
/HEXACON/ · Impalabs
HARA KIRIN
DISSECTING THE PRIVILEGED
COMPONENTS OF HUAWEI MOBILE DEVICES
42
96
12

About Us

Maxime Peterlin – @lyte__
Security researcher & Co-founder

Alexandre Adamski – @NeatMonster_
Security researcher & Co-founder

Impalabs – @the_impalabs
French offensive security company
Reverse engineering, vulnerability research, exploit development

Website – https://impalabs.com
Blog – https://blog.impalabs.com

Outline

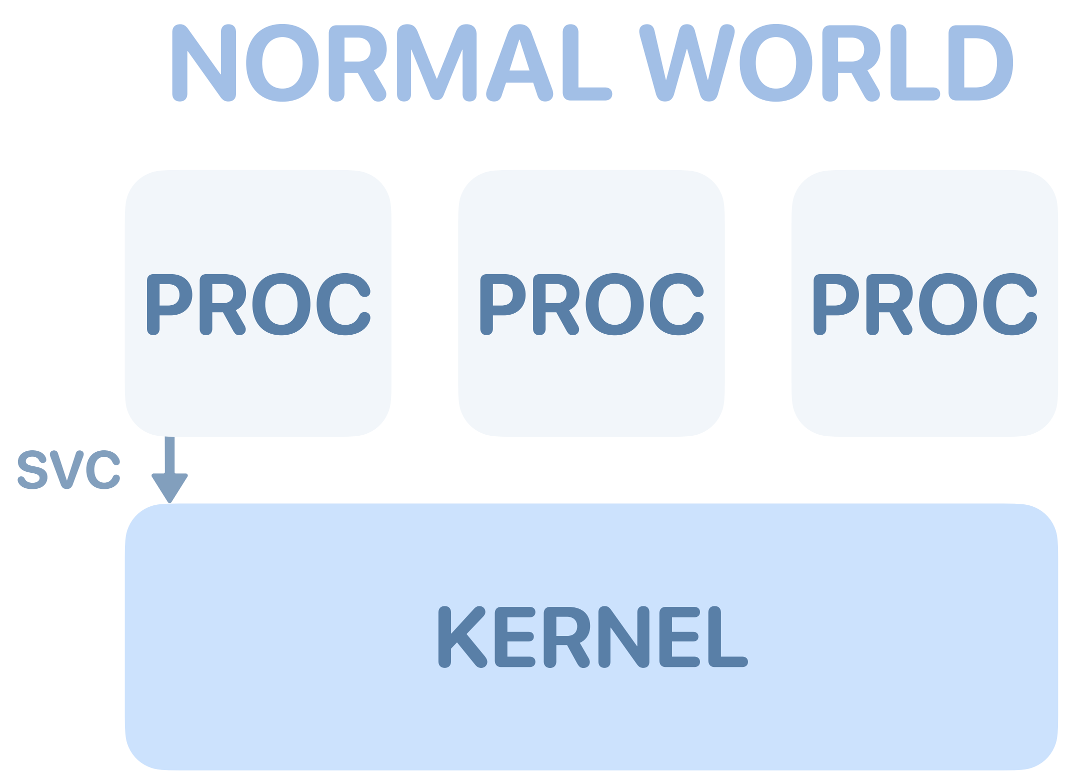
- Introduction
- Bootchain
- Hypervisor
- Secure Monitor
- Secure Kernel
- Trusted OS
- Trusted Applications
- Conclusion

Introduction

Android Device Architecture

Kernel-Based Security

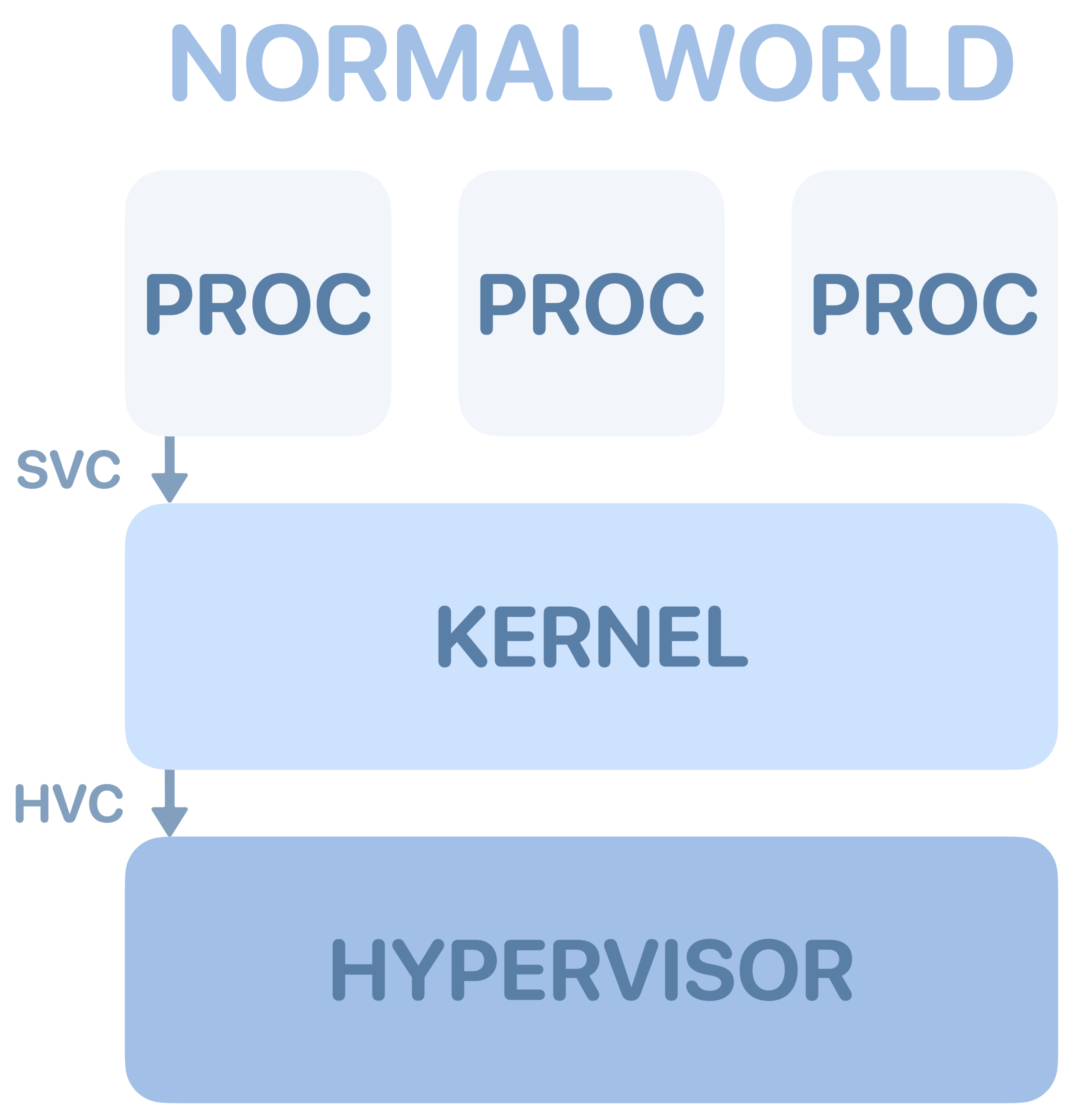
- **Access control** to resources from user space is enforced by the kernel
  - Address space isolation
  - Preemptive multitasking
  - Peripherals access restriction
- **Single point of failure**
  - Breaching kernel defenses results in full system compromise



Android Device Architecture

Security Hypervisor

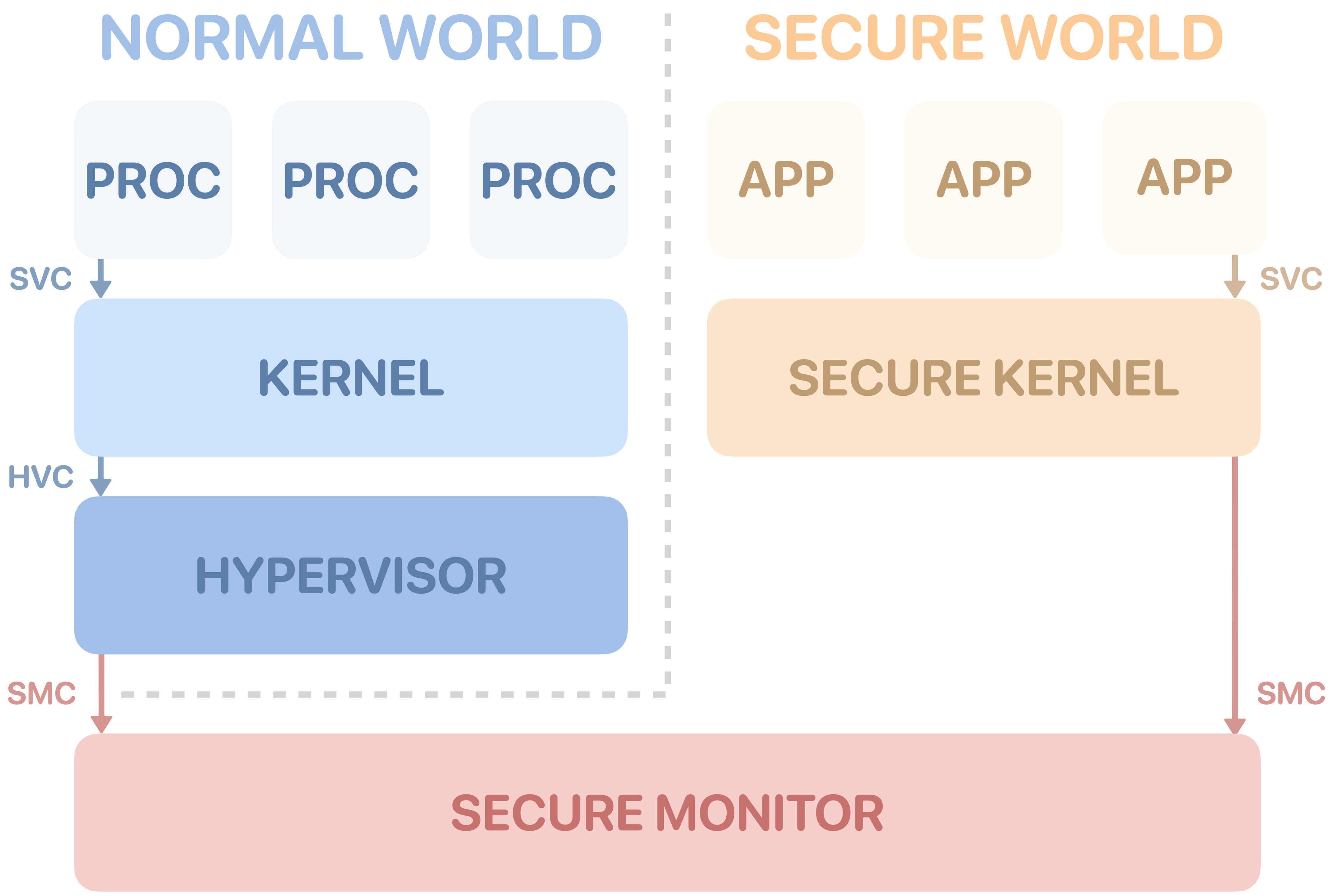
- **CPU virtualization**
  - Traditionally used to execute multiple operating systems in parallel on the same device
  - Leveraged on Android devices to enhance system security instead
- **ARM virtualization extensions**
  - Additional privilege level
  - Memory access restrictions
  - Exceptions interception
- Protects **critical data structures** at run time
  - Credentials, security contexts, page tables, etc.



Android Device Architecture

TrustZone for Cortex-A

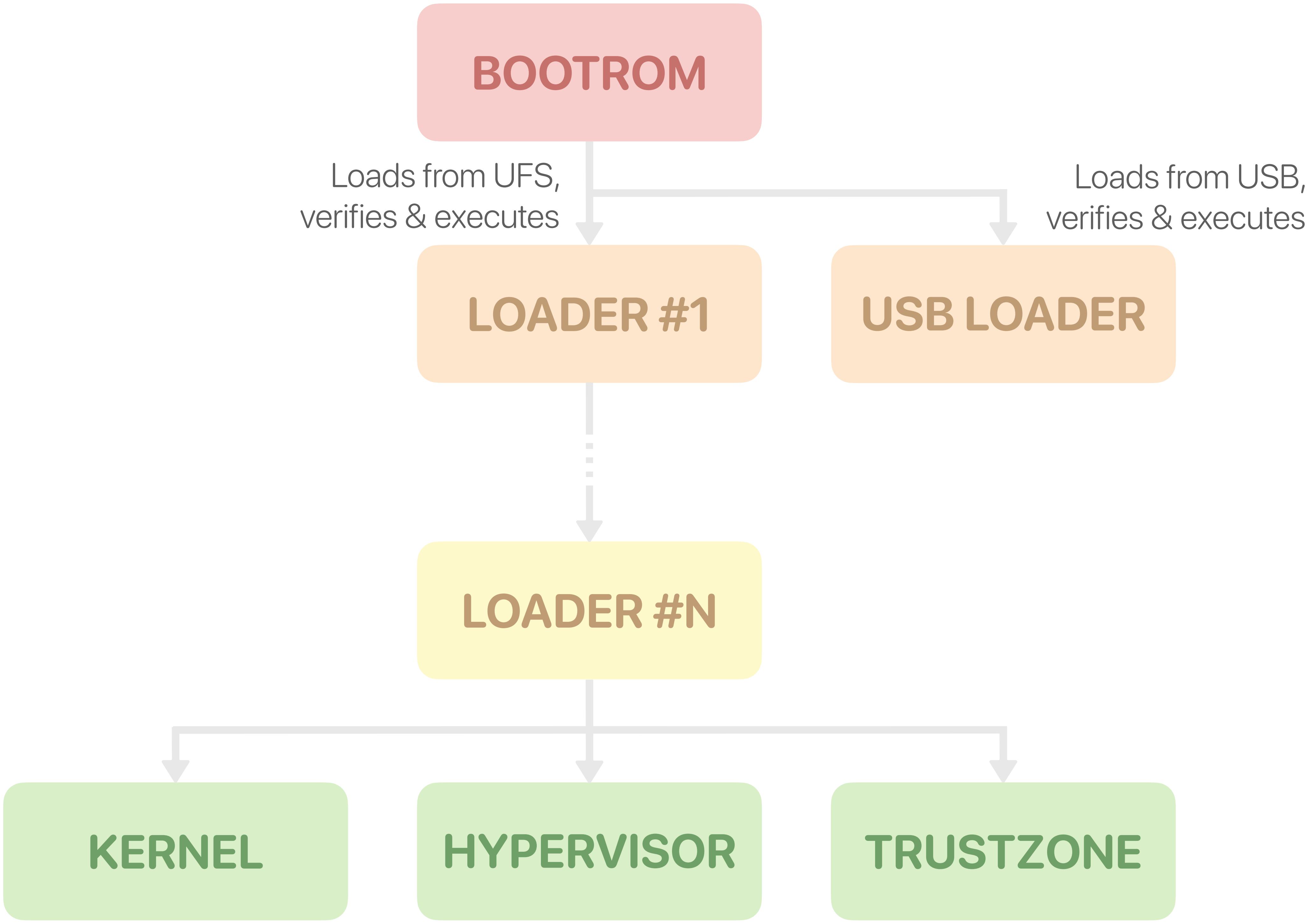
- System-wide **hardware separation**
  - An untrusted **Normal World** and a trusted **Secure World**
  - Access to secure hardware resources from non-secure software is prohibited
  - Inter-world communications through the **Secure Monitor**
- TrustZone and Secure Boot are used to create a **Trusted Execution Environment** (TEE)
  - Authentication (e.g. for encrypted filesystem)
  - Mobile payment, secrets management, etc.
  - Content management (DRM)



Android Device Architecture

Secure Boot

- Each stage **cryptographically checks** that the next image is authorized to run
  - Creates a *chain of trust*
  - Starting from the *root of trust*, an **immutable component**
- Prevents unauthorized or modified software from executing on the device
- OEMs implement **additional features**
  - Anti-rollback mechanism
  - Emergency boot over USB
  - Boot images encryption

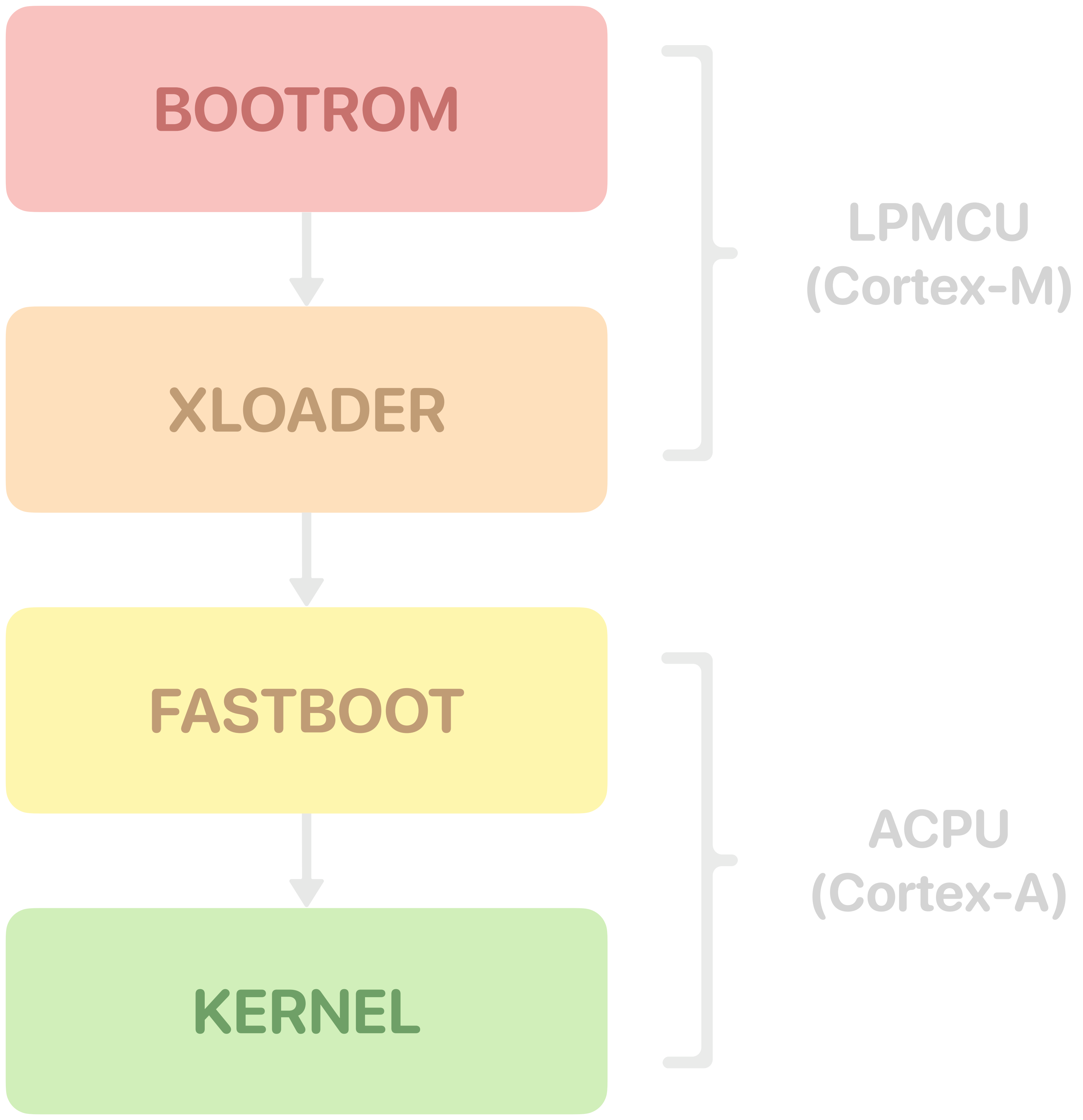


Boot Chain

Boot Chain

Overview

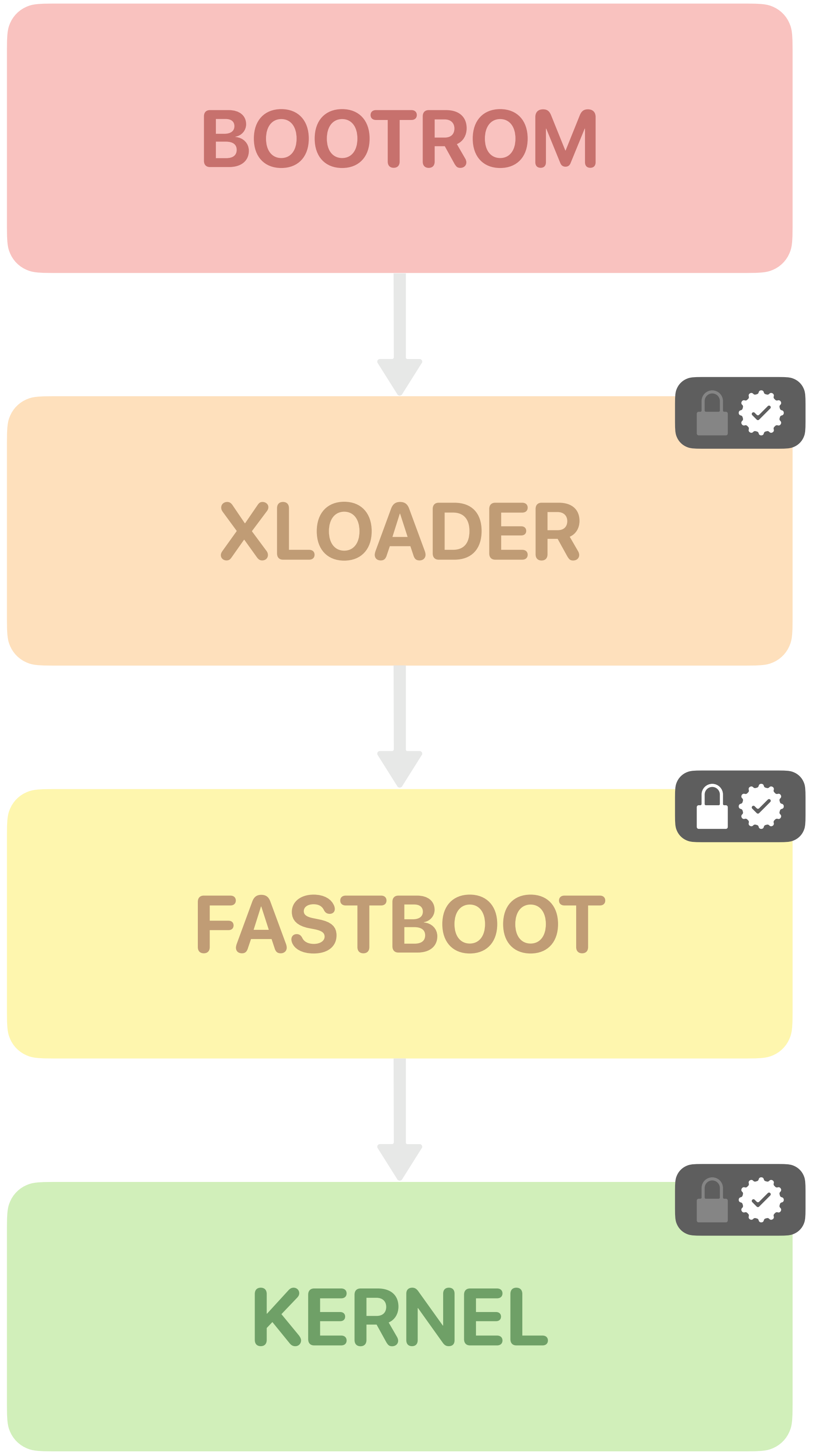
- **Security mechanisms**
  - **Secure boot:** prevents replacing or modifying boot chain images
  - **Bootloader lock:** prevents reflashing the partitions or running a custom kernel
- **Bootstrapping challenges**
  - All critical partitions are **encrypted**
  - Can't talk directly to targeted components
  - Countermeasures in kernel and userland
- Getting control over the boot chain
  - High entry cost: we need to find a **vulnerability** first



Boot Chain

First Research Device

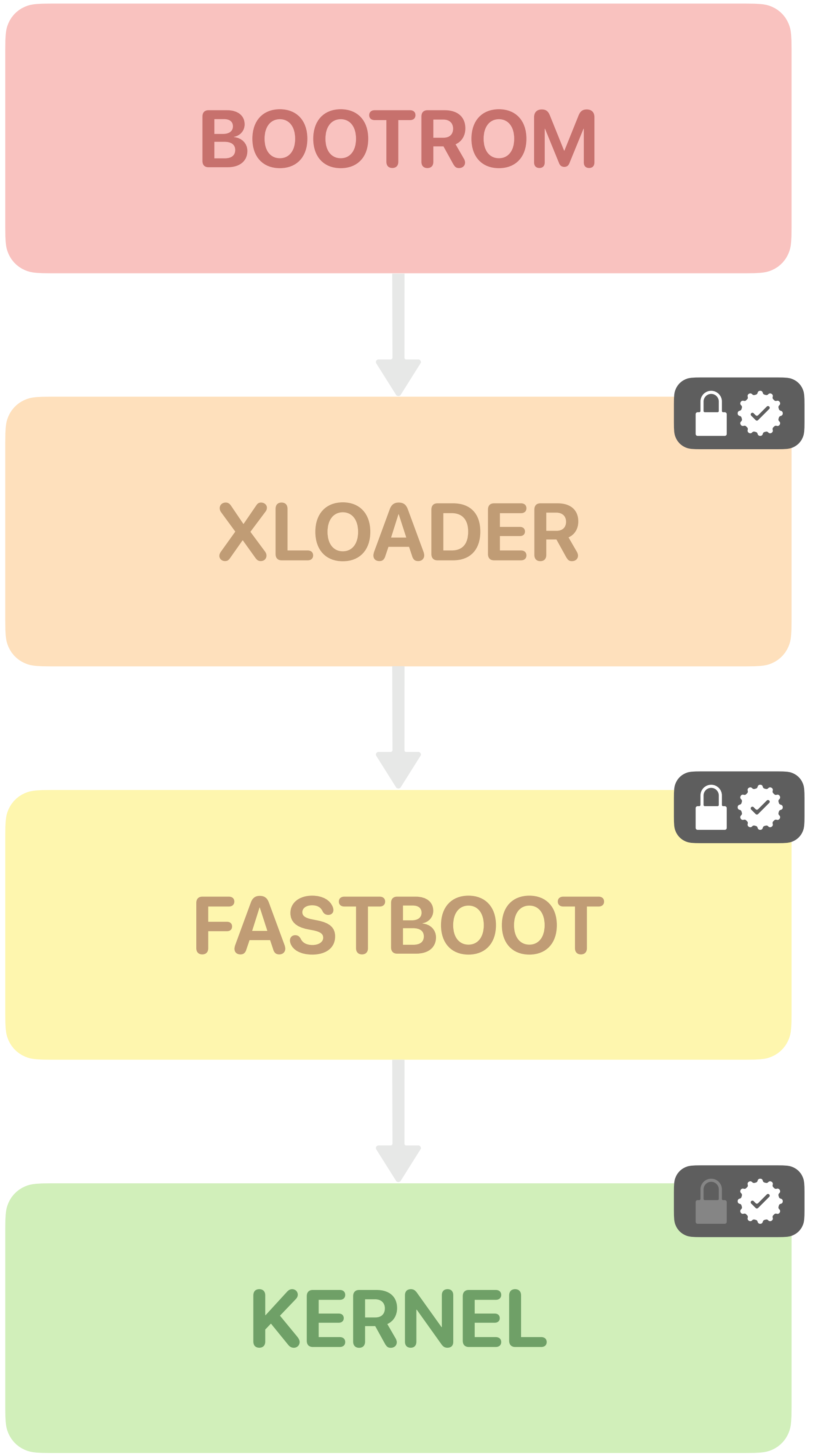
- **P30 Lite** (Kirin 710 chipset)
  - Xloader is **signed** but **not encrypted**, thus can be retrieved from a firmware update
  - Found a **vulnerability** in its implementation of *xmodem*, the USB recovery protocol
    - The next stage binary's base address is **not verified**
    - Can be leveraged to modify Xloader itself (all memory is RWX)
    - Shorting a **test point** on the device activates the download mode feature



Boot Chain

Second Research Device

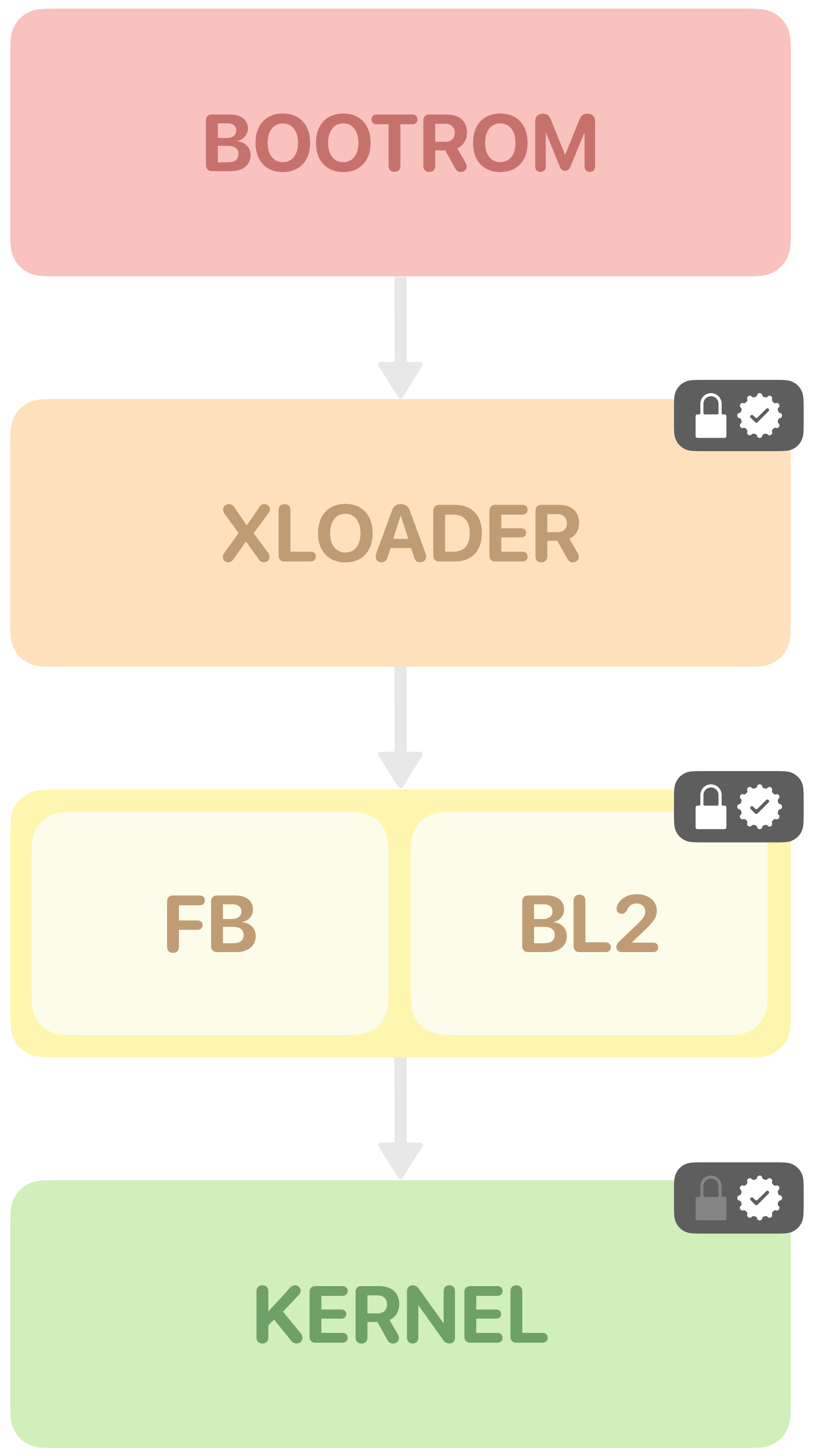
- **P40 Lite** (Kirin 810 chipset)
  - Xloader is **signed** and **encrypted**
  - But it is also affected by the *xmodem* vulnerability that needs to be exploited **blindly**
  - Decryption key no longer stored in fuses and is only accessible to the **crypto engine**
    - Firmware images are retrieved by using the device as an **oracle**



Boot Chain

Third Research Device

- **P40 Pro** (Kirin 990 chipset)
  - Xloader is **signed**, **encrypted**, but **not vulnerable** to the *xmodem* bug
  - Fastboot is **split** into a privileged and an unprivileged component
  - **Another vulnerability** is needed to get control over the boot chain



Boot Chain

How to Tame Your Unicorn

- Talk presented at *BlackHat USA 2021* by **Taszk Security Labs**
  - Revealed multiple Xloader and BootROM bugs
  - Including the Xloader vulnerability that we had discovered
- **CVE-2021-22434**: Head Chunk Resend State Machine Confusion
  - Internal state is not reset when sending an incorrect payload address
  - **BootROM** code execution can be achieved from this **arbitrary write primitive**
  - Must be exploited **blindly** on the Kirin 990 chipset
    - Dump Xloader using the *Flash Patch and Breakpoint* unit of the LPMCU
- Huawei “fixed” the BootROM bugs by burning a fuse to disable the USB recovery mode

Boot Chain

Continuation of Execution

Steps

- Send patched Xloader to the BootROM
- Force its execution by overwriting a return address

Steps

- Send patched Fastboot to Xloader
- Patches allow execution to continue normally

Patches

- Remove the address and length checks
- Disable decryption and signature verification

Patches

- Change boot mode from USB to UFS
- Ignore the Android Verified Boot failure

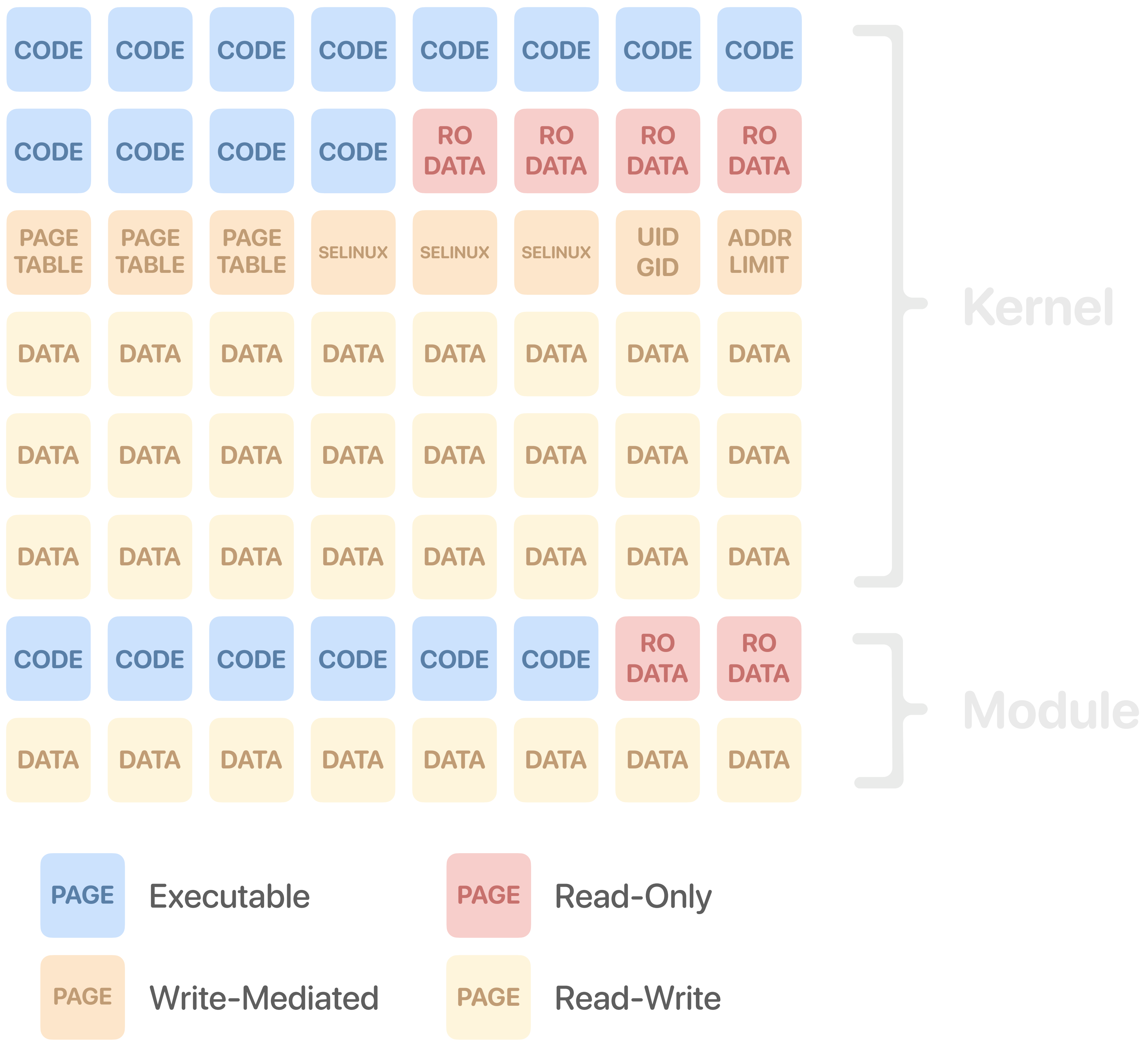
- Similarly to "CHECKM30" presented at *MOSEC 2021* by **Pangu Team**

Security Hypervisor

Security Hypervisor

Introduction

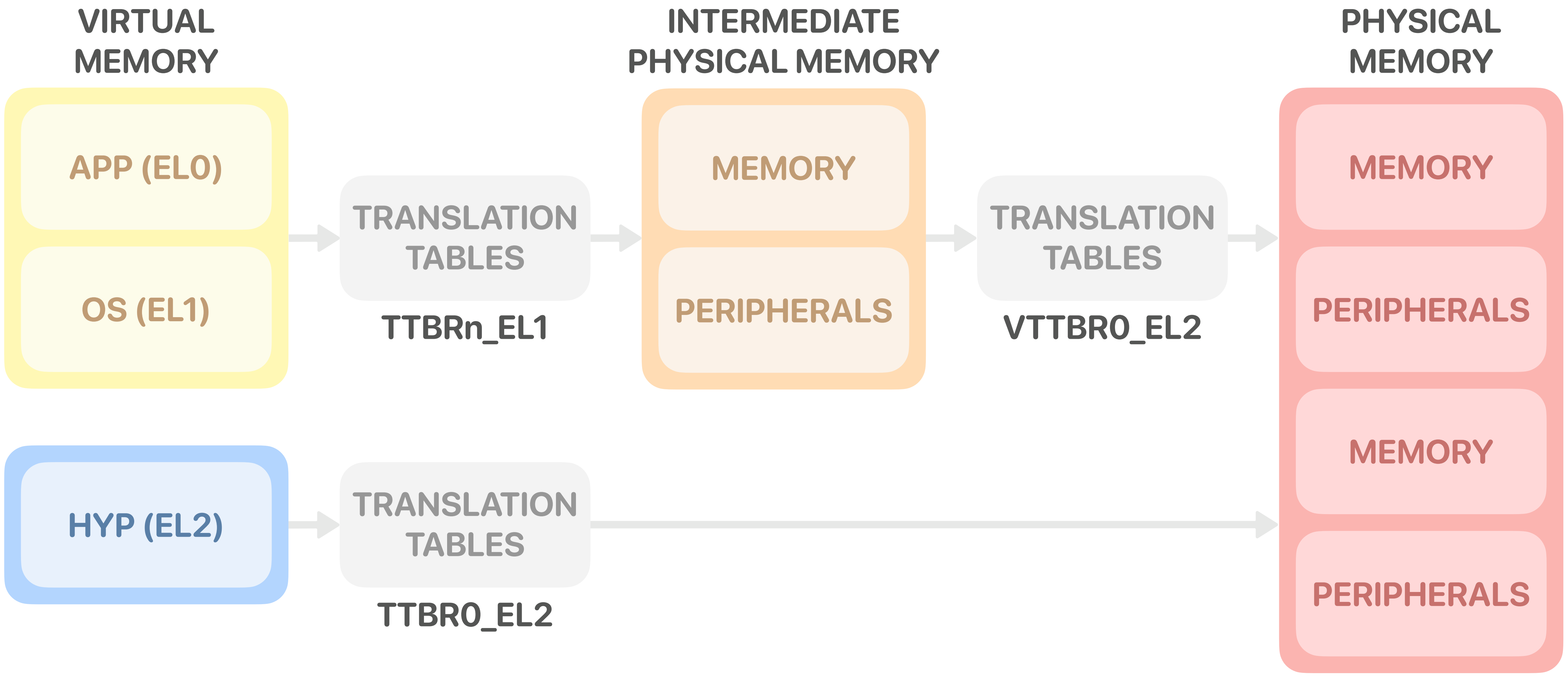
- Called **Huawei Hypervisor Execution Environment** (HHEE)
  - Similar to **uH/RKP** on Samsung's Exynos or **QHEE** on Qualcomm's Snapdragon
- **Main Security Features**
  - Prevents arbitrary changes to the kernel read-only data, its page tables, SELinux structures, etc.
  - Keeps a read-only copy of tasks' information to detect **privilege escalation** on the next syscall or file access
  - Ensures only the pages belonging to the **kernel** and **modules** code segment can be executed at EL1
  - Makes critical physical memory regions (e.g. sensorhub, secure npu, modem, etc.) inaccessible to EL0 and EL1
  - Enables **execute-only** user space memory that is unreadable from the kernel



Security Hypervisor

Second Stage of Address Translation

- Virtual address translation is extended with a **second stage**
  - The VA is first translated into an *Intermediate Physical Address*
  - The IPA is then translated into a PA
- It uses a second set of page tables under the control of the hypervisor
  - These page tables can apply **additional access control**
- The hypervisor also has its own page tables for its virtual address space



Security Hypervisor

Second Stage Limitations

XN[1]	XN[0]	Access
0	0	Executable at EL0 and EL1
0	1	Executable only at EL0
1	0	Not executable at EL0 or EL1
1	1	Executable only at EL1

With **FEAT_XNX**

XN[1]	XN[0]	Access
0	RES0	Executable at EL0 and EL1
1	RES0	Not executable at EL0 or EL1

Without **FEAT_XNX**

S2AP	EL1 and EL0 Access
00	None
01	Read-only
10	Write-only
11	Read/write

- Stage 2 permissions **cannot distinguish** between EL0 and EL1 for:
  - Read and write accesses
  - Executability, if *FEAT_XNX* is not implemented
- It is the main reason stage 1 page tables also need to be **controlled** by the hypervisor

Security Hypervisor

Kernel Page Tables

- **Initial processing**
  - Traps changes made to the *TTBR1_EL1* and *SCTRL_EL1* system registers
  - Performs a page table walk and ensures every descriptor is sane and coherent
    - e.g. descriptors with the contiguous bit set actually point to contiguous memory
  - Enforces **EL0/EL1 distinction** for read-write accesses and executability
    - By default, kernel pages are set non executable at EL1 and non accessible at EL0
- **Changes monitoring**
  - Kernel page tables are set as **read-only** in the second stage
    - Except when permissions can be enforced at previous table level (PXNTable/APTable)
  - A **write** to a stage 1 descriptor or a **translation fault** during a page table walk raises an exception
    - Handled by the hypervisor to ensure modifications are permitted and update stage 2 accordingly

Security Hypervisor

Software Attributes

Attrs	Description
0b0000	Unmarked
0b0100	Level 0 Page Table
0b0101	Level 1 Page Table
0b0110	Level 2 Page Table
0b0111	Level 3 Page Table
0b1000	OS Read-Only
0b1001	OS Module Read-Only
0b1010	Hyp-mediated OS Read-Only
0b1011	Hyp-mediated OS Module Read-Only
0b1100	Shared Obj Protection Execute-Only

- **Hypervisor Software Attributes**
  - Bitfield stored in bits [58:55] of a stage 2 descriptor
  - Contains **usage information** about the underlying memory region
  - Used to prevent **disallowed changes** to protected memory
    - e.g. making a OS read-only page writable again
- **Rules** enforced while modifying them
  - Only **unmarked** descriptors can be marked
  - To unmark a descriptor, the **current marking** must be provided

Security Hypervisor

Methodology

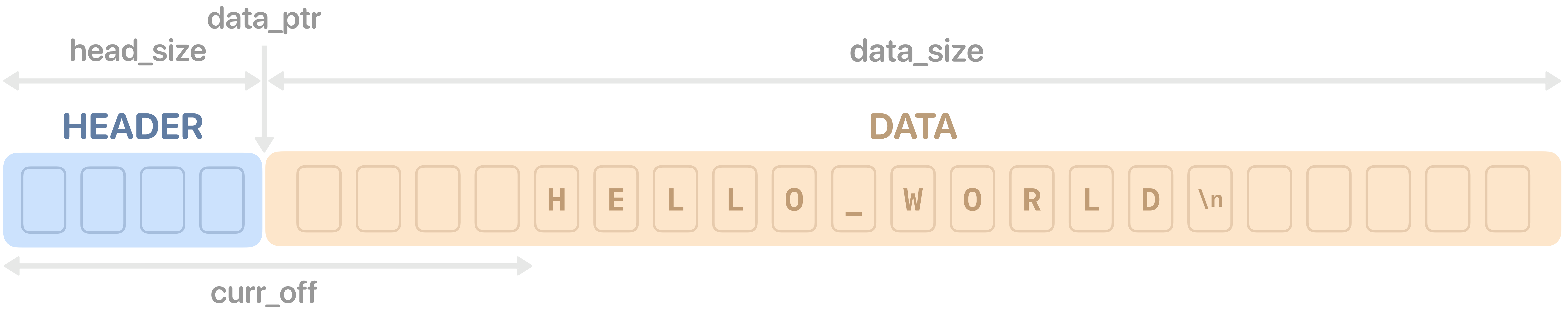
- Extensive **reverse engineering**
  - **Static analysis**
    - 68 KB raw binary
    - AArch64 code
    - 295 functions
    - No symbols
    - ~10 log strings
  - Analysis can be augmented with information coming from external sources
    - HVC names from the kernel source code
    - *Armv8-A Architecture Reference Manual*

- Identifying the **attack surface**
  - HVC and SMC handlers
  - Faulting memory accesses
  - Trapped system registers accesses
    - e.g. *SCTLR_EL1*, *TCR_EL1*, etc.
  - Memory shared with the kernel

- **Comparing** the security hypervisors of different OEMs might highlight implementation flaws

Security Hypervisor

Vulnerability

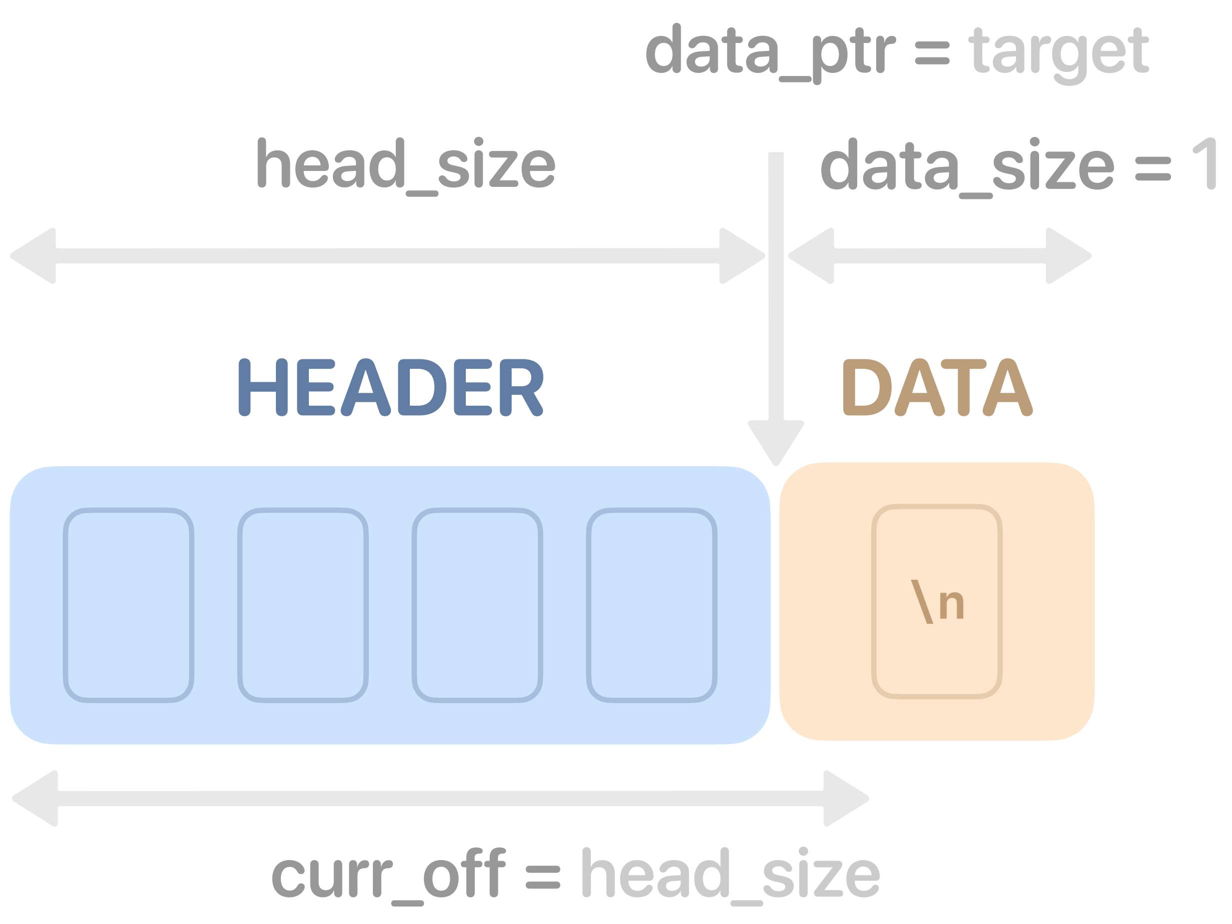


- **CVE-2021-39979**
  - Logging system use a control structure located in **shared memory** that is accessible to the kernel
  - Pointer, offset and sizes fields are all **unchecked**
  - We can **write log strings** at any virtual address that is mapped into the hypervisor

Security Hypervisor

Exploitation

- **Constrained write primitive**
  - The log string being written is **not user-controlled**
  - Since the buffer is **circular** and written character by character
    - Only the **last byte** will remain in memory if we set the data size of the buffer to 1
    - It's always the **new line** character: \n (0xA)
- **Linear heap allocator**
  - Heap region has a fixed base address and size
    - The current offset is stored in a **global variable**
  - The allocation function **assumes** the offset value is sane (smaller than the heap size)
    - If it isn't, an integer underflow happens and the allocator returns out-of-bounds memory
  - Right after the heap is a **kernel-accessible** region

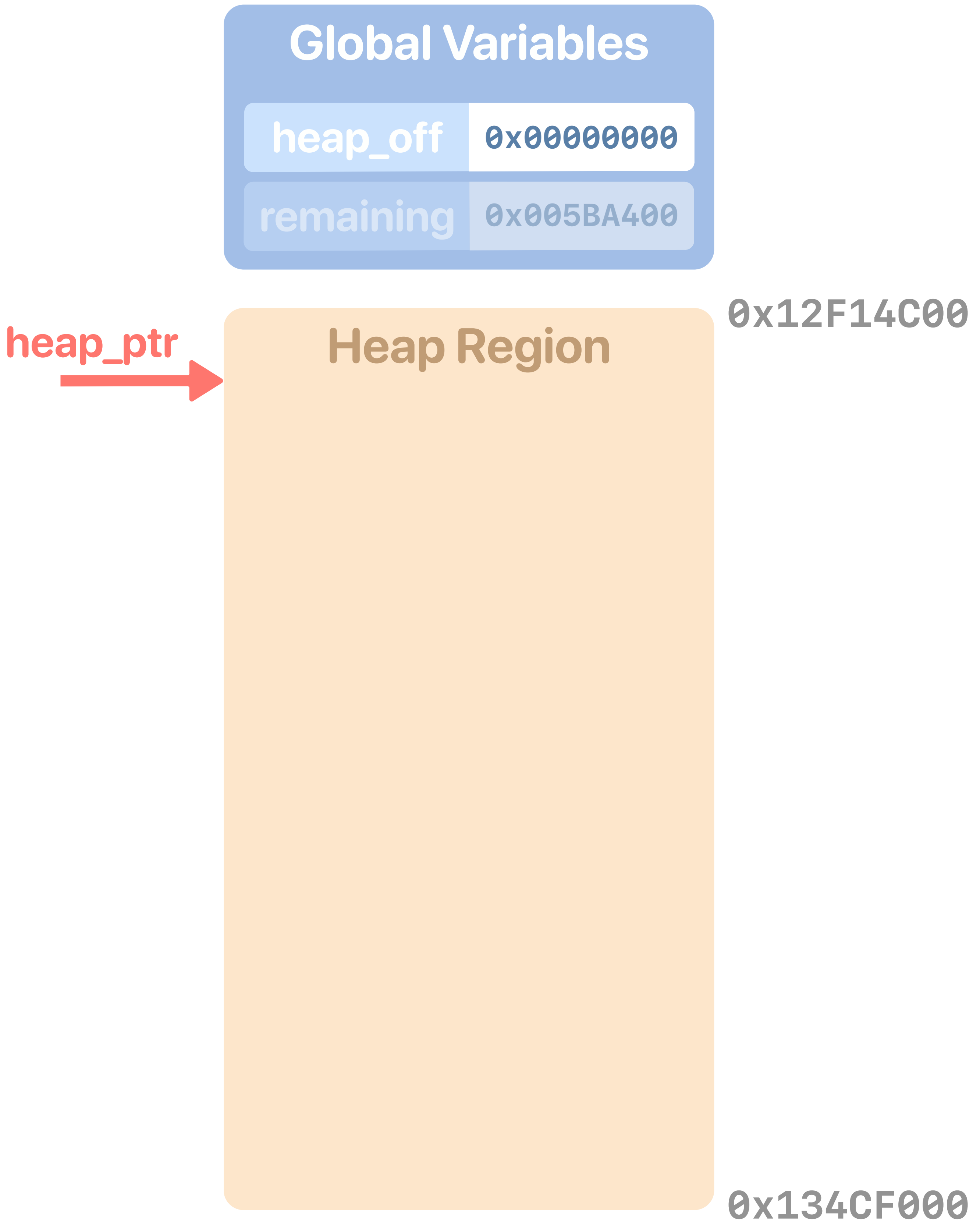



```c
void malloc(uint64_t size) {
    if (HEAP_SIZE - heap_off < pad + size)
        return 0;
    heap_off += pad + size;
    return HEAP_ADDR + heap_off + pad;
}
```

Security Hypervisor

Exploitation

- Getting code execution



Security Hypervisor

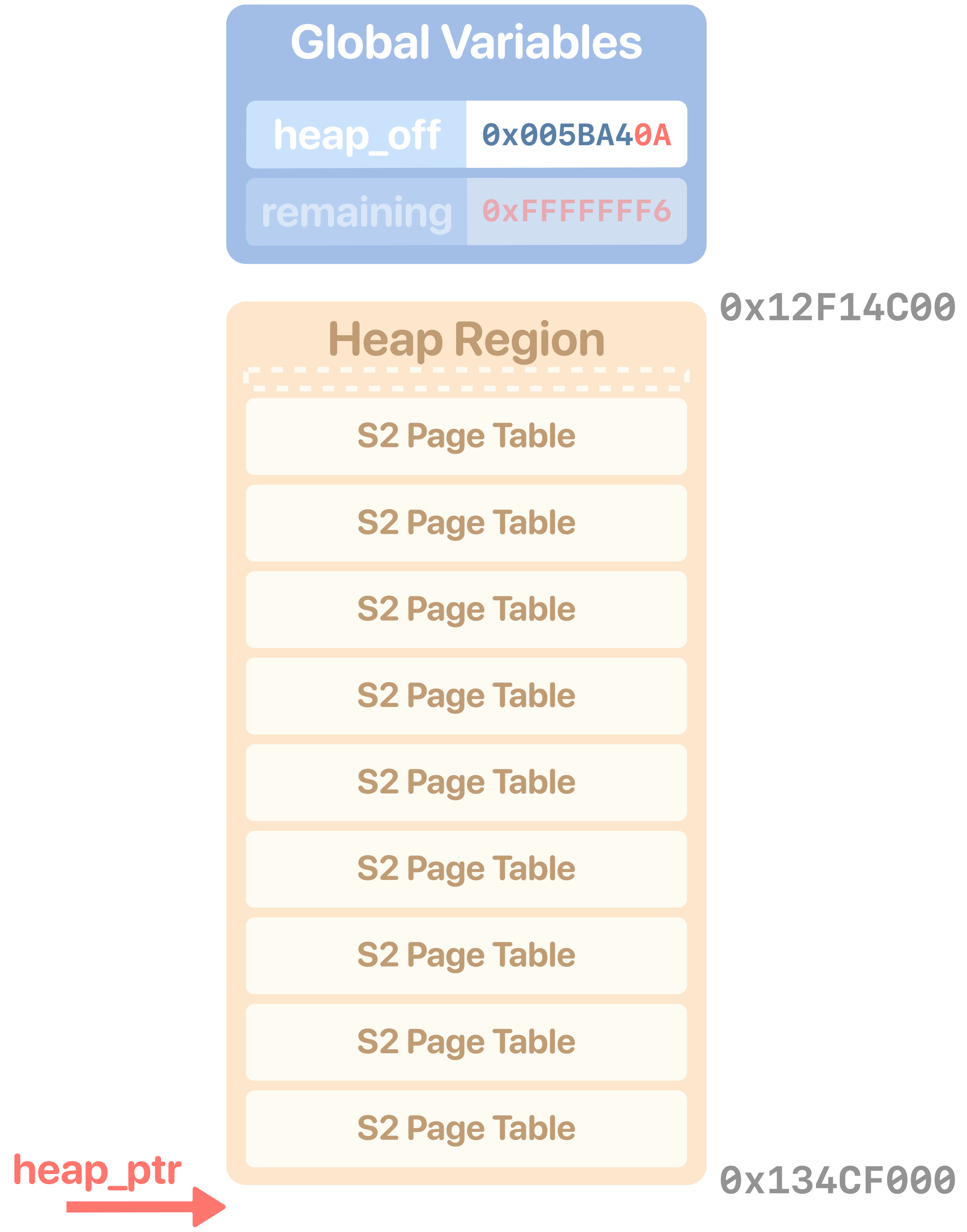
Exploitation

- Getting code execution
  - **Step 1:** Fill up the heap to its maximum by triggering **stage 2 page tables** allocations

Security Hypervisor

Exploitation

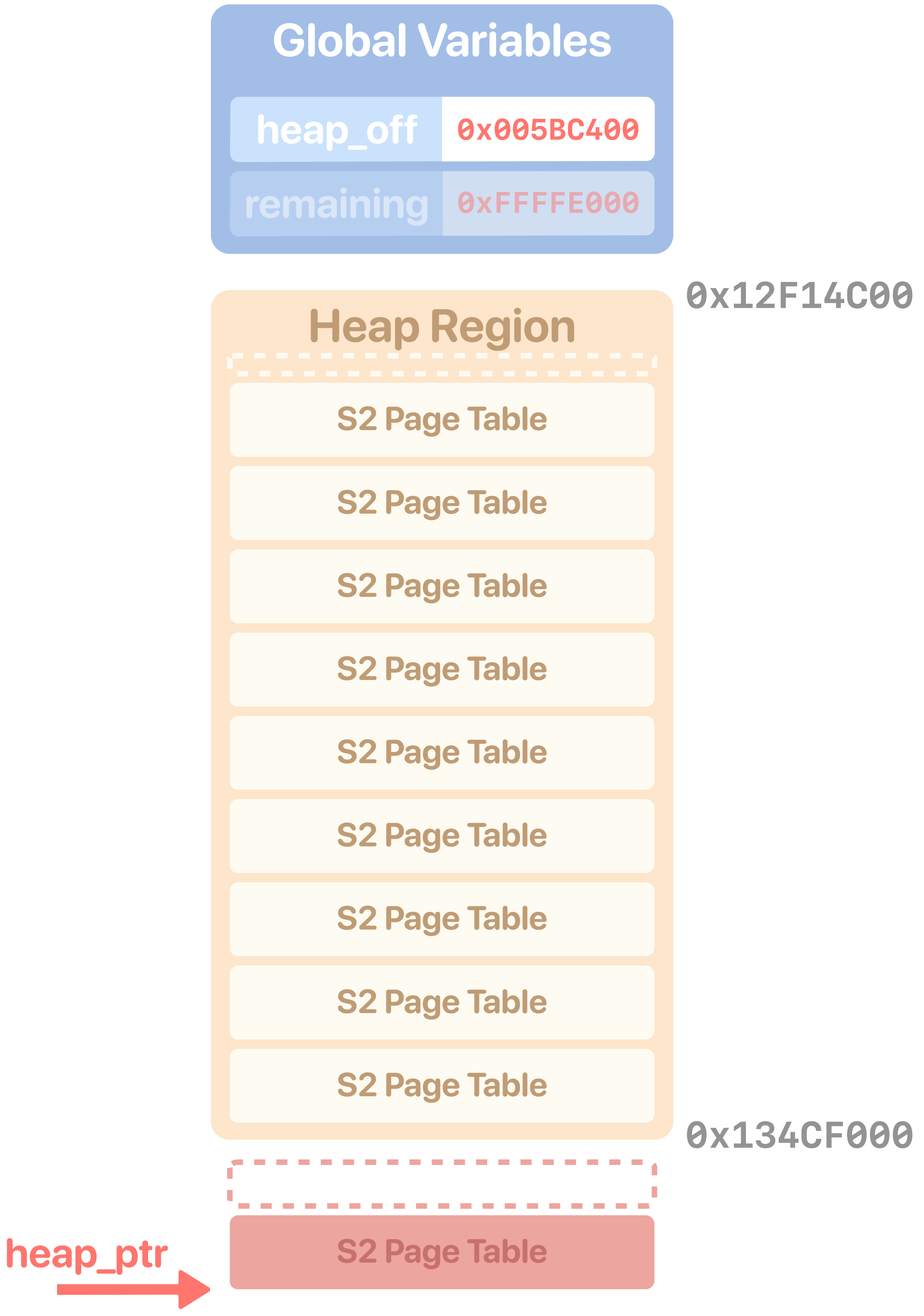
- **Getting code execution**
  - **Step 1:** Fill up the heap to its maximum by triggering **stage 2 page tables** allocations
  - **Step 2:** Use the constrained write primitive to move the offset right past the end of heap



Security Hypervisor

Exploitation

- Getting code execution
  - **Step 1:** Fill up the heap to its maximum by triggering **stage 2 page tables** allocations
  - **Step 2:** Use the constrained write primitive to move the offset right past the end of heap
  - **Step 3:** Trigger a last stage 2 page table allocation that is made **out-of-bounds** because of the **integer underflow**



Security Hypervisor

Exploitation

- **Getting code execution**
  - **Step 1:** Fill up the heap to its maximum by triggering **stage 2 page tables** allocations
  - **Step 2:** Use the constrained write primitive to move the offset right past the end of heap
  - **Step 3:** Trigger a last stage 2 page table allocation that is made **out-of-bounds** because of the **integer underflow**

S2 Page Table

0x10000000	0x10000000	RO
0x10001000	0x10001000	RO
0x10002000	0x10002000	RO
...	...	...
0x101FD000	0x101FD000	RO
0x101FE000	0x101FE000	RO
0x101FF000	0x101FF000	RO

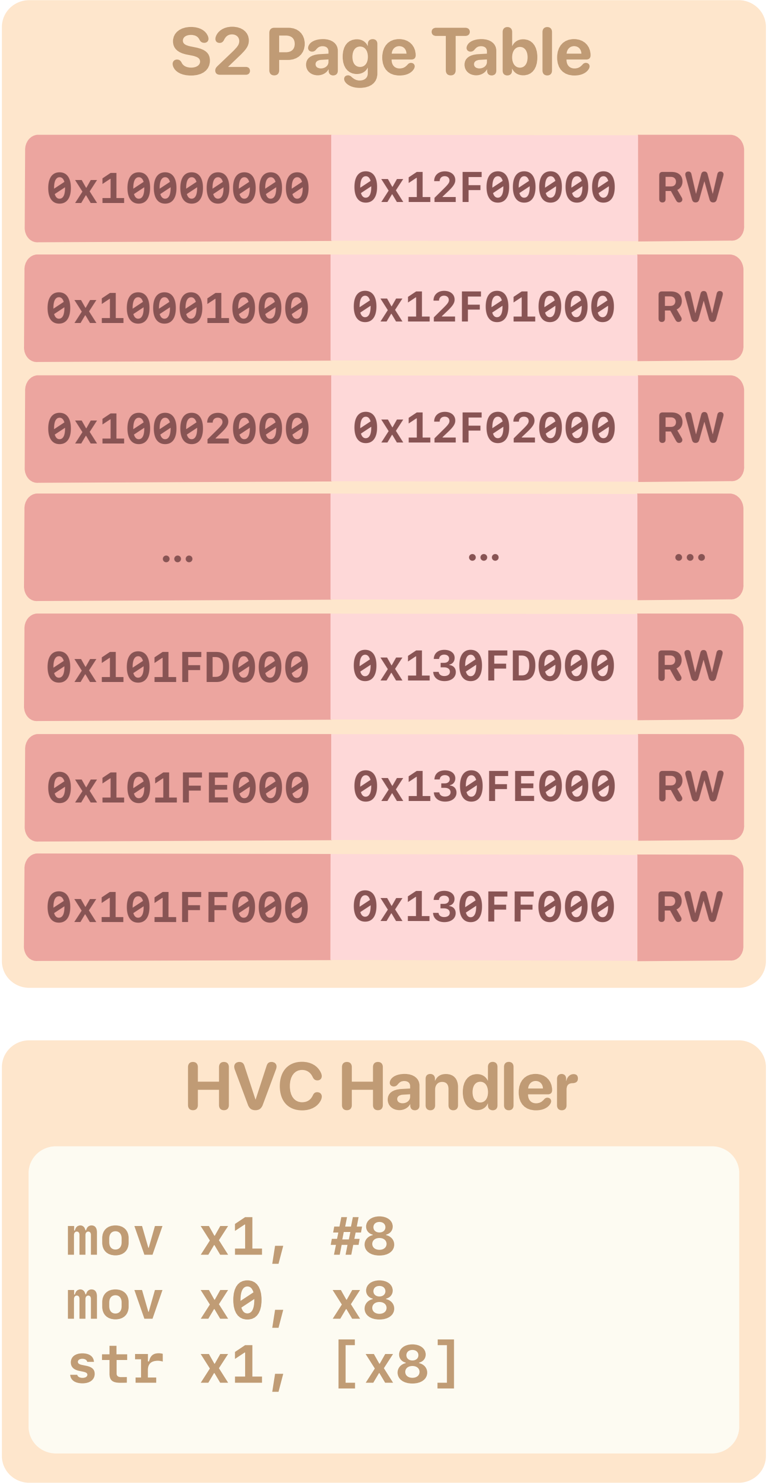
HVC Handler

```
mov x1, #8
mov x0, x8
str x1, [x8]
```

Security Hypervisor

Exploitation

- **Getting code execution**
  - **Step 1:** Fill up the heap to its maximum by triggering **stage 2 page tables** allocations
  - **Step 2:** Use the constrained write primitive to move the offset right past the end of heap
  - **Step 3:** Trigger a last stage 2 page table allocation that is made **out-of-bounds** because of the **integer underflow**
  - **Step 4:** Change the page table from the kernel to **remap** the hypervisor as **read-write**



Security Hypervisor

Exploitation

- **Getting code execution**
  - **Step 1:** Fill up the heap to its maximum by triggering **stage 2 page tables** allocations
  - **Step 2:** Use the constrained write primitive to move the offset right past the end of heap
  - **Step 3:** Trigger a last stage 2 page table allocation that is made **out-of-bounds** because of the **integer underflow**
  - **Step 4:** Change the page table from the kernel to **remap** the hypervisor as **read-write**
  - **Step 5:** Patch the hypervisor memory and get code execution at EL2 from EL1
    - e.g. targeting one of the HVC handlers

S2 Page Table

0x10000000	0x12F00000	RW
0x10001000	0x12F01000	RW
0x10002000	0x12F02000	RW
...	...	...
0x101FD000	0x130FD000	RW
0x101FE000	0x130FE000	RW
0x101FF000	0x130FF000	RW

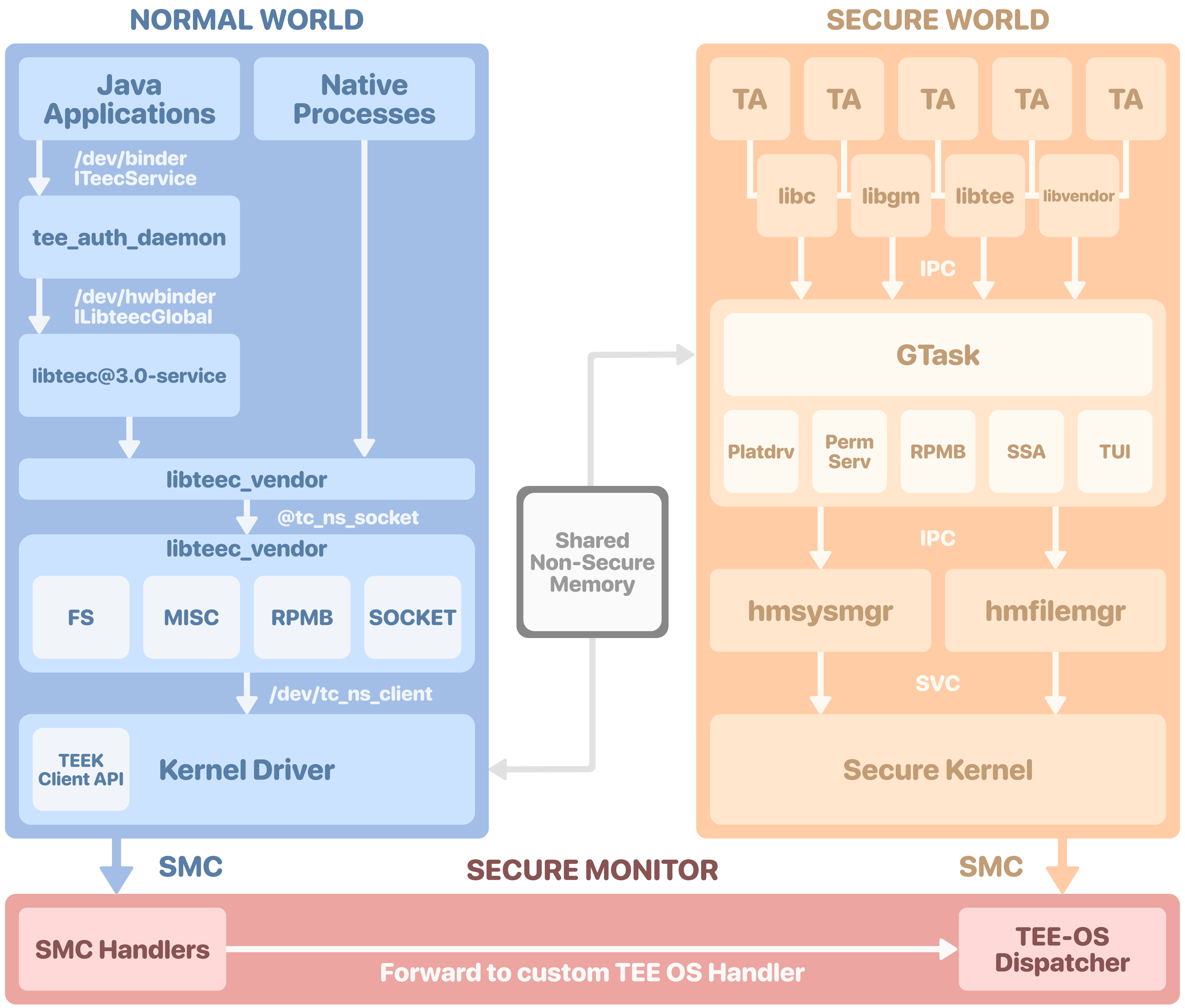
HVC Handler

```
mrs x0, CurrentEL
str x0, [x8]
ret
```

TrustZone

TrustZone

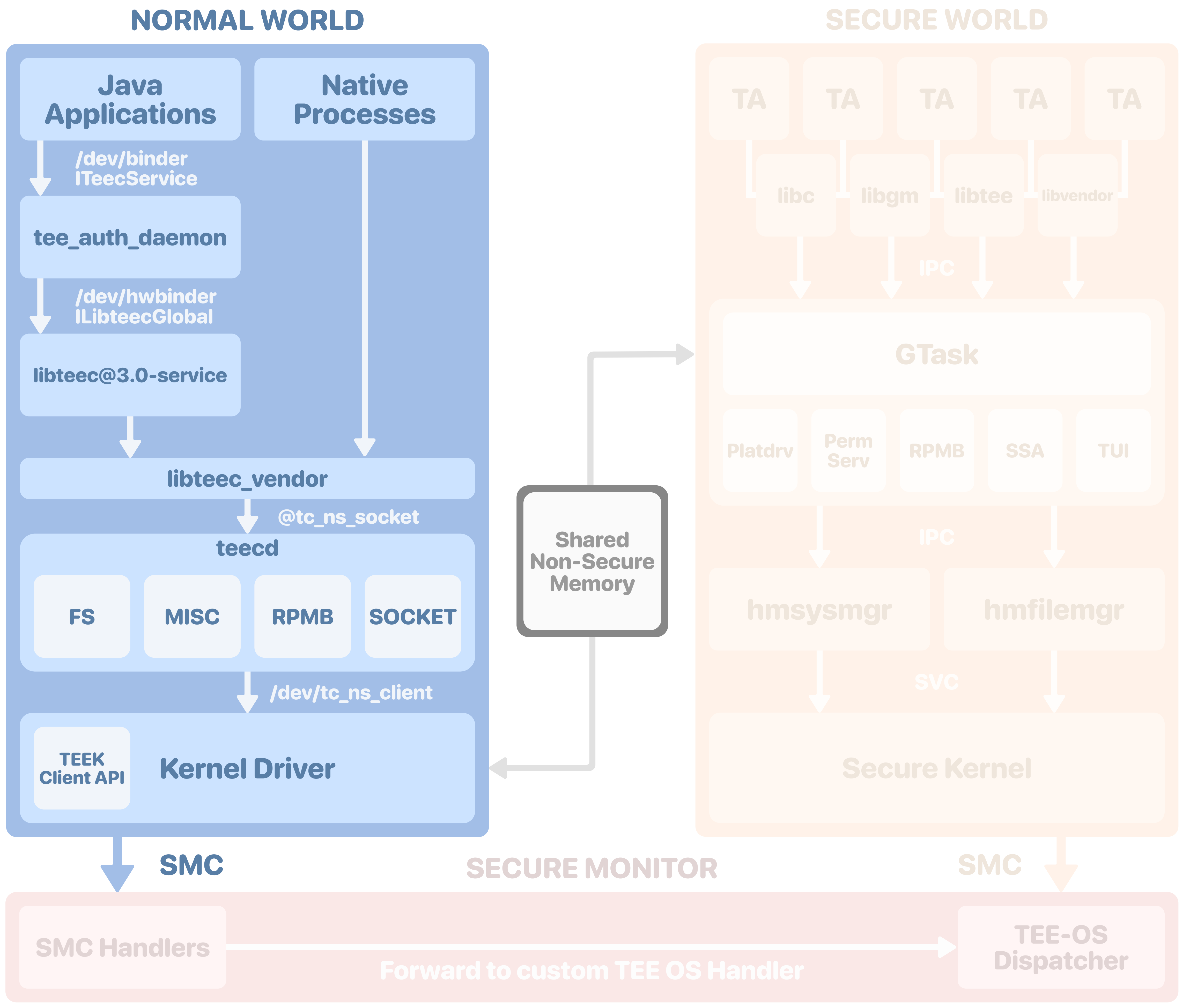
Overview



TrustZone

Normal World Overview

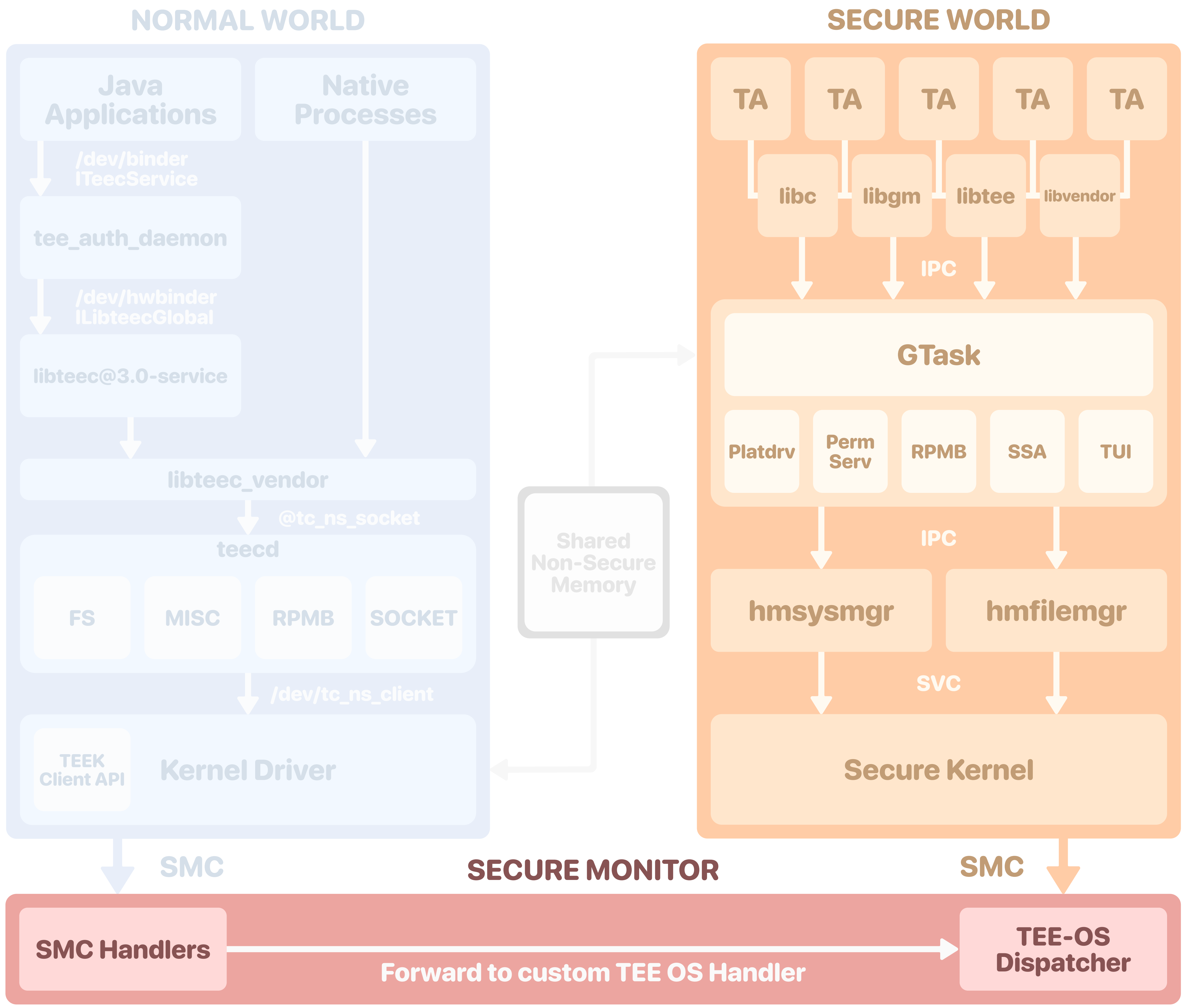
- **Java applications & native processes**
  - **Main users** of secure world features
  - But not privileged enough to send requests to the Secure World
    - Use the **kernel** as a proxy
- Steps to **send messages** to the Secure World from **userland**
  - Requests are received by the userland daemon *teecd*
    - First go through *tee_auth_daemon* for Java applications
  - And then forwarded to the kernel through the character device *tc_ns_client*
    - Implements the **agents** (filesystem, networking, etc.)
    - Provides a **shared library** to communicate with it
  - The kernel then sends the requests to the Secure World through an SMC
- Each interface has its own **SELinux context** to restrict access



TrustZone

Secure World Overview

- **Secure Monitor**
  - Handles SMCs and forwards requests to the trusted OS
- **Trusted OS**
  - Based on a micro-kernel architecture
  - Trusted applications running on top of privileged tasks and drivers

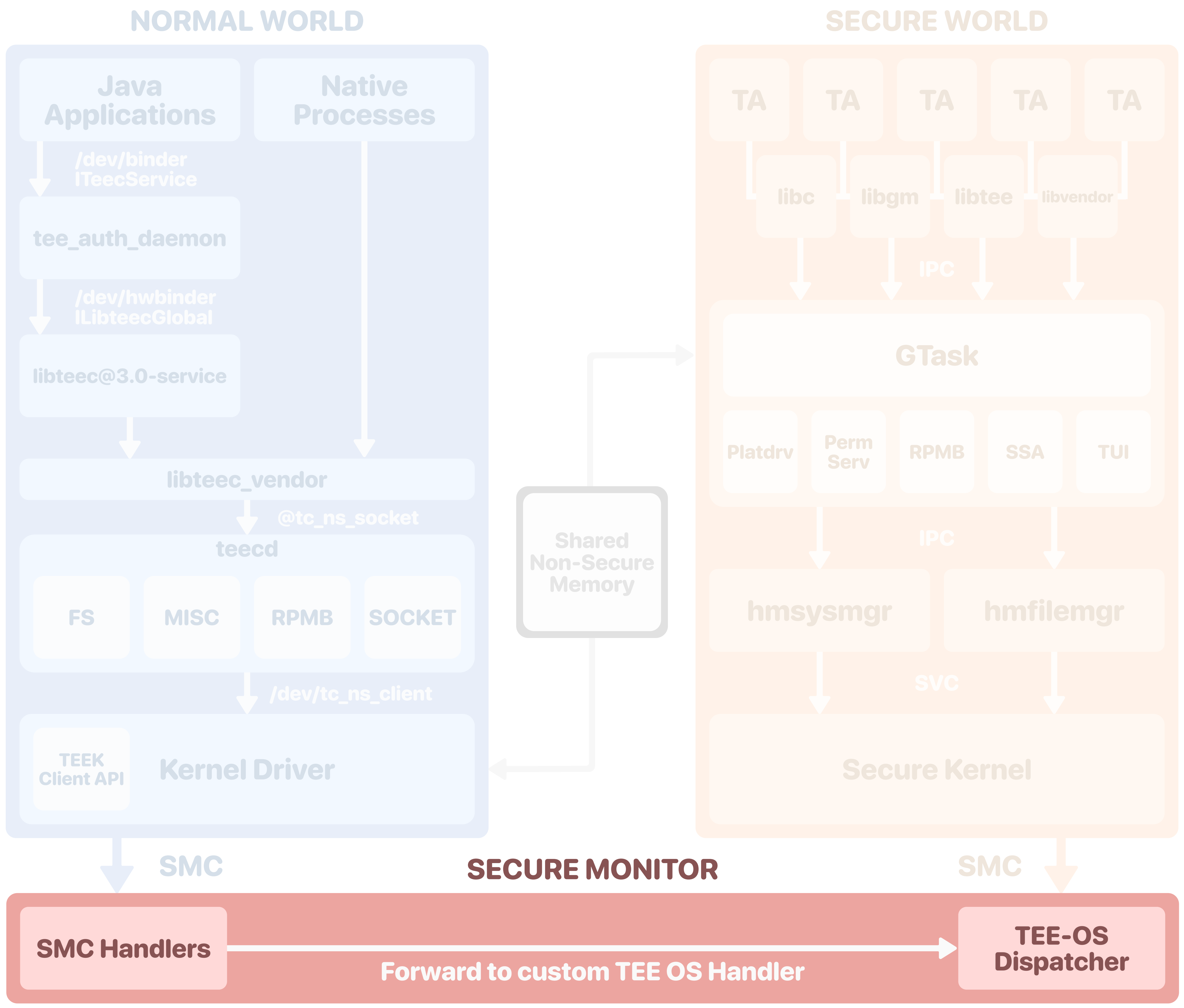


Secure Monitor

Secure Monitor

Introduction

- Executes at **EL3**, the highest privilege level
  - Performs **privileged operations** and manages critical hardware **peripherals**
    - e.g. efuses, power controls, RPMB, etc.
  - Bridge between the **Normal** and **Secure Worlds**
    - Forwards requests between the kernel and the trusted OS
- Huawei's implementation based on the **ARM Trusted Firmware** (ATF)
  - Open source, probably heavily reviewed
- Huawei implemented additional **runtime services**
  - These handlers are more likely to be vulnerable



Secure Monitor

Vulnerability

- **CVE-2021-39994**
  - Secure Monitor acts as a pass-through for the kernel to interact with the **Secure Element** (SE)
  - A response from the SE uses the *user_data* structure where the user controls:
    - The address of *user_data,* that contains the response **metadata**
    - The address and size of the reponse **data**: *user_data.addr* and *user_data.size*
  - **Bounds check**
    - The user-provided addresses for *user_data* and *user_data.addr* must be in a specific **world-shared memory buffer**
    - However, in one of the requests, the **check is missing** for *user_data*
  - Information about the SE's response is thus written at a **user-controlled address**
    - The response code *0xAABBCC55* at offset 4
    - The response size in the range *0x0-0xC* at offset 0xC
    - The response data address *user_data.addr*, which is **checked**

```
struct {
  uint32_t unkn;
  uint32_t code;
  uint32_t addr;
  uint32_t size;
} user_data;

uint32_t user_size;

/* check(user_data, user_size) is missing */
void on_reply(uint32_t addr, uint32_t size) {
  user_data.code = 0xAABBCC55;
  user_data.size = min(size, user_size);
  if (check(user_data.addr, user_data.size))
    memcpy(user_data.addr, addr, user_data.size);
}
```

Secure Monitor

Exploitation

- **Step 1:** Use the response **metadata** to disable the check on the shared memory region
  - Allows copying the response **data** at an arbitrary *user_data.addr*
  - Data isn't controlled either, but gives us more options

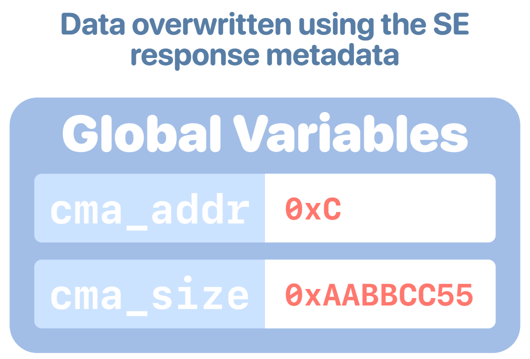
Data overwritten using the SE response metadata

Global Variables	
cma_addr	0x40000000
cma_size	0x10000000

Secure Monitor

Exploitation

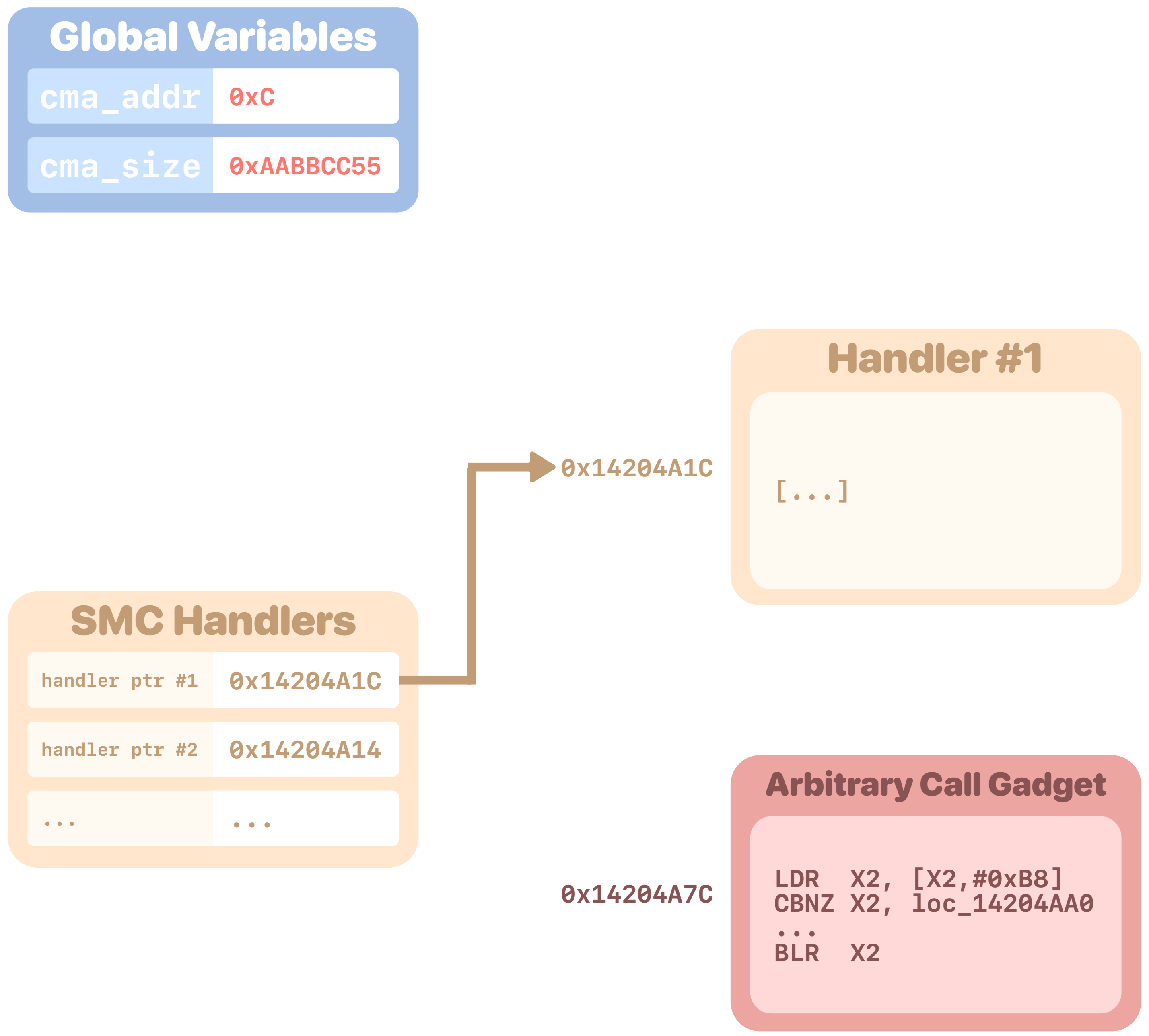
- **Step 1:** Use the response **metadata** to disable the check on the shared memory region
  - Allows copying the response **data** at an arbitrary *user_data.addr*
  - Data isn't controlled either, but gives us more options



Secure Monitor

Exploitation

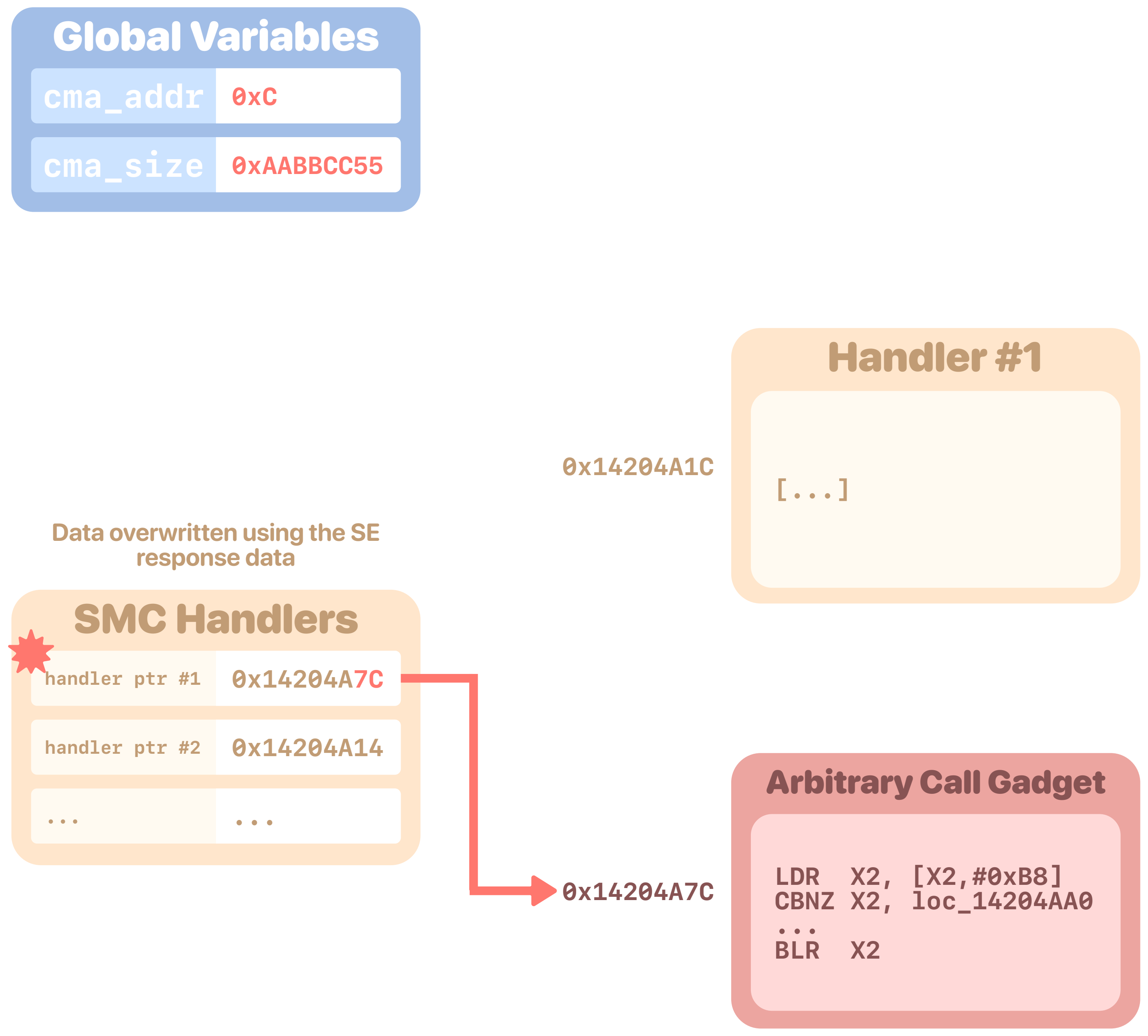
- **Step 1:** Use the response **metadata** to disable the check on the shared memory region
  - Allows copying the response **data** at an arbitrary *user_data.addr*
  - Data isn't controlled either, but gives us more options
- **Step 2:** Hijack a SMC handler pointer
  - 1-byte overwrite by specifying a **response size** of 1
  - Change an existing function pointer to an interesting gadget
    - BLR X2 —> arbitrary function call



Secure Monitor

Exploitation

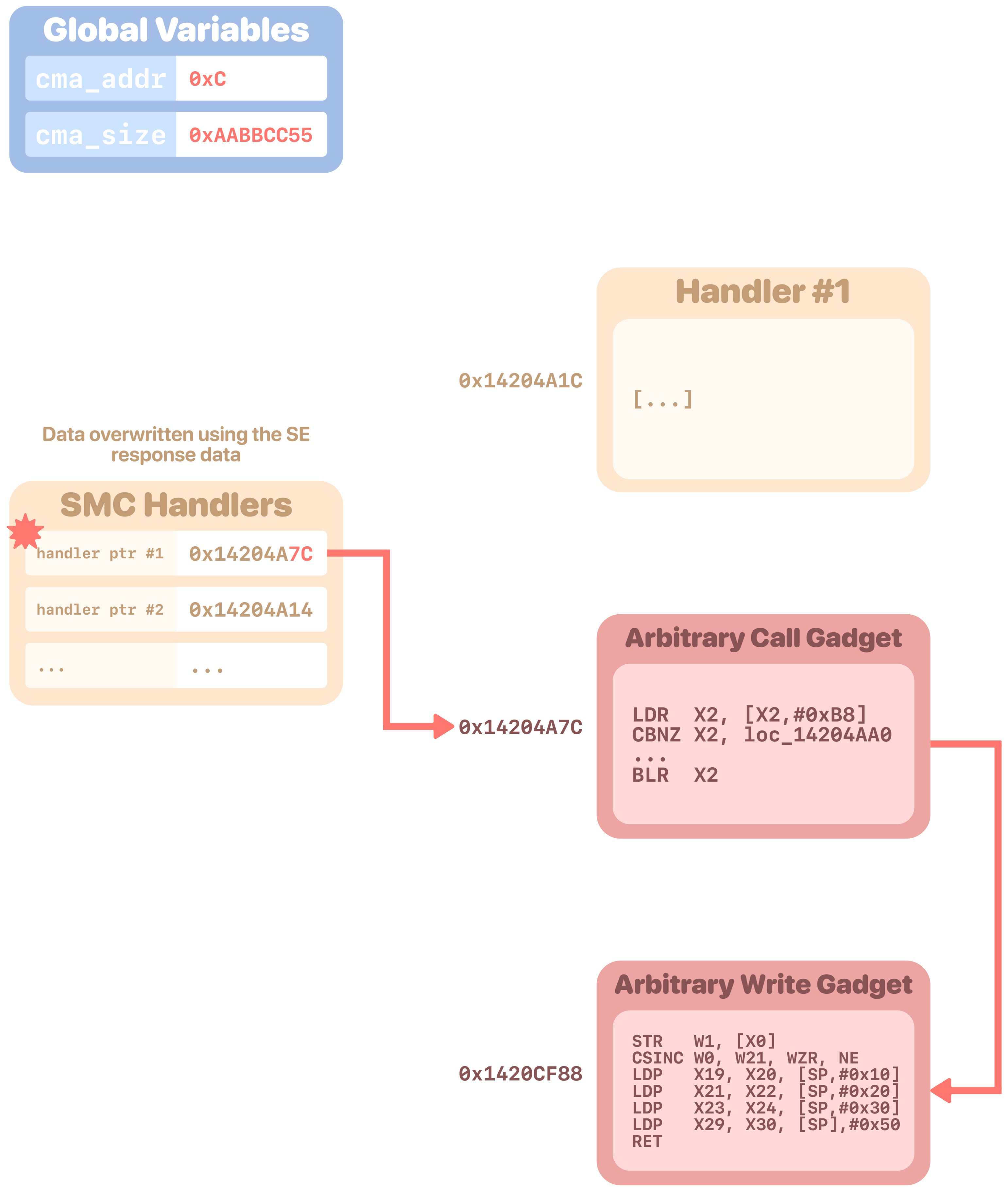
- **Step 1:** Use the response **metadata** to disable the check on the shared memory region
  - Allows copying the response **data** at an arbitrary *user_data.addr*
  - Data isn't controlled either, but gives us more options
- **Step 2:** Hijack a SMC handler pointer
  - 1-byte overwrite by specifying a **response size** of 1
  - Change an existing function pointer to an interesting gadget
    - BLR X2 —> arbitrary function call



Secure Monitor

Exploitation

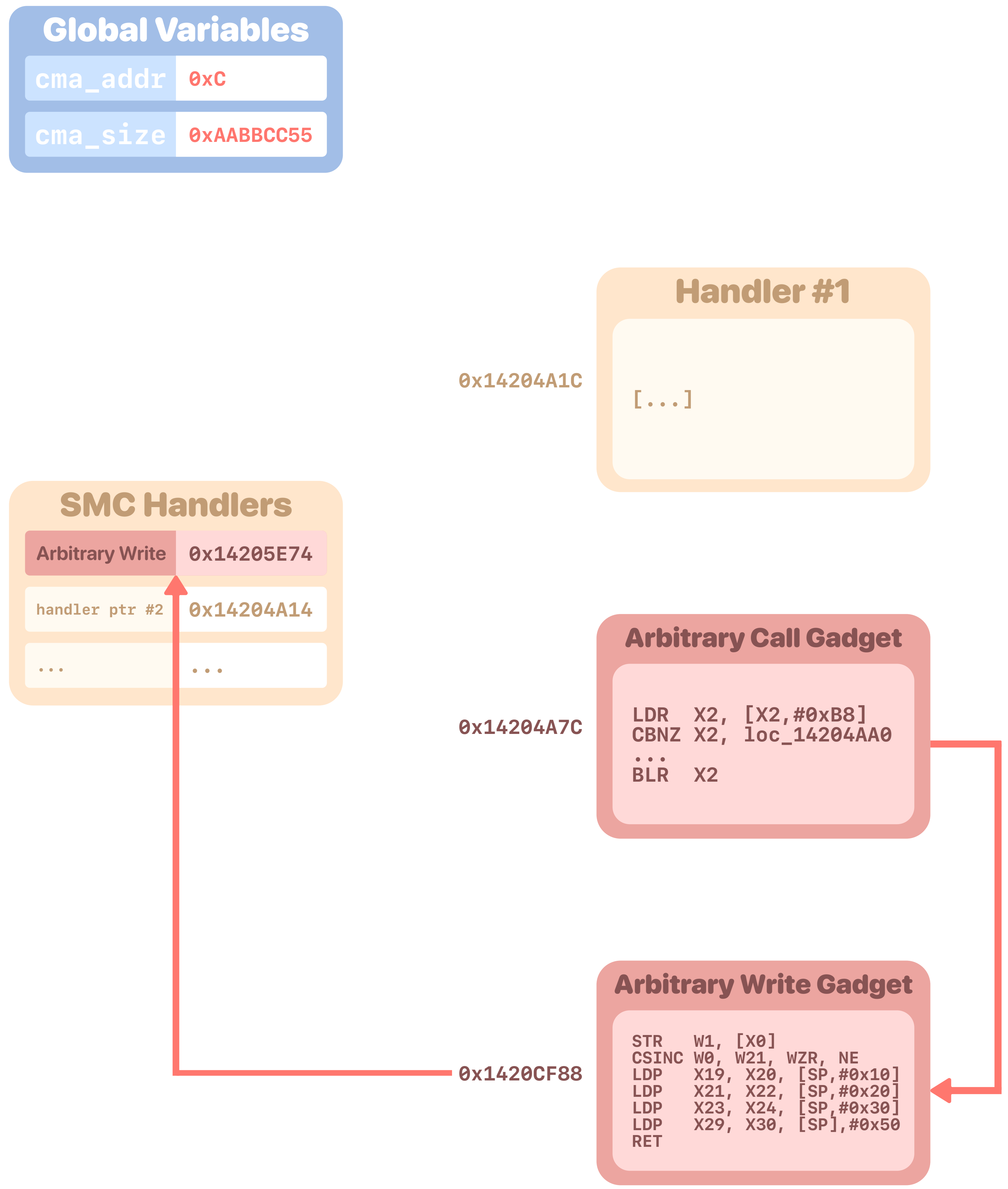
- **Step 1:** Use the response **metadata** to disable the check on the shared memory region
  - Allows copying the response **data** at an arbitrary *user_data.addr*
  - Data isn't controlled either, but gives us more options
- **Step 2:** Hijack a SMC handler pointer
  - 1-byte overwrite by specifying a **response size** of 1
  - Change an existing function pointer to an interesting gadget
    - BLR X2 —> arbitrary function call
- **Step 3:** Call a write gadget to create stable read and write primitives



Secure Monitor

Exploitation

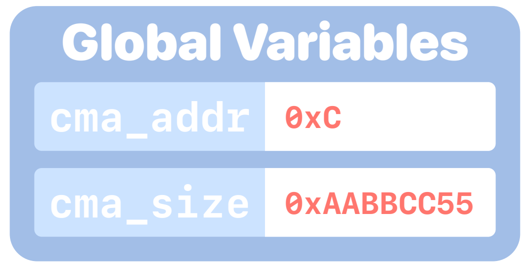
- **Step 1:** Use the response **metadata** to disable the check on the shared memory region
  - Allows copying the response **data** at an arbitrary *user_data.addr*
  - Data isn't controlled either, but gives us more options
- **Step 2:** Hijack a SMC handler pointer
  - 1-byte overwrite by specifying a **response size** of 1
  - Change an existing function pointer to an interesting gadget
    - BLR X2 —> arbitrary function call
- **Step 3:** Call a write gadget to create stable read and write primitives

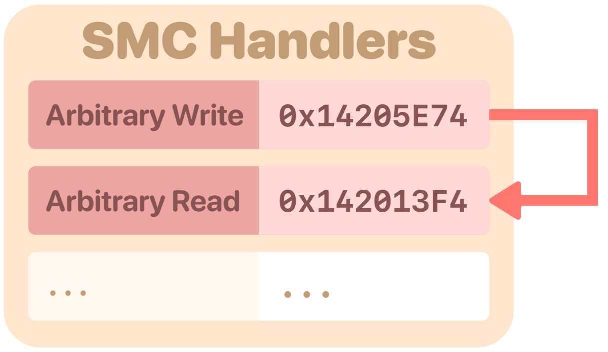


Secure Monitor

Exploitation

- **Step 1:** Use the response **metadata** to disable the check on the shared memory region
  - Allows copying the response **data** at an arbitrary *user_data.addr*
  - Data isn't controlled either, but gives us more options
- **Step 2:** Hijack a SMC handler pointer
  - 1-byte overwrite by specifying a **response size** of 1
  - Change an existing function pointer to an interesting gadget
    - BLR X2 —> arbitrary function call
- **Step 3:** Call a write gadget to create stable read and write primitives

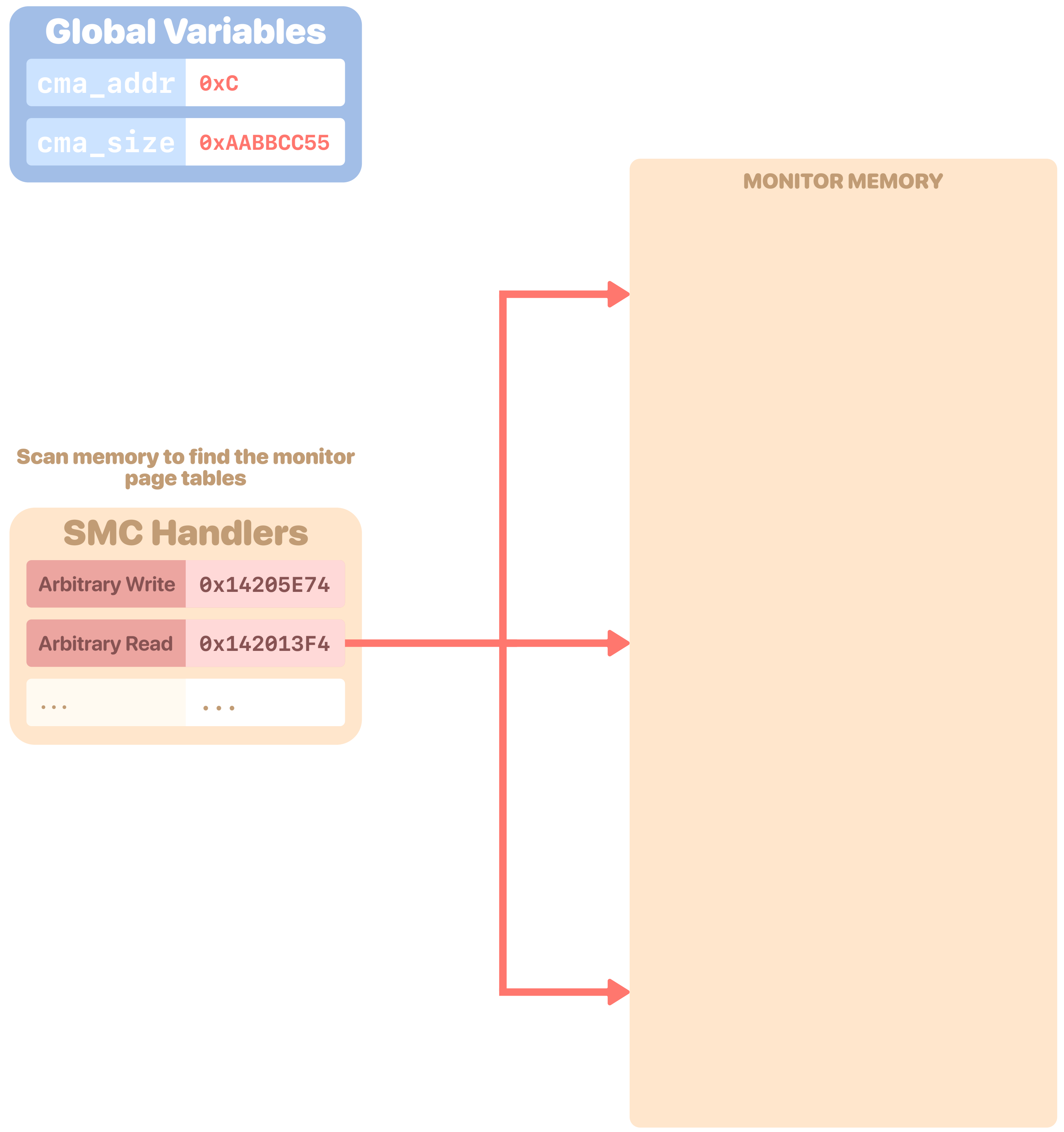




Secure Monitor

Exploitation

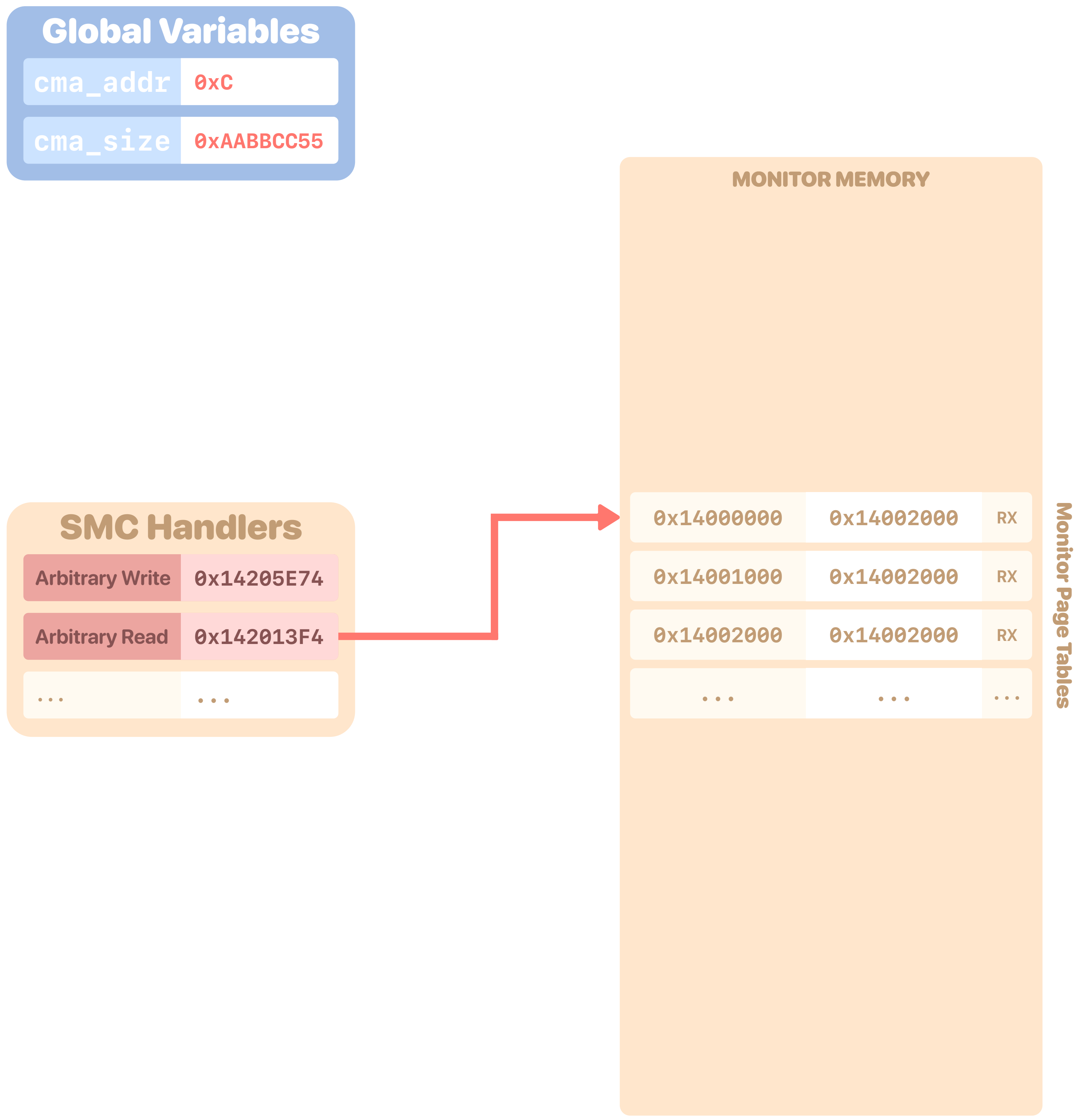
- **Step 1:** Use the response **metadata** to disable the check on the shared memory region
  - Allows copying the response **data** at an arbitrary *user_data.addr*
  - Data isn't controlled either, but gives us more options
- **Step 2:** Hijack a SMC handler pointer
  - 1-byte overwrite by specifying a **response size** of 1
  - Change an existing function pointer to an interesting gadget
    - BLR X2 —> arbitrary function call
- **Step 3:** Call a write gadget to create stable read and write primitives
- **Step 4:** Double map the Secure Monitor because of WXN
  - Locate the secure monitor page tables
  - Add new entries where the memory is read-write
  - Patch the code to gain code execution



Secure Monitor

Exploitation

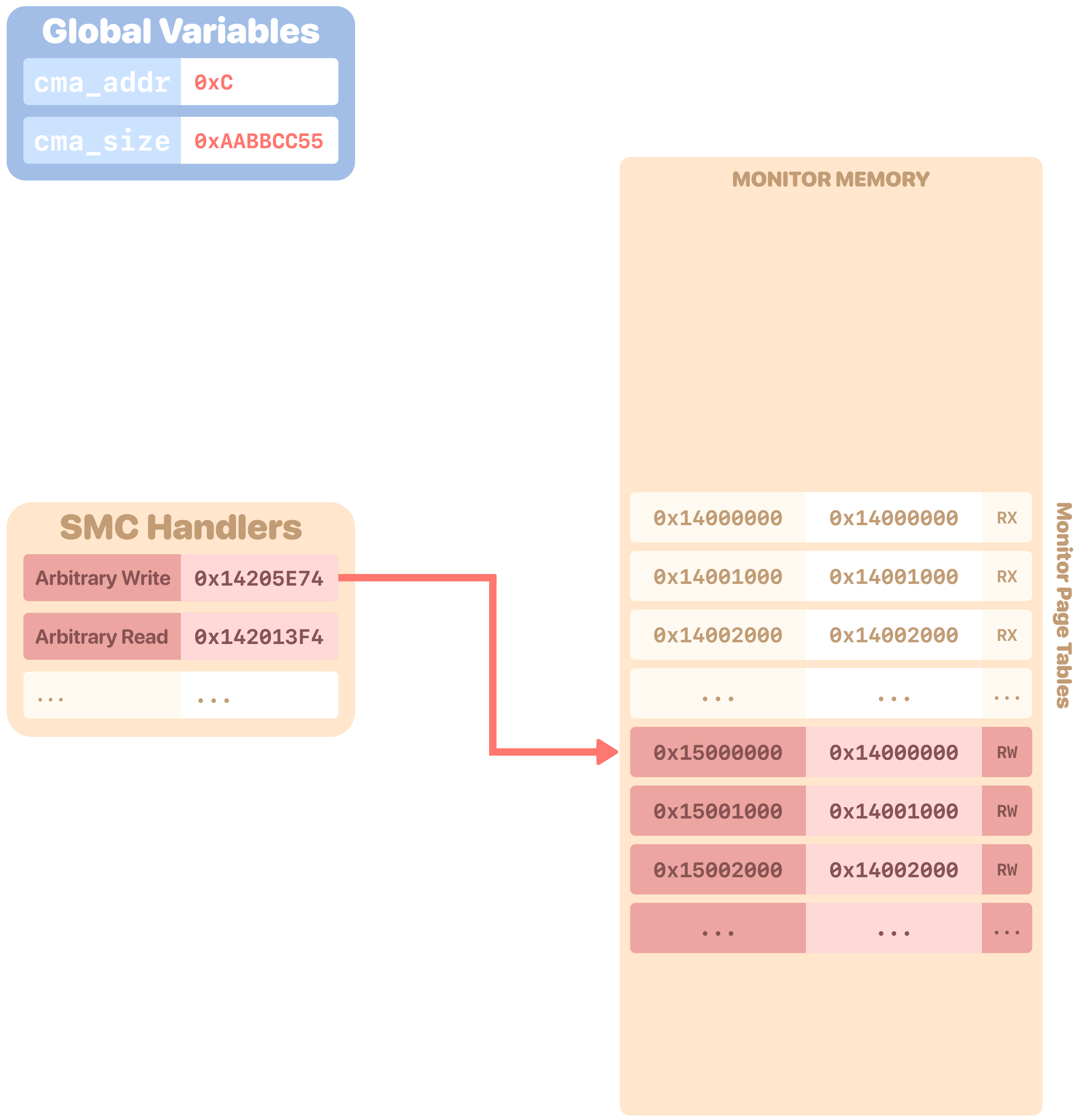
- **Step 1:** Use the response **metadata** to disable the check on the shared memory region
  - Allows copying the response **data** at an arbitrary *user_data.addr*
  - Data isn't controlled either, but gives us more options
- **Step 2:** Hijack a SMC handler pointer
  - 1-byte overwrite by specifying a **response size** of 1
  - Change an existing function pointer to an interesting gadget
    - BLR X2 —> arbitrary function call
- **Step 3:** Call a write gadget to create stable read and write primitives
- **Step 4:** Double map the Secure Monitor because of WXN
  - Locate the secure monitor page tables
  - Add new entries where the memory is read-write
  - Patch the code to gain code execution



Secure Monitor

Exploitation

- **Step 1:** Use the response **metadata** to disable the check on the shared memory region
  - Allows copying the response **data** at an arbitrary *user_data.addr*
  - Data isn't controlled either, but gives us more options
- **Step 2:** Hijack a SMC handler pointer
  - 1-byte overwrite by specifying a **response size** of 1
  - Change an existing function pointer to an interesting gadget
    - BLR X2 —> arbitrary function call
- **Step 3:** Call a write gadget to create stable read and write primitives
- **Step 4:** Double map the Secure Monitor because of WXN
  - Locate the secure monitor page tables
  - Add new entries where the memory is read-write
  - Patch the code to gain code execution

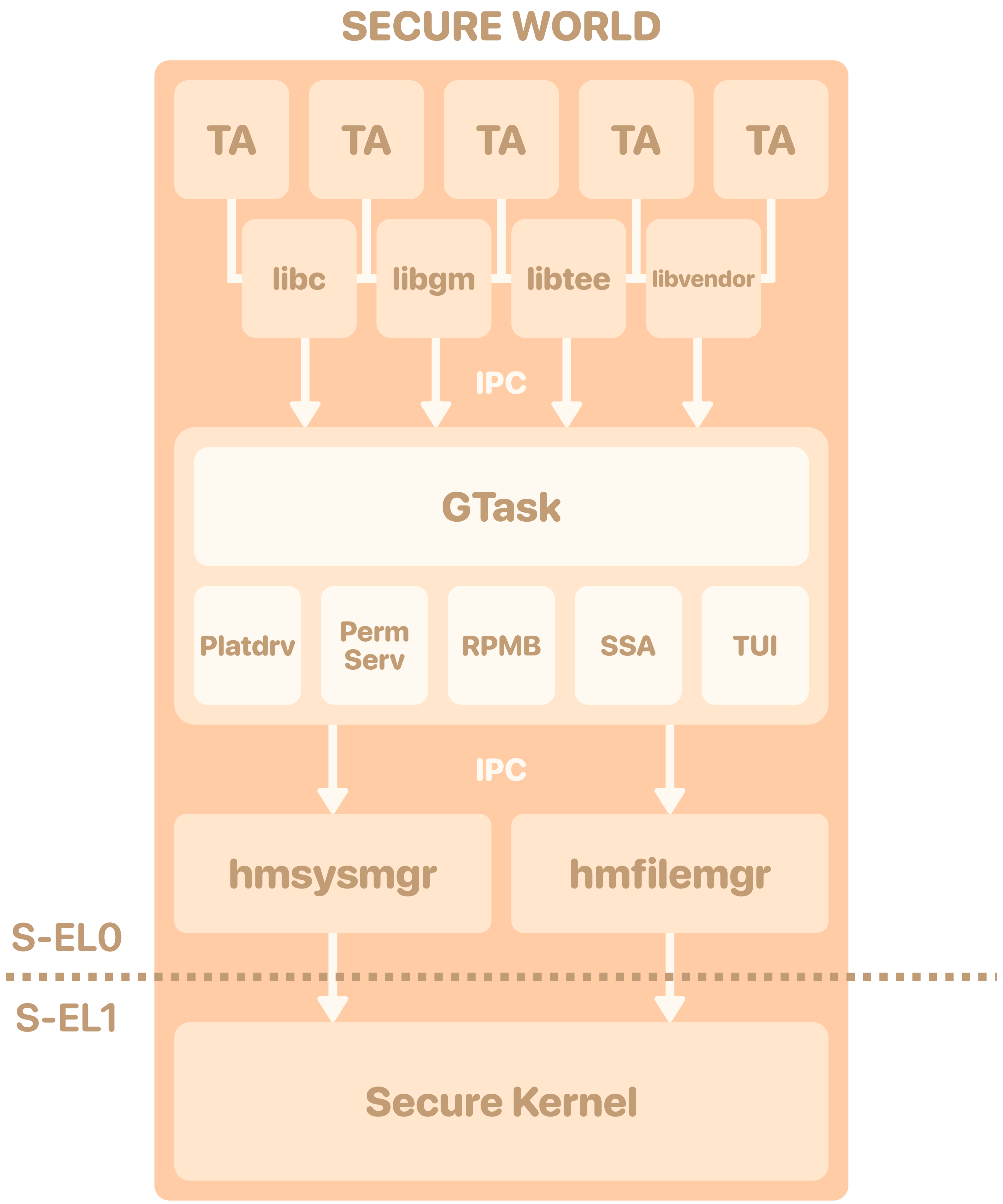


Trusted OS

Trusted OS

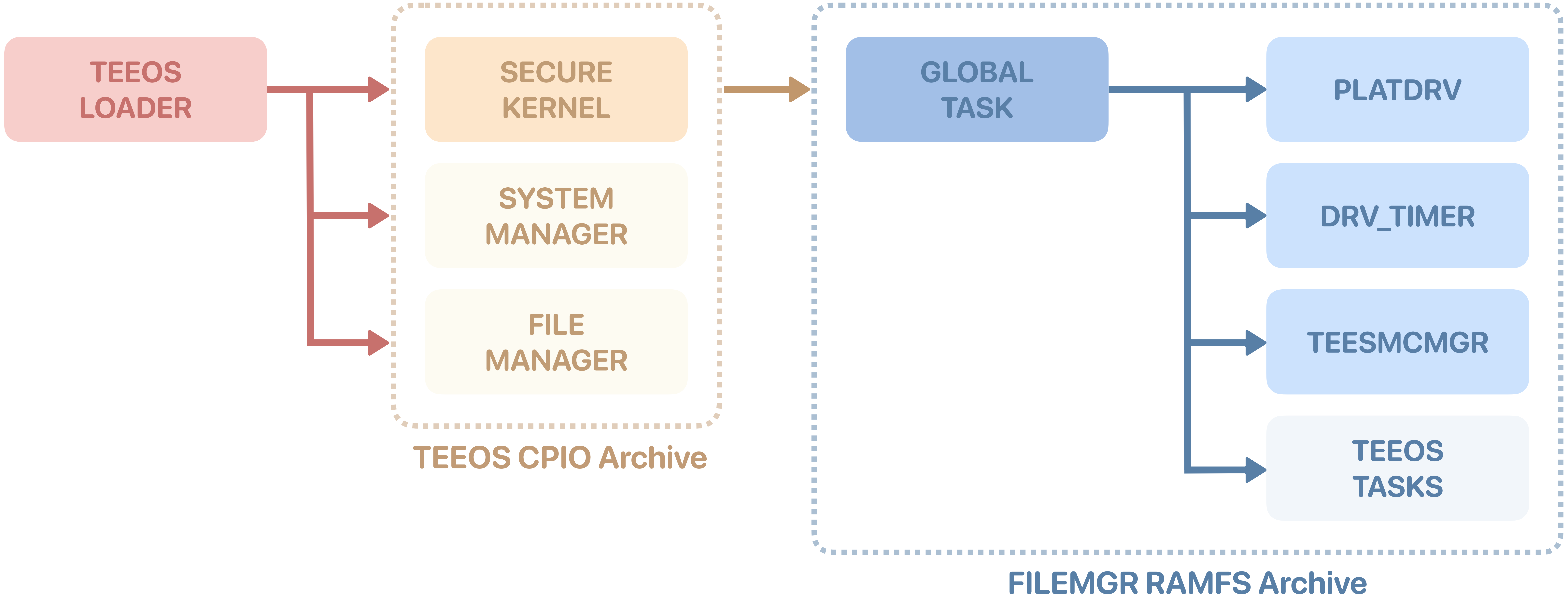
Introduction

- Huawei Trusted OS based on a **micro-kernel architecture**
  - **Secure Kernel** (S-EL1)
    - Responsibilities kept to the bare minimum
    - Critical operations are performed through an API restricted to Managers in userland
  - **Processes** (S-EL0)
    - **Managers:** privileged processes providing the core functionality of the trusted OS
    - **Tasks & Drivers:** implement additional OS services used by the trusted applications
    - **Trusted Applications:** Huawei and 3rd party applications providing services to the REE



Trusted OS

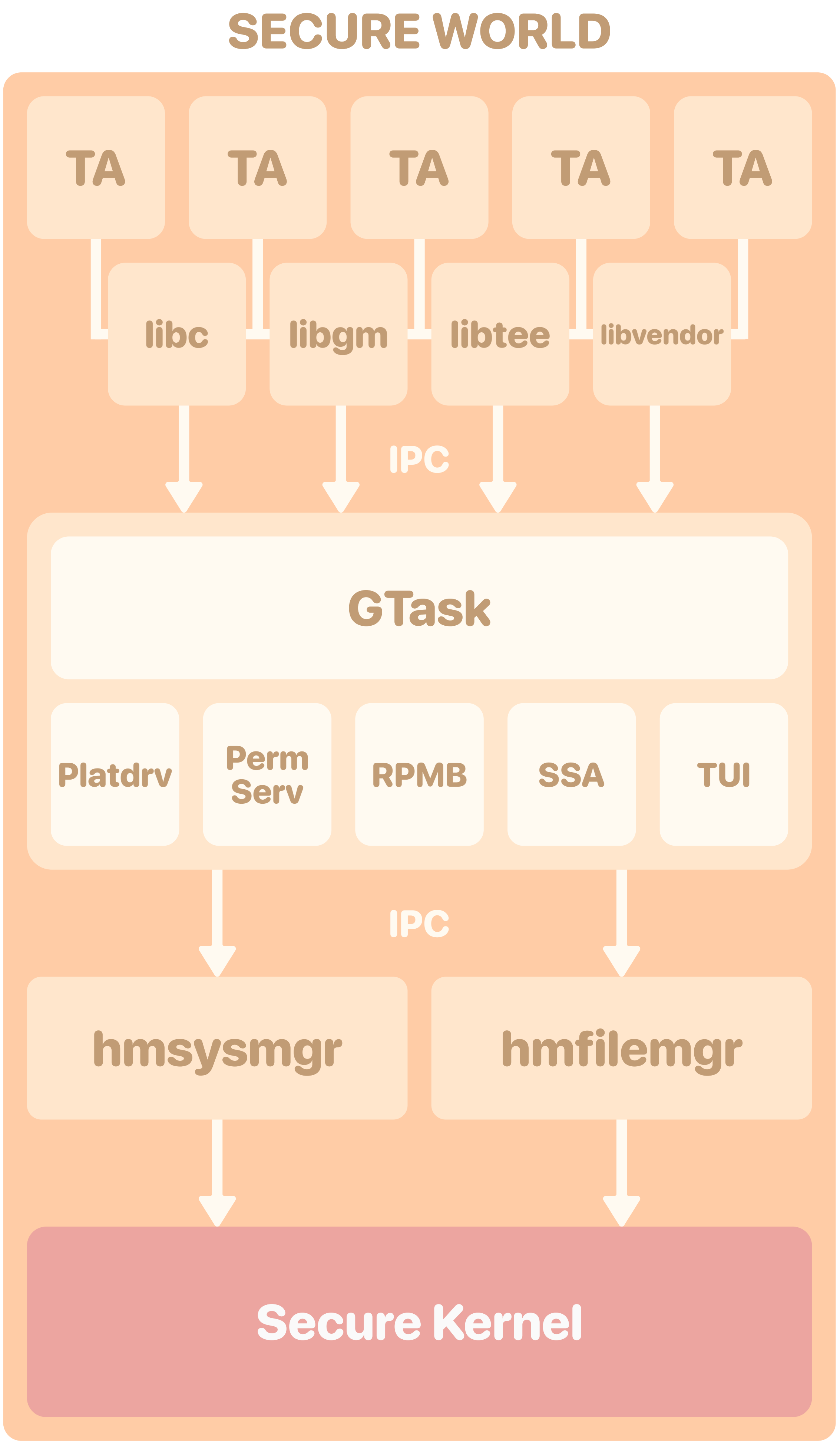
Boot Process



Secure Kernel

Introduction

- Only performs **low-level operations**, such as:
  - Physical memory allocation
  - Inter-process communication
  - Process scheduling
  - Access control management
- Everything else is implemented in **userland**
- SVCs for **critical operations** restricted to the Managers



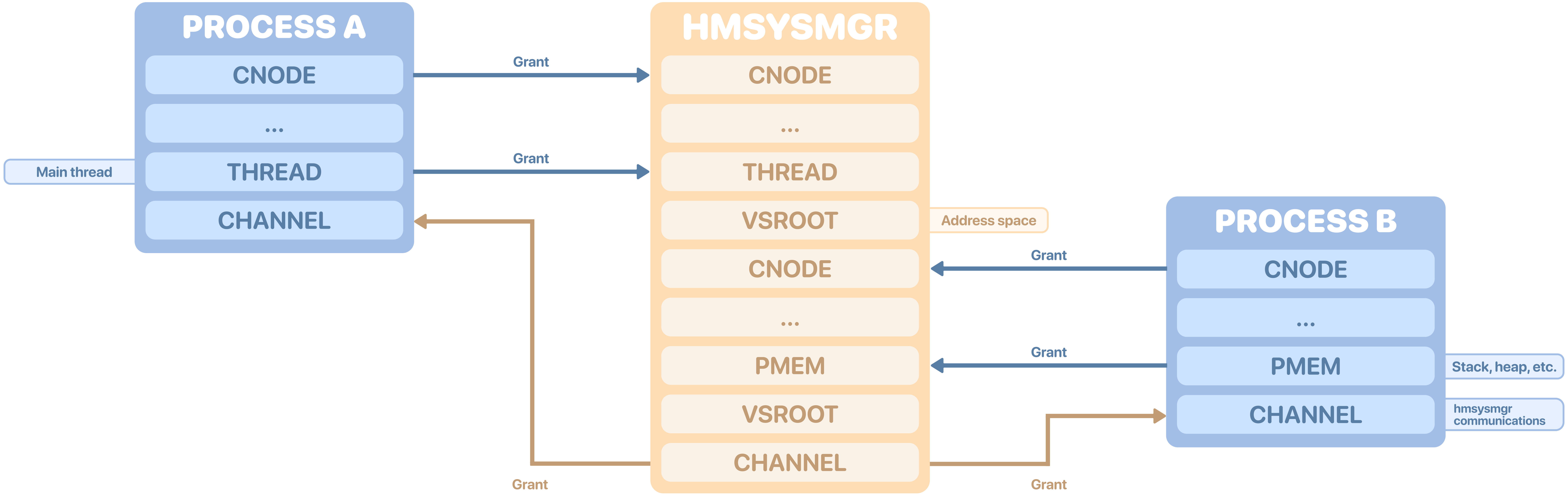
Secure Kernel

Capabilities

- **Capability-based OS**
  - Privileges are divided into distinct units called **capabilities**
  - Provides fine-grained access to kernel resources
- **Huawei Implementation**
  - Most likely inspired by **seL4**
  - Capabilities system described in a **patent** filed in 2019
  - All system resources are associated with a capability
  - Capabilities are **owned** by a **CNode** (capability node)
  - Capabilities can be **granted** to and **revoked** from other CNodes
- **Capability type examples**
  - CNode
  - Thread
  - PMEM
  - Channel / Notification / Message
  - IRQCTRL / IRQHDLR
  - VSRoot
  - Timer
  - TEESMC
  - etc.

Secure Kernel

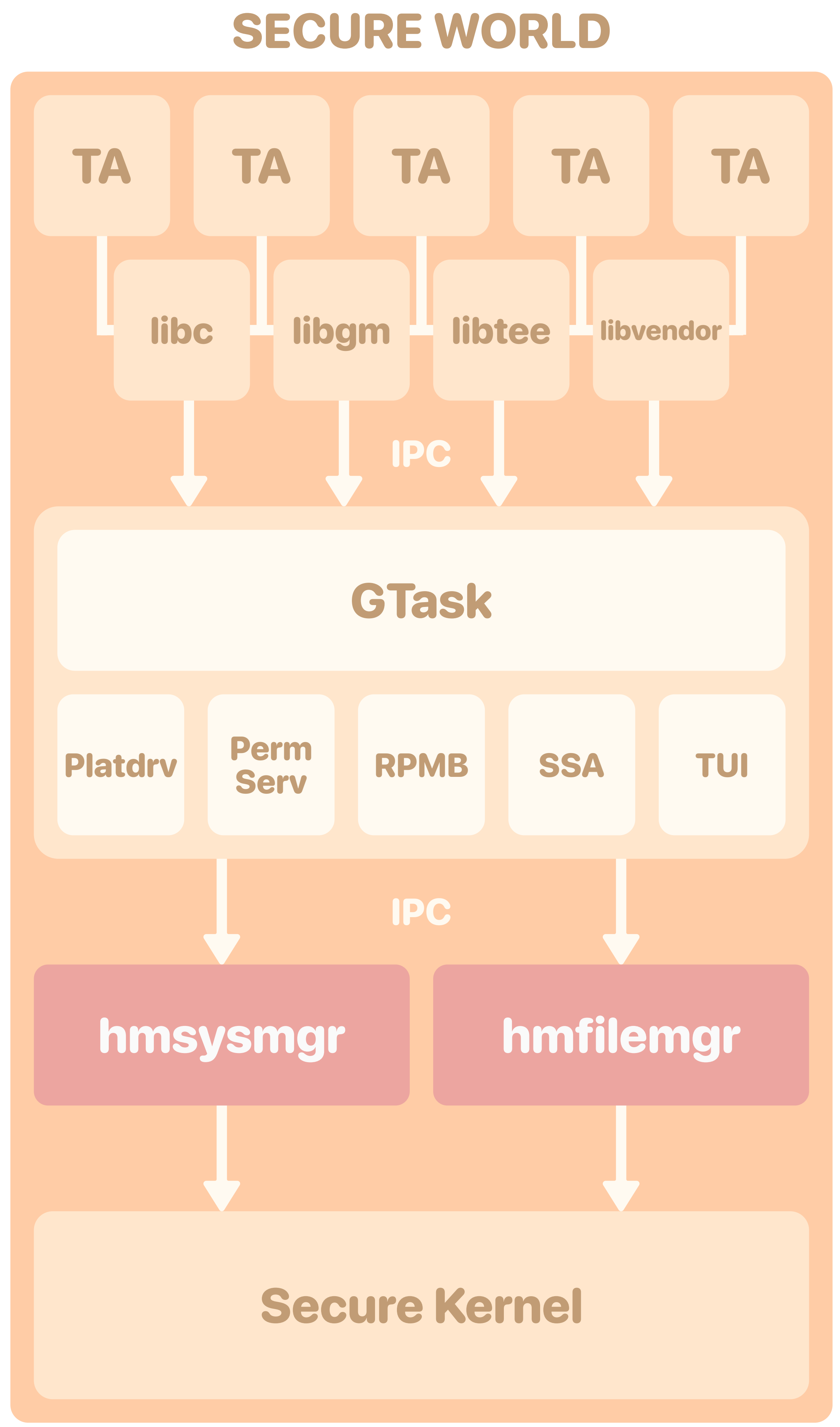
Capabilities Example



Managers

Overview

- **Managers**
  - The only S-EL0 processes allowed to ask the secure kernel to perform critical operations
    - e.g. mapping physical secure memory
  - Can be considered as **extensions** of the micro-kernel in **userland**



Managers

File & System Managers

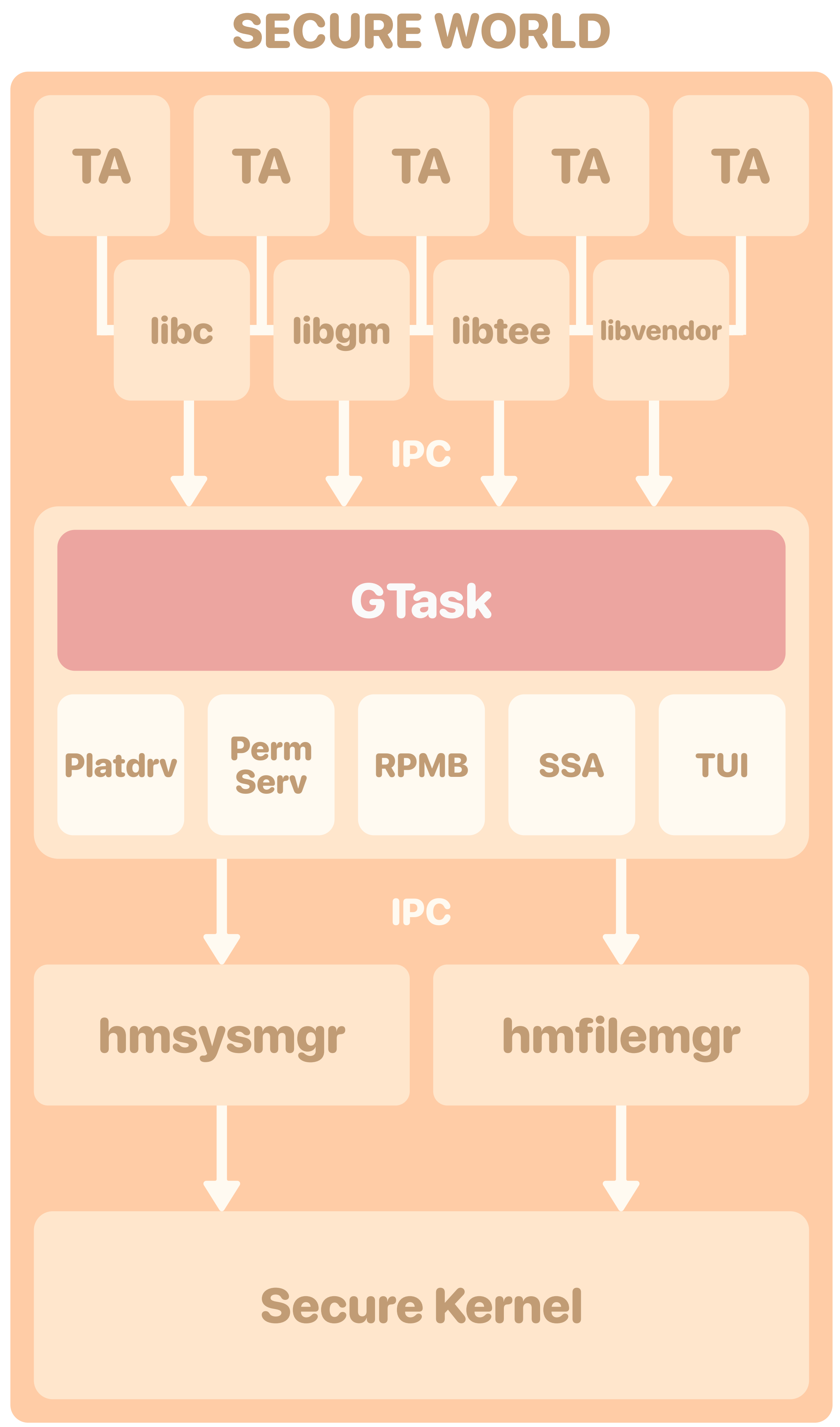
- **File manager** (hmfilemgr)
  - Manages and exposes two virtual file systems
    - **RAMFS**
      - Embedded archive
      - Contains tasks binaries
    - **TAFS**
      - Temporary storage for trustlets and libraries

- **System manager** (hmsysmgr)
  - Implements most of the **fundamental features** of the OS
    - Process creation
    - Virtual memory management
    - Access control
    - etc.
- Communicate with other processes through **IPCs**
- Permissions of the calling process are checked in the command handlers

Tasks & Drivers

Global Task

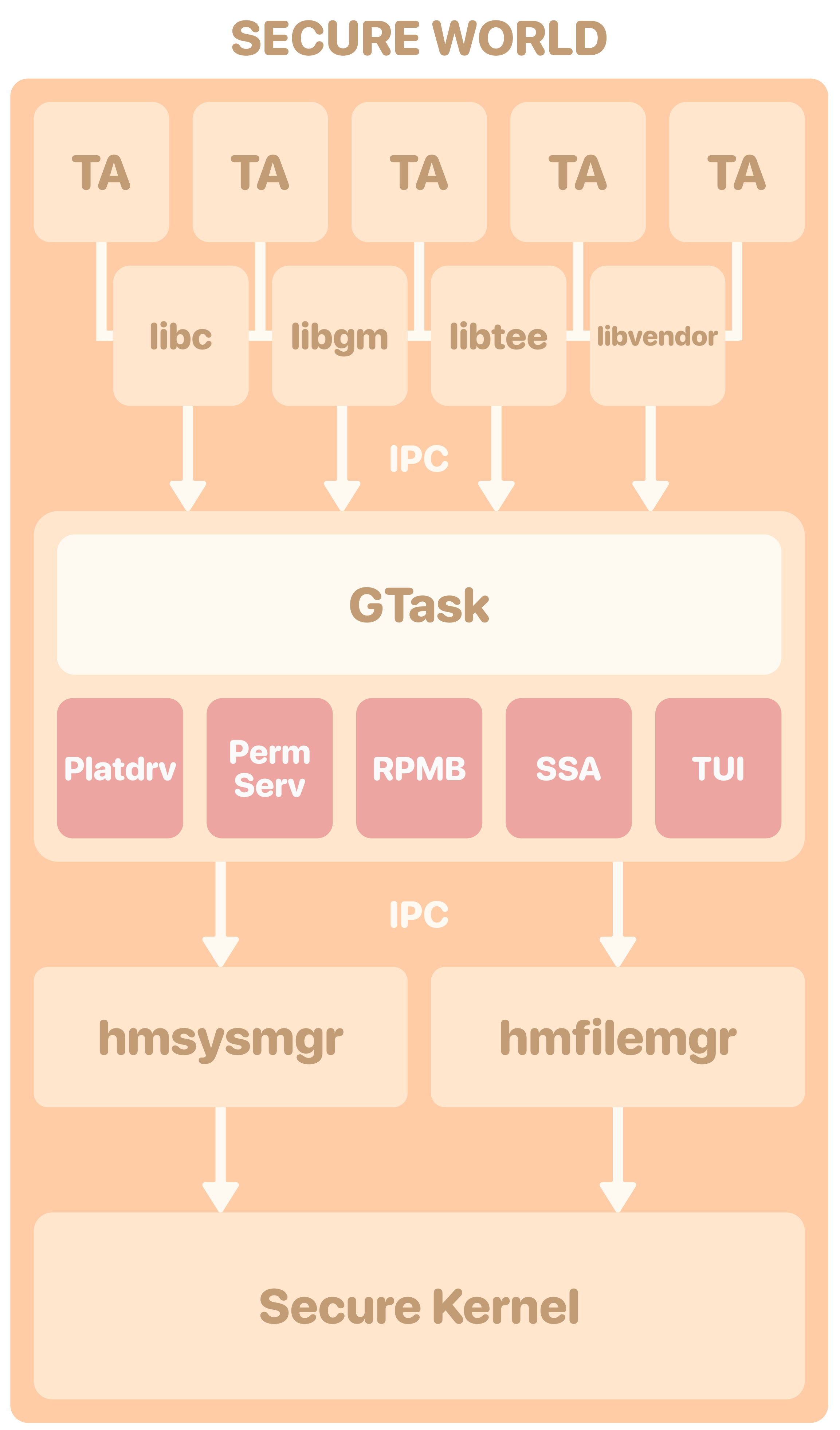
- Equivalent to the **init** process on Unix-based systems
- **Handle normal world commands**
  - Mailbox/shared memory registration
  - Loading of trusted applications
    - Decryption with a private key "derived" from the **provisioned key**
    - Signature verification with a **hardcoded public key**
  - Session management
  - Forwarding of commands to trusted applications



Tasks & Drivers

Examples of Tasks & Drivers

- **DRV_TIMER**
  - Manages secure timers
- **GATEKEEPER**
  - Gatekeeper implementation
- **KEYMASTER**
  - Keymaster implementation
- **PERMISSION_SERVICE**
  - Permissions system for RPMB, SSA and TUI
- **PLATDRV**
  - Platform drivers
  - Interrupts, crypto engine, secure element, fingerprint sensor, etc.
- **RPMB**
  - RPMB filesystem
  - Uses a normal world agent
- **SSA**
  - Trusted Storage API
  - Uses a normal world agent
- **TALOADER & TARUNNER**
  - glue between GlobalPlatform and OS-level APIs
- **TUI**
  - Trusted User Interface implementation



Tasks & Drivers

Security

- **Vulnerability research**
  - IPC command handlers
  - **Permissions system**
    - There is a library for implementing security access controls
    - Tasks have **credentials** and **security contexts**, that can be mapped to permissions
    - Most permissions are static, but can also be added **dynamically**
    - Permissions are checked within the IPC command handlers

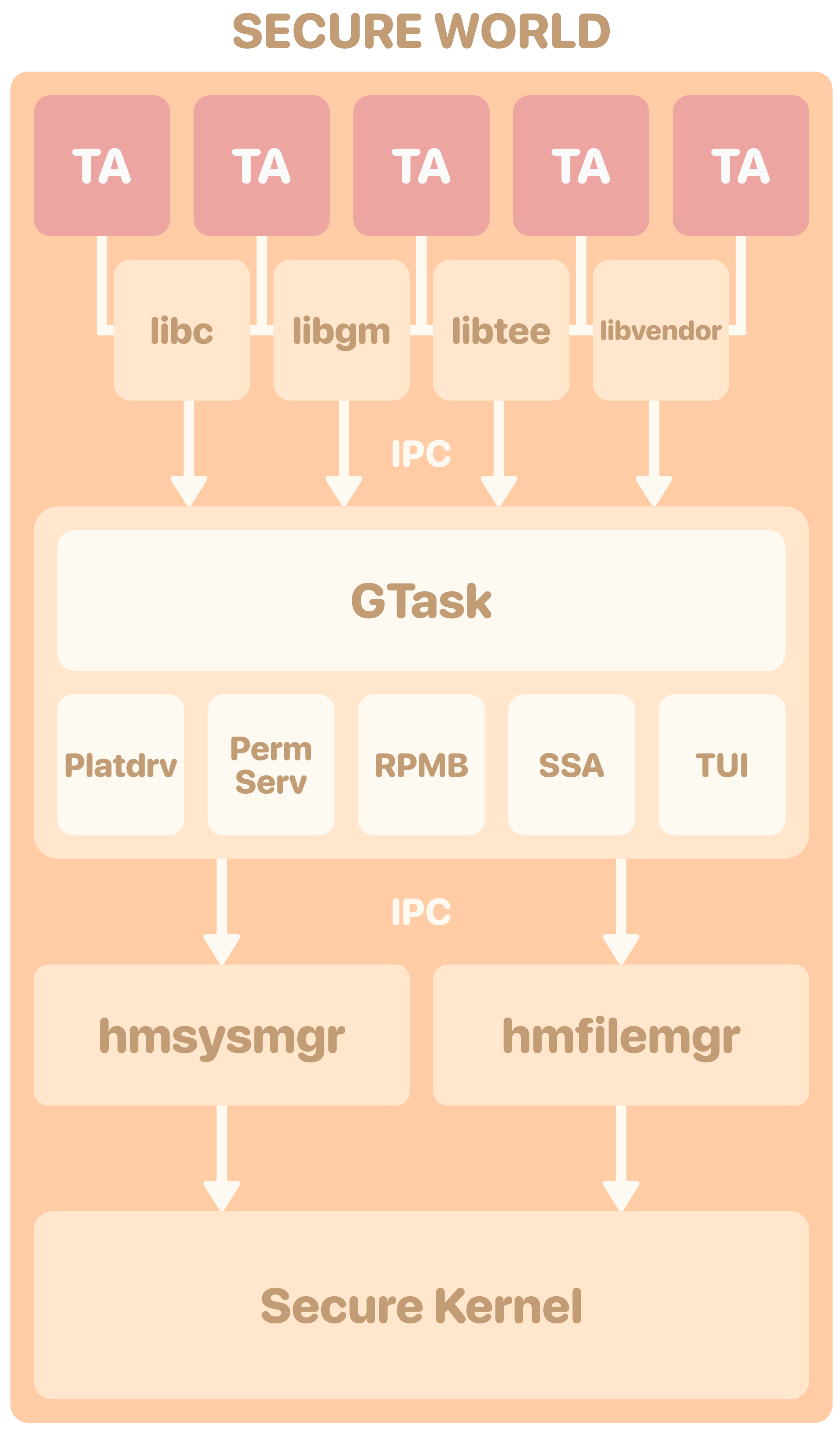
- **Vulnerabilities identified**
  - **TUI Task**
    - Heap buffer overflows
  - **Platdrv Task**
    - Arbitrary memory read/write
    - Non-secure physical memory read
    - Heap buffer overflows
    - Heap pointer leak
  - Only specific tasks can reach the vulnerable IPC command handlers

Trusted Applications

Trusted Applications

Introduction

- Secure world **userland applications**
- Developed by Huawei and 3rd parties to **provide services** to the Normal World
- Use the standard **GlobalPlatform APIs**, as well as some proprietary extensions
- Generally loaded from the Normal World
  - Stored in the Android system/vendor partitions or embedded in APKs
  - Signed and encrypted



Trusted Applications

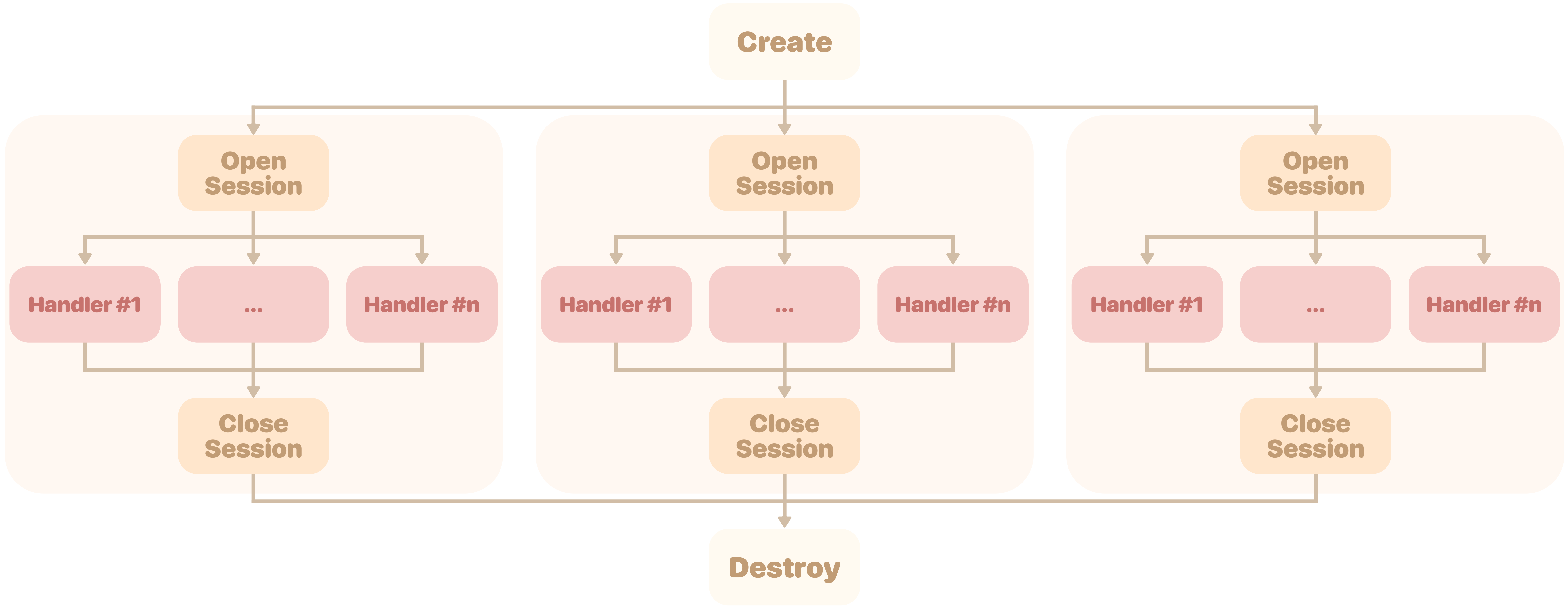
Life Cycle

- **Trusted Applications Properties**
  - Single instance, multi session, instance keep alive, etc.

- **Create and Destroy**
  - Manage the global state
  - Declare the **allowed CAs** list

- **Open and Close Sessions**
  - Manage the per-CA state

- **Command Invocation**
  - Handles a request coming from a CA and sends back a response



Trusted Applications

Authentication

- Trusted applications embed a list of authorized APKs/binaries
  - **APK**: package name + signing public key
  - **Binaries**: file path + user id + hash of code pages
- **Chain of trust**
  - The kernel is assumed to be uncompromised
  - The kernel authenticates teecd
  - teecd forwards information about the binaries

Trusted Applications

Design Choices & Mitigations

- **Design choices**
  - Secure functions (e.g. *memcpy_s*)
  - Parameter buffers are copied to prevent inter-world TOCTOU
  - Robust and generic Parcel-based system to handle data in a safe manner
  - Output buffer sizes can only be reduced
  - Etc.

- **Software Mitigations**
  - NX
  - RelRO
  - Stack cookies
  - ASLR
    - Used to be **bypassable** with an arbitrary read
    - The TA base address was written at a fixed address by the loader
    - Only works for the ELF sections, **stack** and **heap** are still randomized

Trusted Applications

Methodology

- **Reverse engineering**: ~40 trustlets, mainly AArch32 ELF but some AArch64
- The **attack surface** mostly boils down to the command handlers
- **Fuzzing**: developed a custom fuzzer based on *Unicorn/AFL++*
  - **Obstacles:** stubbing the GP APIs, ELF relocations, getting a backtrace
  - **Limitations:** stateless, only *low hanging fruits* can be found

- **Vulnerabilities**
  - Unchecked parameter types
  - Stack & heap buffer overflows
  - Information leaks
  - OOB accesses
  - Race conditions (multi session binaries only)
  - Etc.
- Mostly in **third party** TAs

Trusted Applications

Vulnerabilities in HW_KEYMASTER

- **HWPSIRT-2021-63568**
  - *cmd_unwrap* can be used to write arbitrary data to any files in the *sec_storage_data/PKI/* folder of the secure file system
- **HWPSIRT-2021-80349**
  - *generate_keyblob* copies semi user-controlled data into the output parameter *params[3]*
  - Should be a *memref*, but there is a code path where it can be a value

```c
typedef union {
    struct {
        void* buffer;
        size_t size;
    } memref;
    struct {
        uint32_t a;
        uint32_t b;
    } value;
} TEE_Param;

TEE_Result TA_InvokeCommandEntryPoint(
    void* sessionContext,
    uint32_t commandID,
    uint32_t paramTypes,
    TEE_Param params[4]
);
```

Trusted Applications

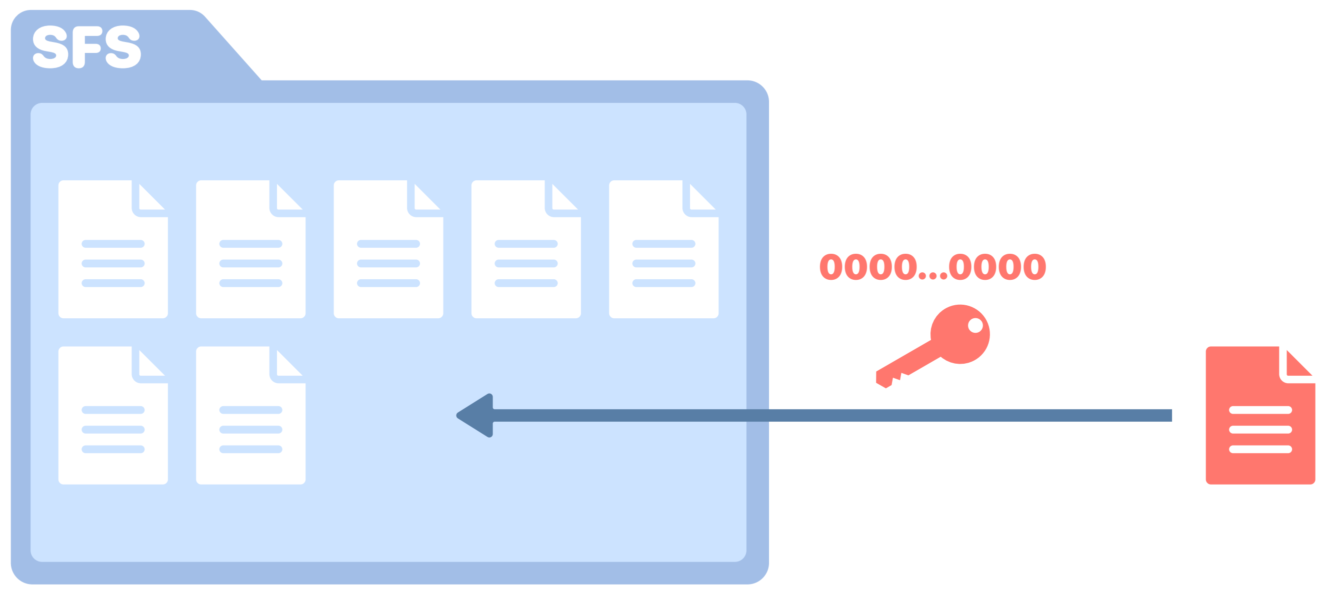
Exploitation of HW_KEYMASTER

- **Arbitrary read**
  - Write a "fake" keyblob to the SFS using a previously imported **all-zeroes** AES key

Trusted Applications

Exploitation of HW_KEYMASTER

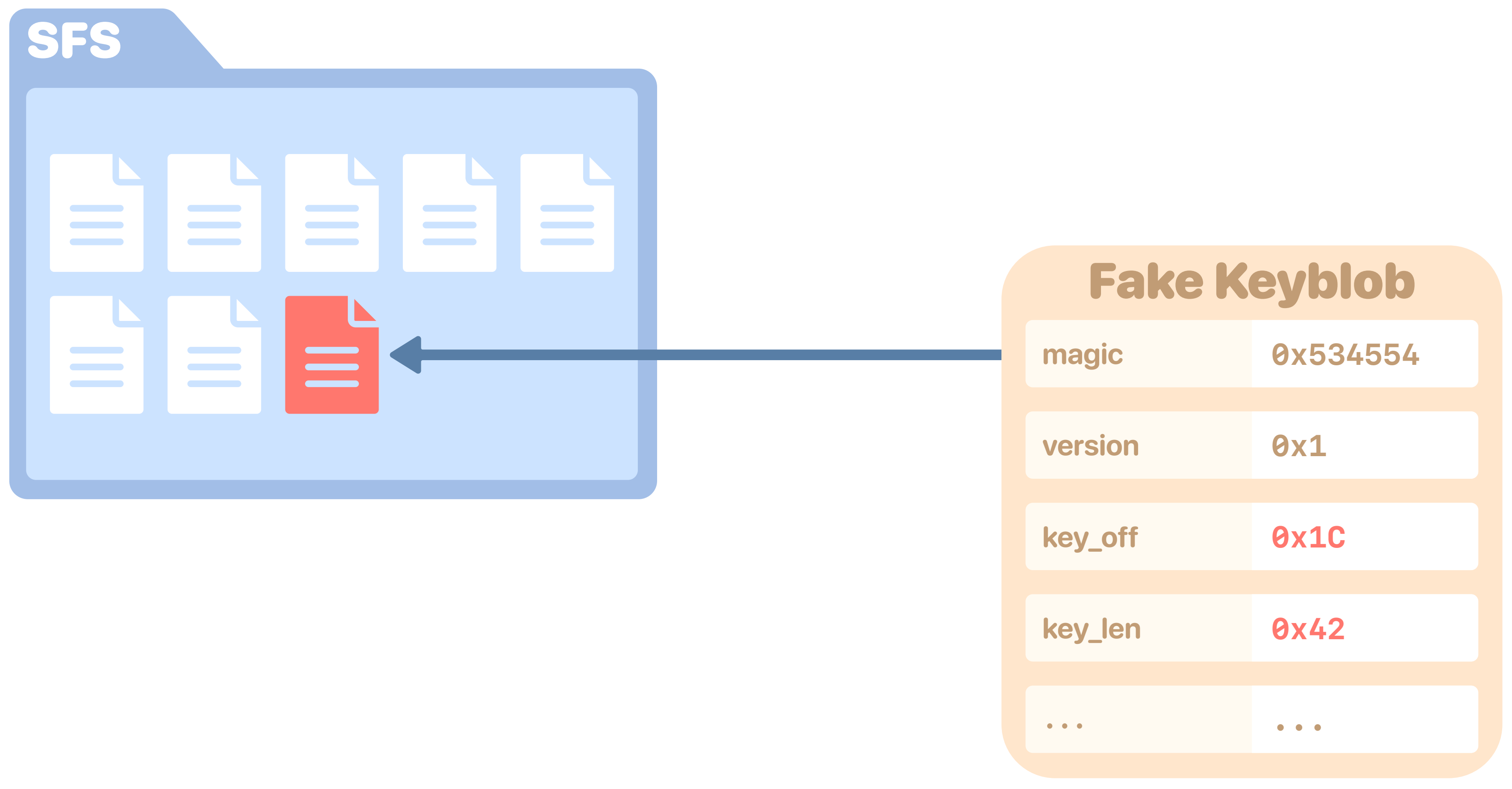
- **Arbitrary read**
  - Write a "fake" keyblob to the SFS using a previously imported **all-zeroes** AES key



Trusted Applications

Exploitation of HW_KEYMASTER

- **Arbitrary read**
  - Write a "fake" keyblob to the SFS using a previously imported **all-zeroes** AES key
  - Call *cmd_get* on the "fake" keyblob to read data from a user-controlled offset



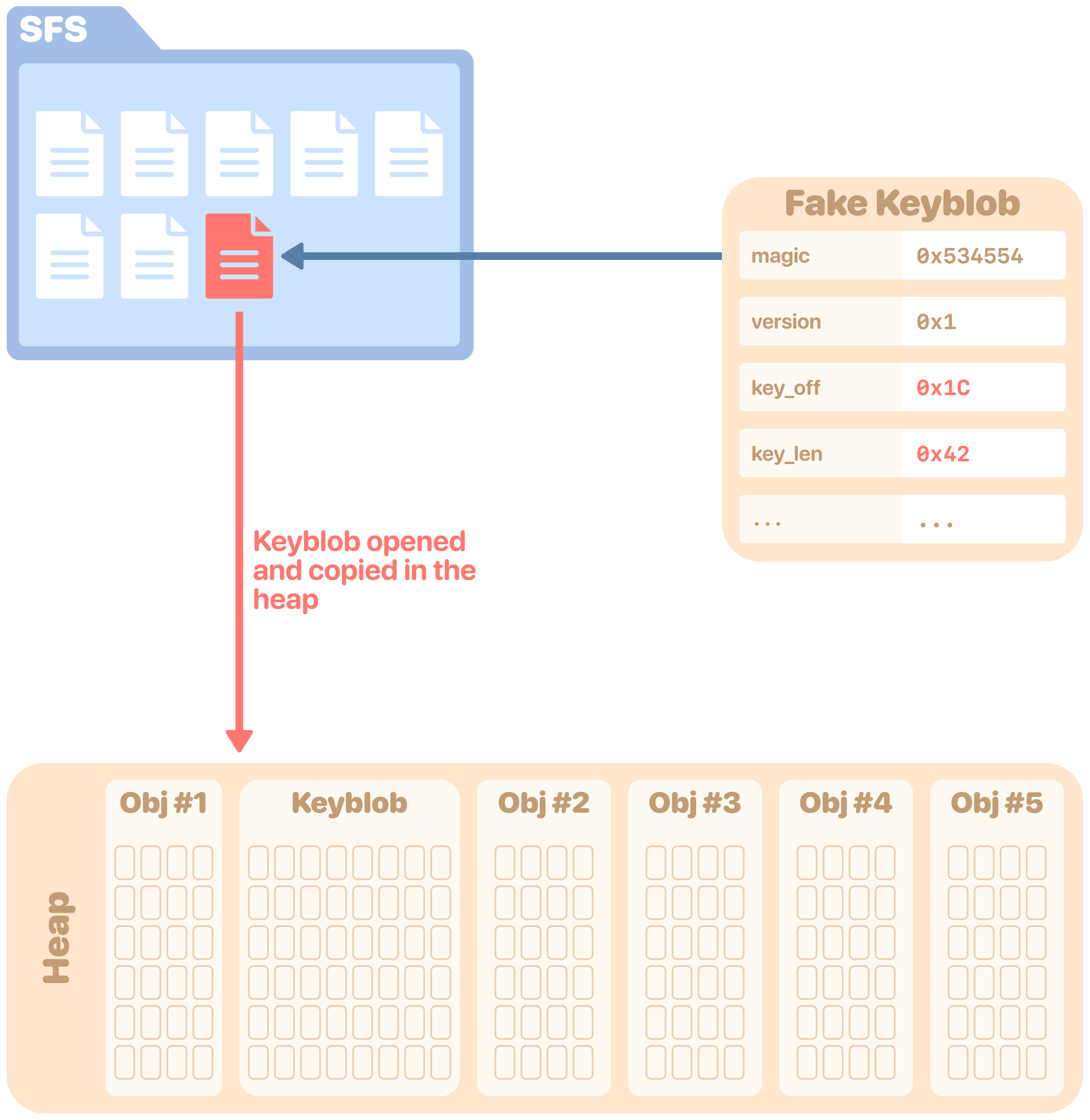

```
if (keyblob->magic == 0x534554
        && keyblob->version <= 0x12C
        && keyblob->keyblob_size == keyblob_size) {
    memcpy_s(
        params[1].memref.buffer,
        params[1].memref.size,
        keyblob + keyblob->key_off,
        keyblob->key_len);
}
```

Trusted Applications

Exploitation of HW_KEYMASTER

- **Arbitrary read**
  - Write a "fake" keyblob to the SFS using a previously imported **all-zeroes** AES key
  - Call *cmd_get* on the "fake" keyblob to read data from a user-controlled offset

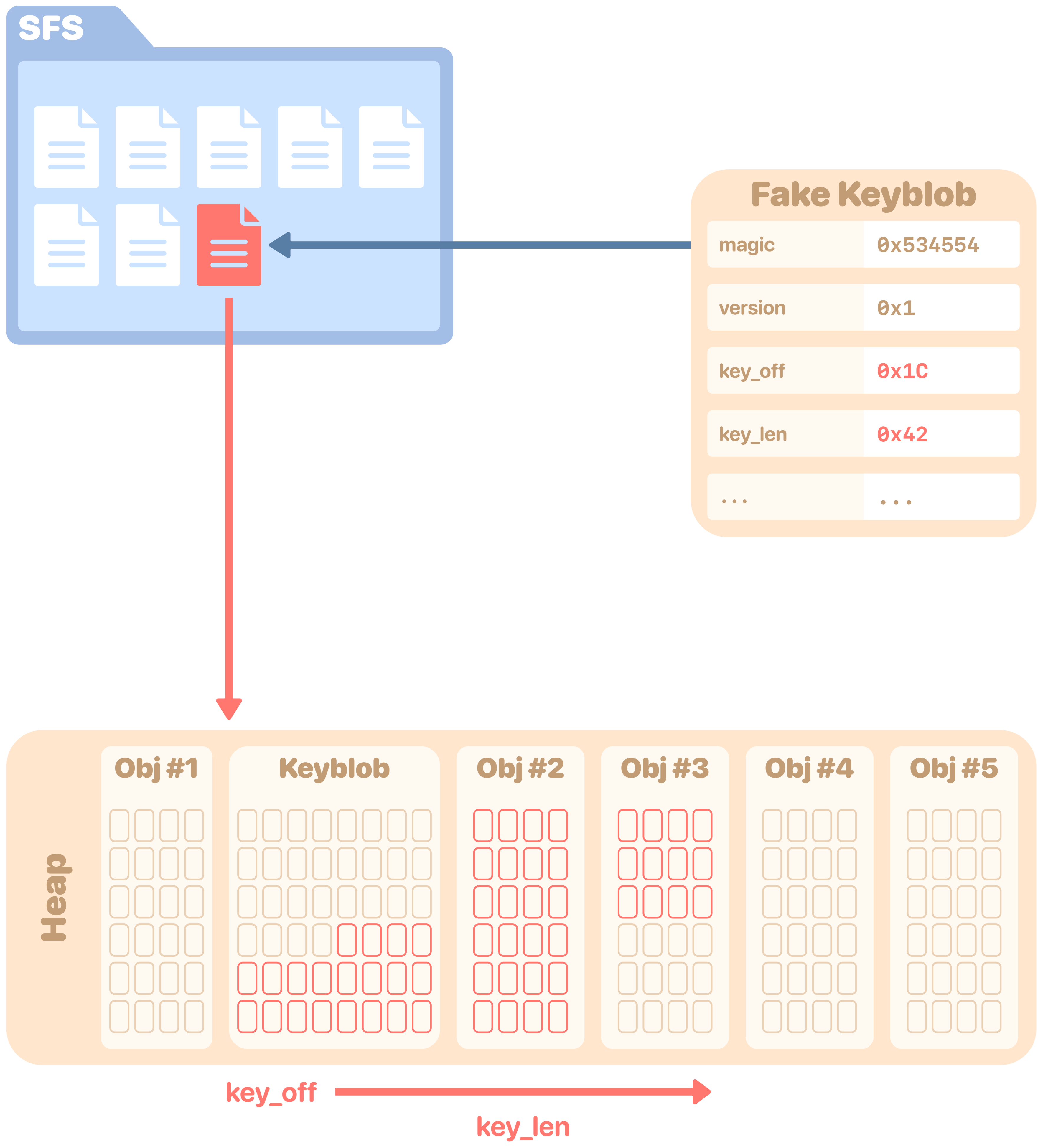
```
if (keyblob->magic == 0x534554
        && keyblob->version <= 0x12C
        && keyblob->keyblob_size == keyblob_size) {
    memcpy_s(
        params[1].memref.buffer,
        params[1].memref.size,
        keyblob + keyblob->key_off,
        keyblob->key_len);
}
```



Trusted Applications

Exploitation of HW_KEYMASTER

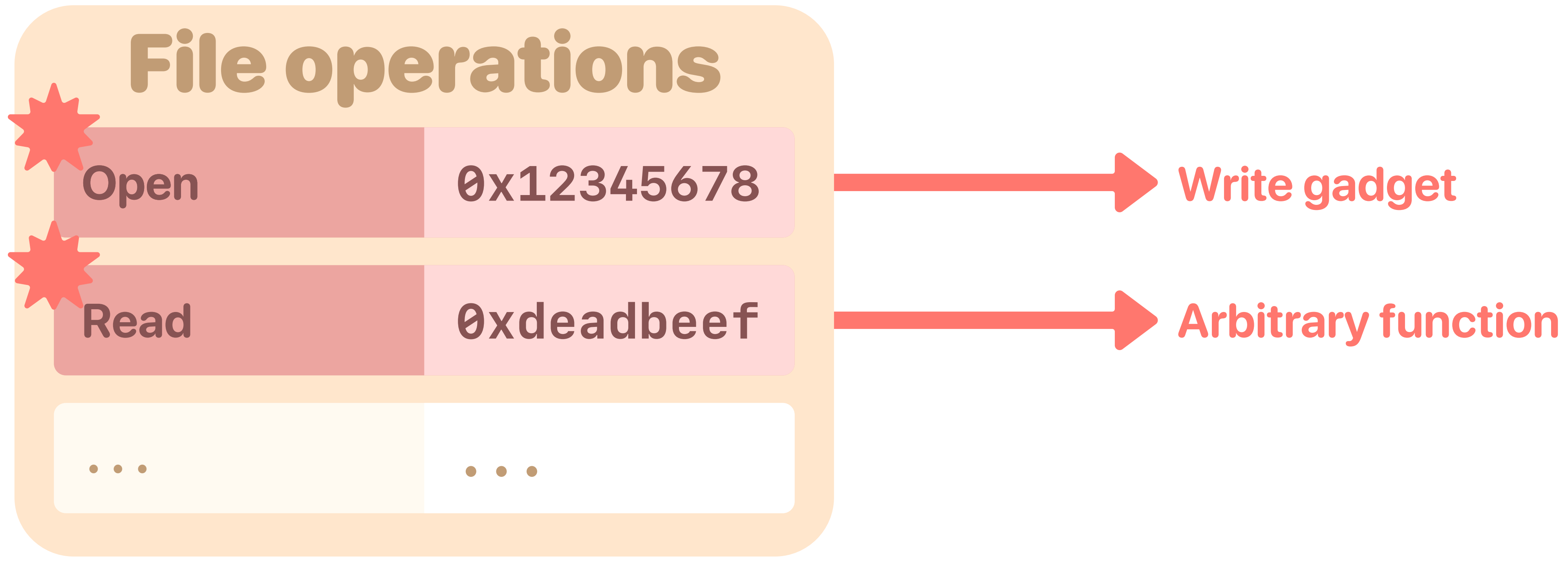
- **Arbitrary read**
  - Write a "fake" keyblob to the SFS using a previously imported **all-zeroes** AES key
  - Call *cmd_get* on the "fake" keyblob to read data from a user-controlled offset
    - First read adjacent heap data to get a leak of the **object's address**
    - Then you can read at arbitrary addresses, and break **ASLR** in particular



Trusted Applications

Exploitation of HW_KEYMASTER

- **Arbitrary read**
  - Write a "fake" keyblob to the SFS using a previously imported **all-zeroes** AES key
  - Call *cmd_get* on the "fake" keyblob to read data from a user-controlled offset
    - First read adjacent heap data to get a leak of the **object's address**
    - Then you can read at arbitrary addresses, and break **ASLR** in particular
- **Arbitrary write**
    - Use it to overwrite a function pointer (e.g. file operations structure) to create a better arbitrary write primitive
    - Can also use it to call arbitrary functions



Conclusion

Conclusion

- All vulnerabilities were reported to *Huawei Bug Bounty Program* and **fixed** in updates released prior to this presentation
- **Well thought-out** security architecture
  - Defense-in-depth measures
  - Privilege limitations and access control
  - Robust implementations (secure coding practices)
  - Mistakes can still happen, but are **mitigated**
- **Binary encryption** is a double edged-sword
  - Harder for an attacker to get access and find bugs
  - But teams with the resources to break the encryption layer might be less likely to share their findings
- **Upcoming blogposts** with the missing details
  - https://blog.impalabs.com

Thank you!