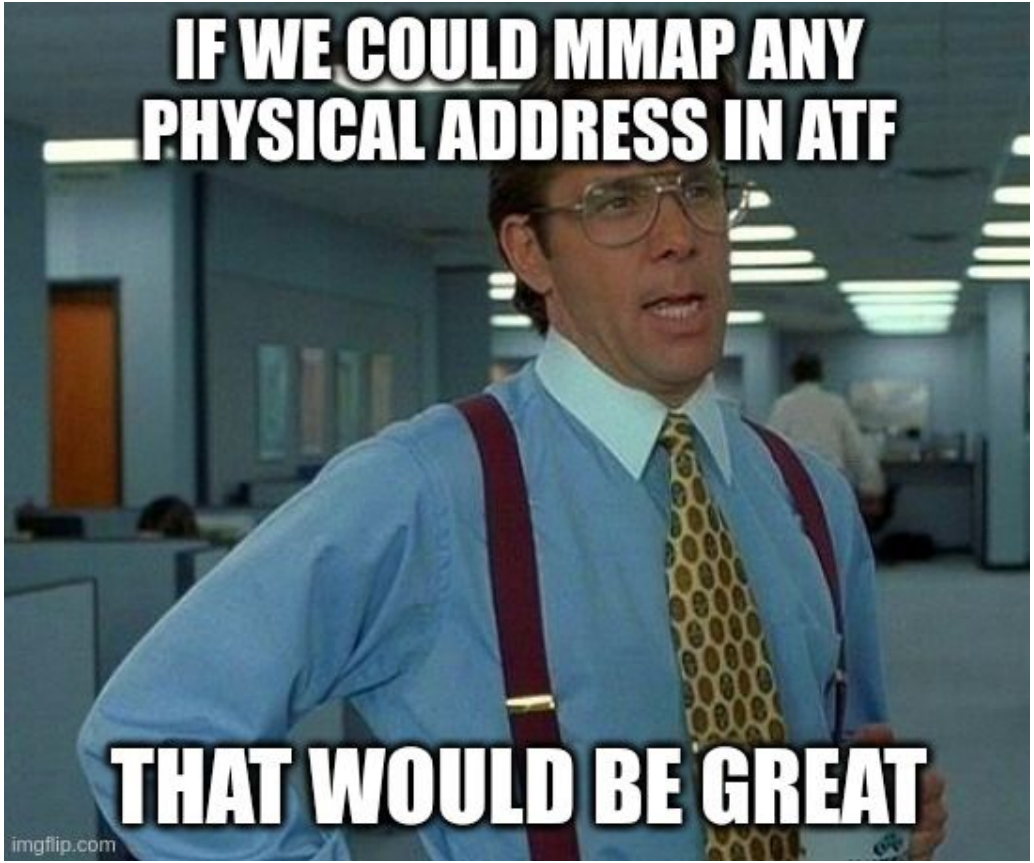
```
uint* global_array = (uint *)0x4ce2f578;
[...]
if (smcid == 0x82000526) {
    out_value = global_array[arg1 * 4];
    goto exit;
}
[...]
    output[2] = out_value;
    output[1] = arg1;
    *output = 0;
    return output;
```

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

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



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

CVE-2024-20021

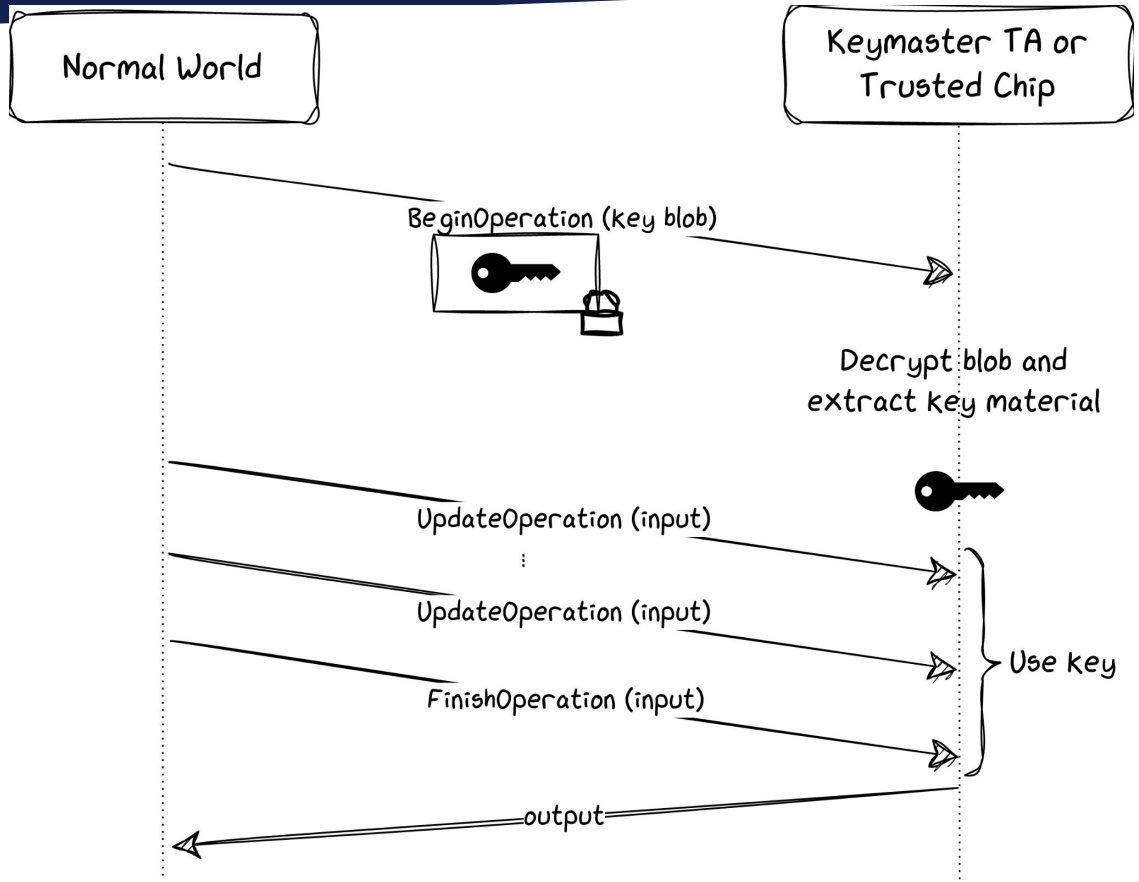
- Also in `mediatek_plat_sip_handler_kernel`
- Will `mmap` with physical base address to the same virtual address
  - ... however we can't `munmap`
    - So we are limited to 8 consecutives `mmaps`
    - Meaning we can leak up to **8MB** of data
- Introduced by Mediatek (impacts plenty of Mediatek SoCs)
- 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 *key blobs*
- Three protection levels:
  - Software only
  - TEE (default)
  - Hardware-backed (StrongBox)
- Raw key should never leave protected environment

Android Keystore system



Where to leak?

```
a22:/ # cat /proc/last_kmsg
[...]
[4425] mblock_reserve-R[5].start: 0x4ce00000, size: 0x60000 map:0
,name:atf-reserved
[...]
[4426] mblock_reserve-R[8].start: 0x7ac00000, size: 0x400000 map:0
,name:tee-secmem
[4426] mblock_reserve-R[9].start: 0x7f300000, size: 0xc0000 map:0
,name:SSPM-reserved
[4426] mblock_reserve-R[10].start: 0x7b200000, size: 0x4000000
, map:0 name:tee-reserved
[...]
[4436] lk finished --> jump to linux kernel 64Bit
```

Where to leak?

```
a22:/ # cat /proc/last_kmsg
[...]
[4425] mblock_reserve-R[5].start: 0x4ce00000, size: 0x60000 map:0
,name:atf-reserved
[...]
[4426] mblock_reserve-R[8].start: 0x7ac00000, size: 0x400000 map:0
,name:tee-secmem
[4426] mblock_reserve-R[9].start: 0x7f300000, size: 0xc0000 map:0
,name:SSPM-reserved
[4426] mblock_reserve-R[10].start: 0x7b200000, size: 0x4000000
, map:0 name:tee-reserved
[...]
[4436] lk finished --> jump to linux kernel 64Bit
```

Where to leak?

- TAs should be part of the *tee-reserved* (`0x7b200000`) memory block
- Keymaster TA binary contains many strings
  - (path in Android fs: `/vendor/tee/00000000-0000-0000-0000-4b45594d5354`)
- Try to find these strings in memory
  - Usually present around `0x7c200000`

Our PoC

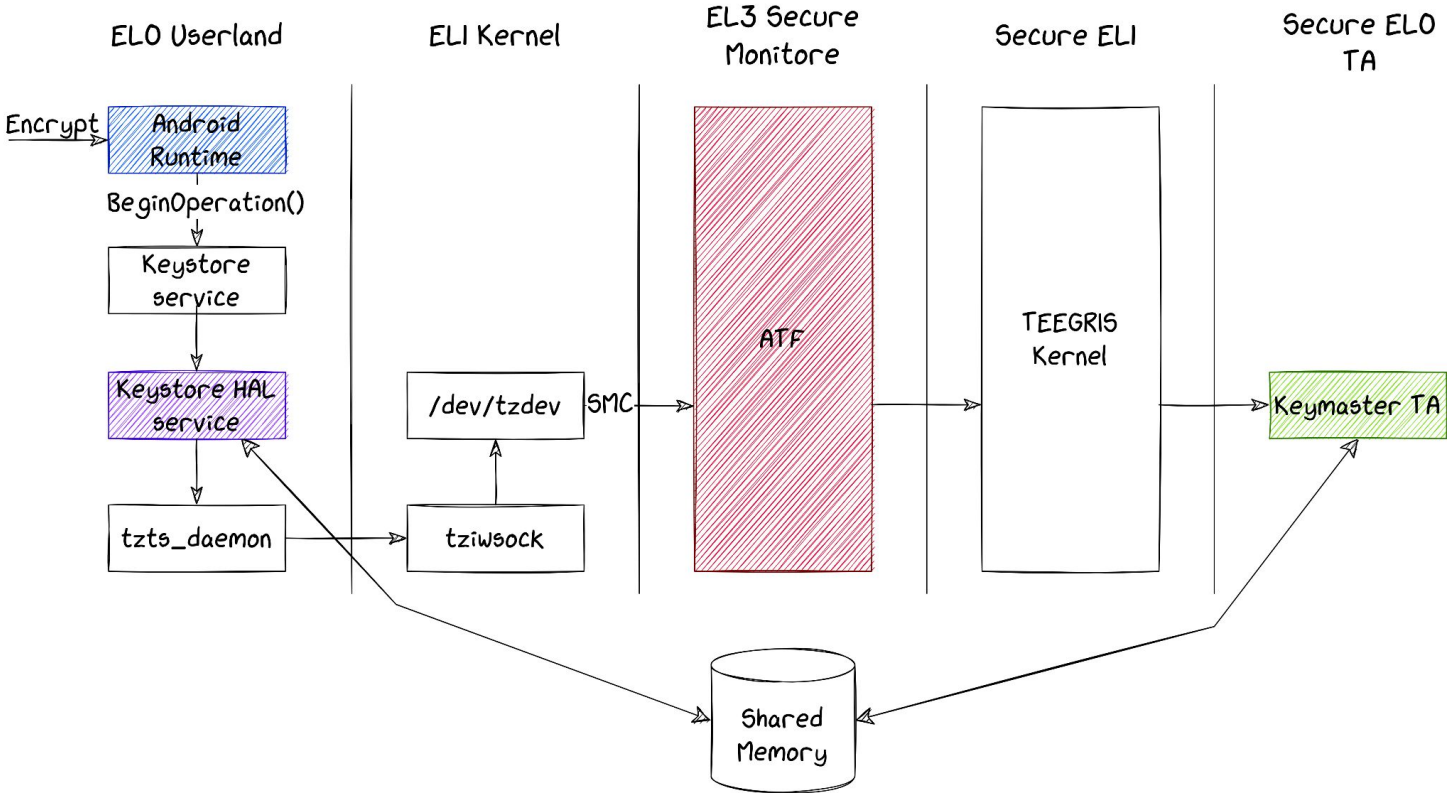
1. **Import** a key into the Android Keystore
2. **Encrypt** using that key
3. **Stop the execution** after `BeginOperation` is called
   - To makes sure the key stays in memory
4. **Leak** the identified region of memory

A Simple PoC

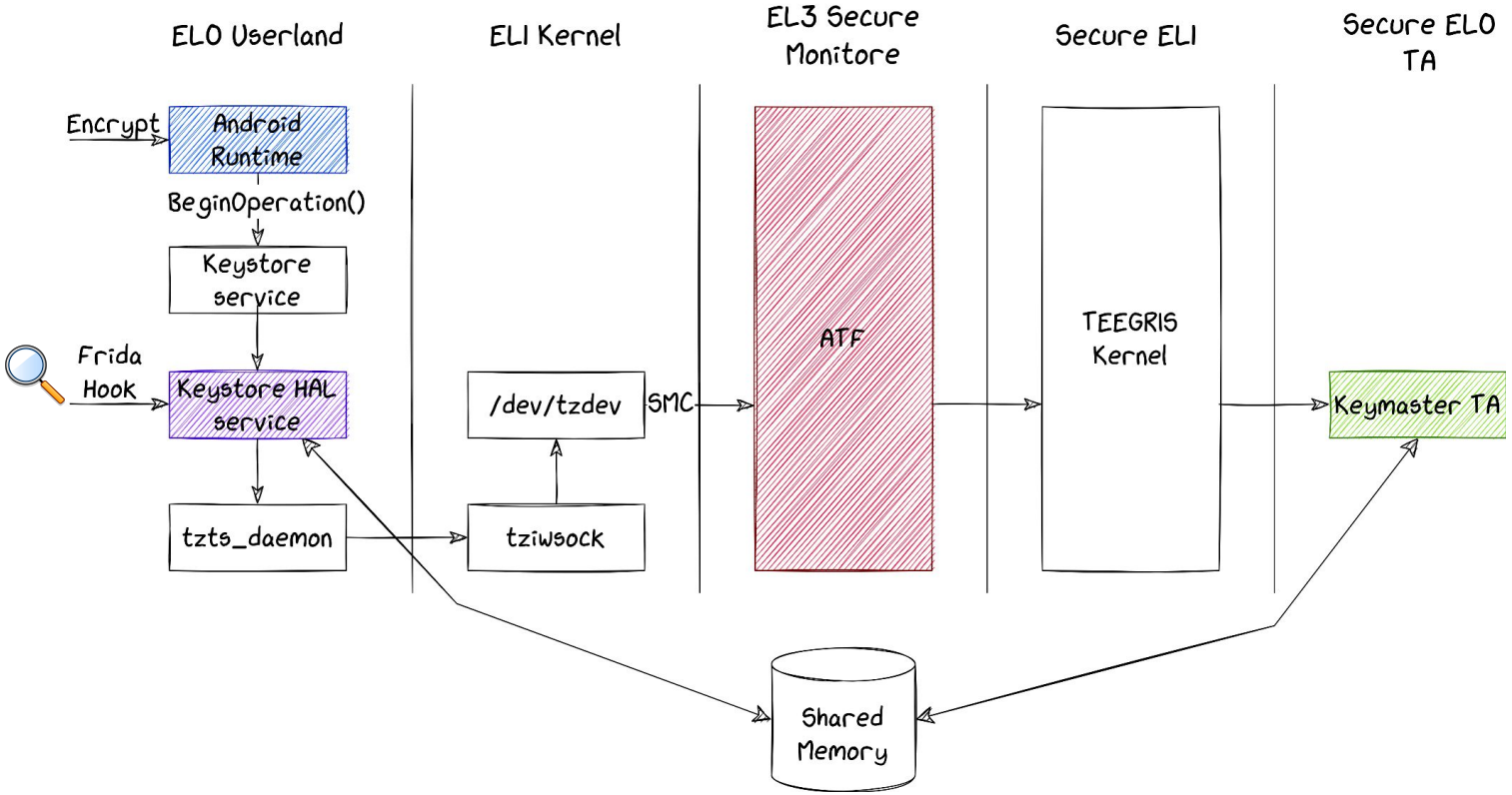
```
byte[] key = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA".getBytes();
SecretKey yourKey = (SecretKey) new SecretKeySpec(key, 0, key.length,
"AES");
[...]
keyStore = KeyStore.getInstance("AndroidKeystore");
keyStore.load(null);
keyStore.setEntry("key1", new KeyStore.SecretKeyEntry(yourKey), new
KeyProtection.Builder(KeyProperties.PURPOSE_ENCRYPT |
    KeyProperties.PURPOSE_DECRYPT)
        .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
        .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
        .build());

keyStoreKey = (SecretKey) keyStore.getKey("key1", null);
```

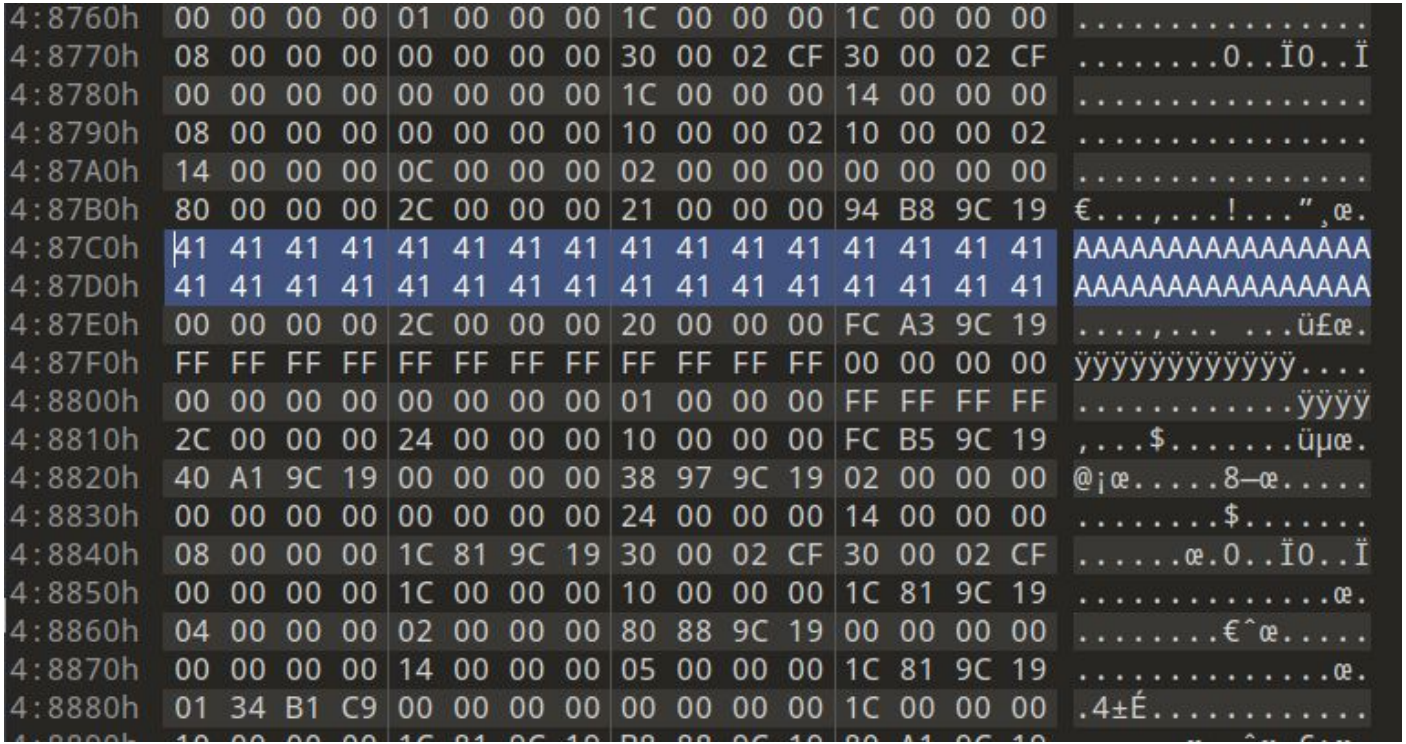
Communication between NSW and SW



Hooking Keystore HAL Deamon



The result



```
4:8760h 00 00 00 00 01 00 00 00 1C 00 00 00 1C 00 00 00 ................
4:8770h 08 00 00 00 00 00 00 00 30 00 02 CF 30 00 02 CF ........0..Ï0..Ï
4:8780h 00 00 00 00 00 00 00 00 1C 00 00 00 14 00 00 00 ................
4:8790h 08 00 00 00 00 00 00 00 10 00 00 02 10 00 00 02 ................
4:87A0h 14 00 00 00 0C 00 00 00 02 00 00 00 00 00 00 00 ................
4:87B0h 80 00 00 00 2C 00 00 00 21 00 00 00 94 B8 9C 19 €...,...!...”¸œ.
4:87C0h 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
4:87D0h 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
4:87E0h 00 00 00 00 2C 00 00 00 20 00 00 00 FC A3 9C 19 ....,... ...ü£œ.
4:87F0h FF FF FF FF FF FF FF FF FF FF FF FF 00 00 00 00 ÿÿÿÿÿÿÿÿÿÿÿÿ....
4:8800h 00 00 00 00 00 00 00 00 01 00 00 00 FF FF FF FF ............ÿÿÿÿ
4:8810h 2C 00 00 00 24 00 00 00 10 00 00 00 FC B5 9C 19 ,...$.......üµœ.
4:8820h 40 A1 9C 19 00 00 00 00 38 97 9C 19 02 00 00 00 @¡œ.....8—œ.....
4:8830h 00 00 00 00 00 00 00 00 24 00 00 00 14 00 00 00 ........$.......
4:8840h 08 00 00 00 1C 81 9C 19 30 00 02 CF 30 00 02 CF ......œ.0..Ï0..Ï
4:8850h 00 00 00 00 1C 00 00 00 10 00 00 00 1C 81 9C 19 ..............œ.
4:8860h 04 00 00 00 02 00 00 00 80 88 9C 19 00 00 00 00 ........€ˆœ.....
4:8870h 00 00 00 00 14 00 00 00 05 00 00 00 1C 81 9C 19 ..............œ.
4:8880h 01 34 B1 C9 00 00 00 00 00 00 00 00 1C 00 00 00 .4±É............
```

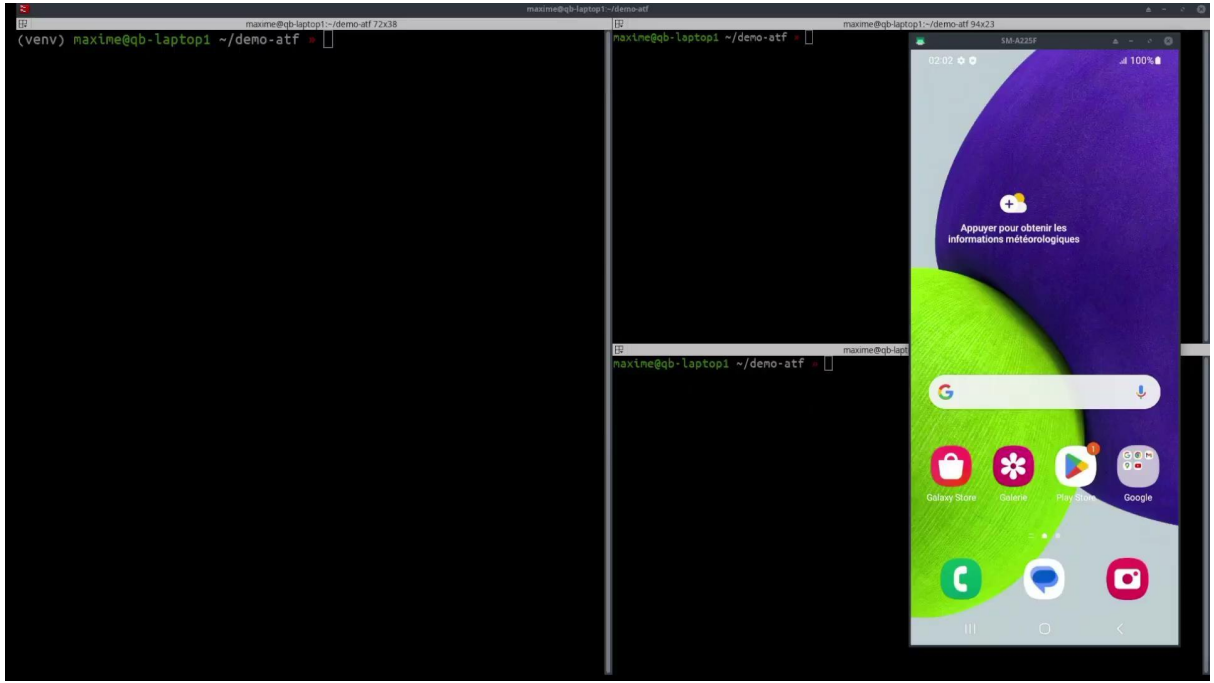
The result


leaking a key we already knew
It ain't much, but it's honest work
imgflip.com

Our PoC

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

Demo


(venv) maxime@qb-laptop1 ~/demo-atf
maxime@qb-laptop1 ~/demo-atf
maxime@qb-laptop1 ~/demo-atf
Appuyer pour obtenir les informations météorologiques
Galaxy Store
Galerie
Play Store
Google

Conclusion

- We presented 4 vulnerabilities leading to
  - Authentication bypass in Odin
  - Code execution with persistence in LK
  - Leak of SW memory, including Keystore 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!

@max_r_b
@DamianoMelotti
@pwissenlit

contact@quarkslab.com

Quarkslab