DANIEL KOMAROMY, EKOPARTY 2017

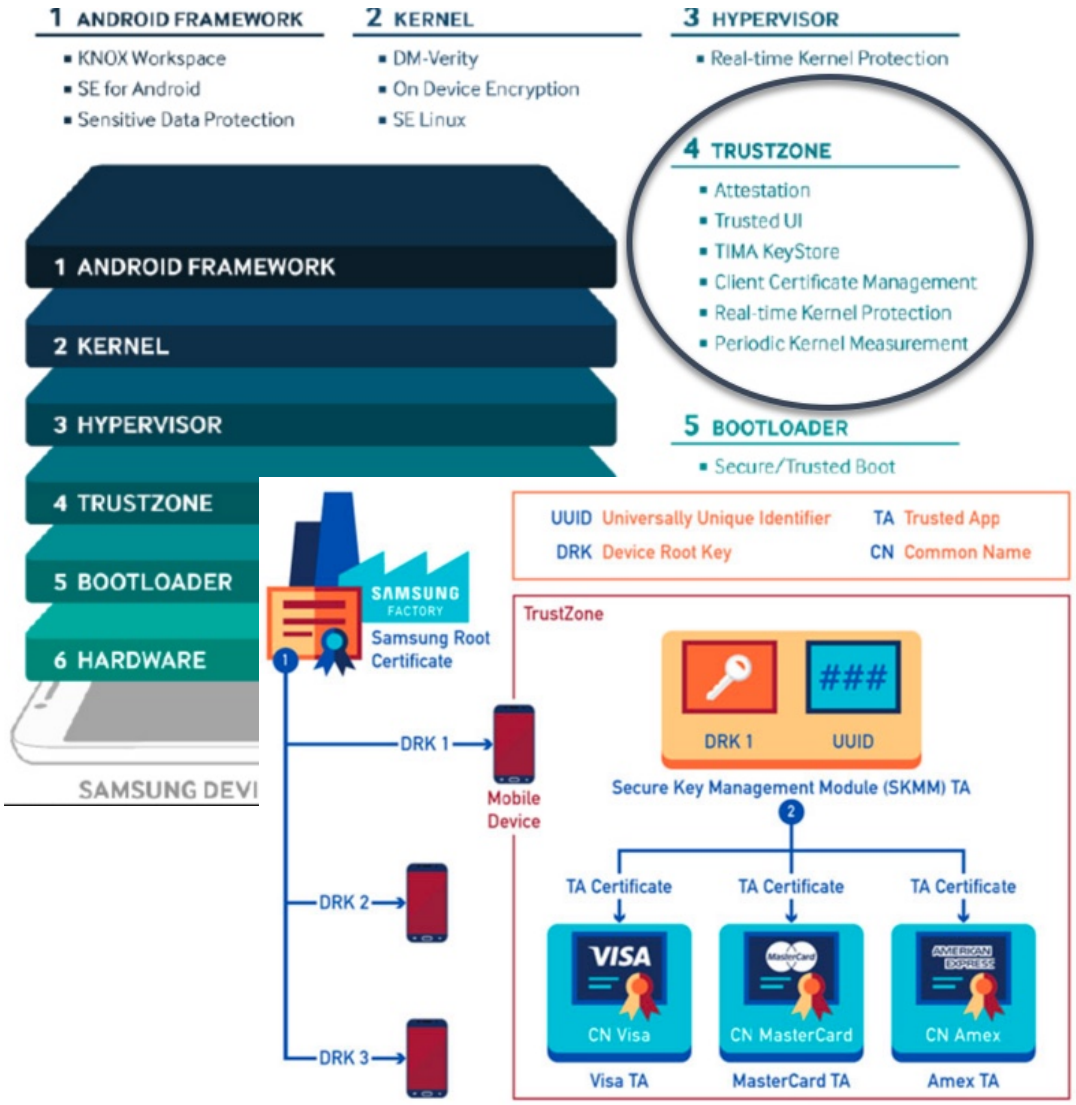
EXPLORING AND BREAKING SAMSUNG'S TRUSTZONE SANDBOXES

INTRO

WHY

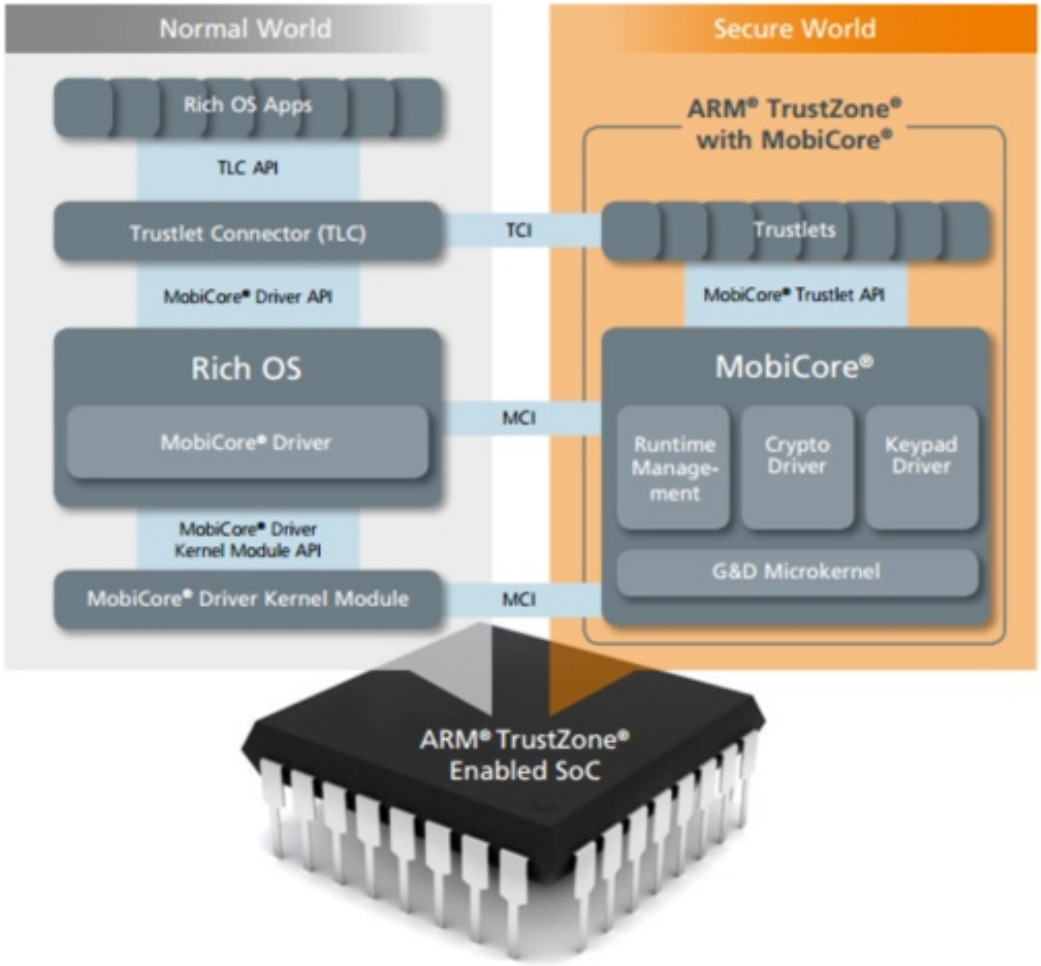
- TrustZone on other platforms
  - extensively researched
  - statically found vulnerabilities, straightforward exploitation due to terrible isolation properties
- TrustZone on Samsung
  - minimal research, until recent work on Trustlets only (P0)
  - OS complete black box
  - my main interest: understanding the system

WHAT WE KNOW ABOUT TRUSTZONE AND SAMSUNG KNOX



- Trusted Applications provide services: DRM, KeyStore, Certificate Management, SPay, Trusted UI, Trusted PIN, etc.
- Trusted Drivers control access to devices: SPI (Synaptic Fingerprint Sensor), eSE (NFC), MST (SPay), etc.
- Trusted Drivers add platform security to Android: Kernel Measurement, Device Encryption, etc.

WHAT WE KNOW ABOUT THE TRUSTONIC TEE



- microkernel-based OS
- SW but NOT EL3 (monitor mode)
- MobiCore Control Protocol for nSW to talk to SW (documented)
- Trustonic provides Android {kernel driver, Daemon, client library} that proxy MCP commands into Android-level mcXYZ APIs
- Trustlet loading format (MCLF), names of APIs (tlApiXYZ) provided by a common "libc" fairly well known / previously researched
- **Trustlet Connector Interface (TLC/TCI), aka the protocols for talking to Trustlets: common TCI header, otherwise entirely proprietary!**

WHAT MAKES TBASE A MICRO KERNEL?

- handful of SMC fastcalls only, two that matter:
  - INIT: Android tells Trustonic location of command queues
  - NOTIFY: tell Trustonic there is a new command in the queues
- everything else is via shared memory, handled by:
  - MCP (loading/configuring trustlets): ??? but the expectation is that it is not the kernel itself
  - Trusted Applications

WHAT WE DON'T KNOW 1.

- How is the t-base microkernel implemented?
  - Where can t-base firmware actually be found on Android?
  - How does it start up and implement common features (MMU, process management, etc)
  - What system calls does it implement?
  - How/where are SMCs handled? Are the any extra SMC calls beyond what's documented?
  - How/where is the MCP implemented?

WHAT WE DON'T KNOW 2.

- Trustlets
  - How exactly are they loaded? E.g. how is the libc (mcLib) loaded?
  - What Trustlet uuid corresponds to what actual (KNOX) feature?
  - What are the differences between Trustlets and Drivers?
  - How are tlApi/drApi calls actually implemented, how do they map to interfaces towards the microkernel?
  - Security hardening and isolation features in SW?

WHAT WE DON'T KNOW 3.

- Android side
  - What happens beyond libMcClient.so?
  - What system components *actually* use TrustZone?
  - What debugging features are present?
  - What (if any) attack surface is there for an unprivileged user?

REVERSING T-BASE

FIRMWARE EXTRACTION

- t-base header embedded inside sboot, identified by “t-base”
- gives addresses and sizes of several firmware components:
  - microkernel
  - mcLib library
  - core trustlets (e.g. crypto driver)
  - the special “dom0” trustlet (MCCB/RTM/S0CB … whatever :)

FIRMWARE EXTRACTION

```
                DCB 0, 0
tbase_extract_table DCB "t-base ",0     ; descriptor struct:
                                        ;     char name[8]
                                        ;     int offset
                                        ;     int size
                                        ;     char padding[0x10]
                                        ;
                                        ; real start offset: 0x132000
                                        ;
                                        ; Mtk: 0->0x147000 -> so that's the image itself -> so go back 0x15000 from "t-base"
                                        ; Image_h: 0x147000 -> 0x148000 -> so that's this
                                        ; Rtm: 0x148000 -> RTM is SOCB (IPC/MCP implementing "dom0" root trustlet)
                                        ;
                                        ; drcrypt: 0x167000 -> MCLF header there
                                        ; tlproxy: 0x17A000 -> MCLF header there
                                        ; sth2: 0x17B000    -> MCLF header there
                                        ; mclib: 0x183000   -> tlLib indeed
                DCD 0
                DCD 0x5D000
                ALIGN 0x20
aMtk            DCB "mtk    ",0
                DCD 0
                DCD 0x15000
                ALIGN 0x20
aImage_h        DCB "image_h",0
                DCD 0x15000
                DCD unk_1000
                ALIGN 0x20
aRtm            DCB "rtm    ",0
                DCD 0x16000
                DCD 0x1F000
                ALIGN 0x20
aDrcrypt        DCB "drcrypt",0
                DCD 0x35000
                DCD 0x13000
                ALIGN 0x20
aTlproxy        DCB "tlproxy",0
                DCD 0x48000
                DCD 0x1000
                DCD 0
                DCD 0
                DCD 0
                DCD 0
aSth2           DCB "sth2   ",0
                DCD 0x49000
                DCD unk_8000
                ALIGN 0x20
aMclib          DCB "mclib  ",0
                DCD 0x51000
                DCD unk_C000
                ALIGN 0x1000
SOCB_start      DCB "SOCB",0
```

FIRMWARE ARCHITECTURE

- t-base is connected to nSW by ATF (ARM Trusted Firmware)
  - https://blog.quarkslab.com/reverse-engineering-samsung-s6-sboot-part-ii.html awesome explanation of this monitor mode, read that :)
- let's start from the microkernel initialization and identify: SMC handling (MC fastcall commands), MCP handling, SVC handling
- hopefully we'll understand what role the S0CB has

T-BASE MICROKERNEL START UP

- loading to ida: start address 0x7F00000
  - can be figured out based on embedded pointer values, but also boot logs that can be googled. Prior work also mentioned this by now (Gal Beniamini)
- typical helpers: MCR instructions (setting VBAR, enabling MMU, etc)
- but in any case, it was still pretty painful. lot of things unclear at first that I was able to come back to when I understood more about higher layers

T-BASE MICROKERNEL START UP

- early steps: pretty typical stuff
  - configure VBAR, enable MMU, enable cache, reset various context's stacks (interrupt, smc, etc)
- then starts the t-base specific part
  - tell ATF where the VBAR is for handling SMC calls
  - map the mcLib and the S0CB
  - initialize process structures and start S0CB

T-BASE MICROKERNEL START UP: SMC

- ATF needs to know the VBAR, because its SMC handler uses the offset from there to trap into t-base's smc/fastcall handler. From here we can find the SMC handlers, eg fastcalls:

```
tbase_smc_send_VBAR
LDR             R0, =0xB2000002
MOV             R1, #1
LDR             R2, =vector_table_normal
SMC             #0      ; SMC call tells ATF the VBAR address
BX              LR
; End of function tbase_smc_send_VBAR
```

```
void __fastcall __noreturn fastcall_handler(int a1, int a2, int a3, int a4, int a5)
{
  unsigned int main_id_reg; // r2@1

  main_id_reg = __mrc(15, 0, 0, 0, 5);
  __mcr(15, 0, vector_table_int_cxt, 12, 0, 0); // set the VBAR for fastcall handling
  if ( cpu_revision_number == 4 * ((main_id_reg >> 8) & 0xF) + (main_id_reg & 0xF) )
    process_fastcall(&a5, a2);                  // used here
  else
    handle_fastcall_2(&a5, a2);
  __mcr(15, 0, vector_table_normal, 12, 0, 0);  // restore normal VBAR
  __asm { SMC             #0 }                  // back to ATF
  __isb();
  while ( 1 )
    ;
}
```

```
int __fastcall process_fastcall(int a1, int a2)
{
  int v2; // r4@1
  signed int *fc_args; // r0@1
  unsigned int fastcall_id_; // r1@1

  v2 = a2;
  fc_args = get_fastcall_args_addr(a1, a2);     // offset + SP (in R0)
  fastcall_id_ = *fc_args & 0xFF000000;
  if ( fastcall_id_ == 0x84000000 )
    return process_psci_fastcalls(fc_args, v2);
  if ( fastcall_id_ == 0xFF000000 )
    return process_mc_fastcalls(fc_args);
  return process_custom_fastcalls(fc_args, v2);
}
```

T-BASE MICROKERNEL START UP: MAP LIB, SOCB

- to find where processes are mapped into memory, I had to find the functions used for managing page table entries
- identifying page table operations helped by:
  - MMU enablement code fixes TTBR0 to a static address, references easy to see
  - uses normal AARCH64 two level page table design, so code that e.g. creates a pte has a pretty easily recognizable look
  - intuition: it's readily apparent from the SMC handlers (fastcall and normal) that there isn't any implementation of MCP in here, so there HAD to be a mapping and creation of a new process in the init sequence (this turns out to be the S0CB).

T-BASE MICROKERNEL START UP: REVERSING HINTS

- S0CB header format known, data flow from accesses to it in code can be followed
- matching syscalls made in similar operations in different places, e.g. syscall made by drApiMapClientBuffer to syscall made by mcMap MCP call, or functions used by syscall made to start process by mcOpenSession to functions used by microkernel startup sequence to start S0CB
- helpful: 0 heap in microkernel, all descriptors are at fixed locations, references trivial to see
- unlike in the microkernel, Samsung trustlets and drivers have a TON of debug strings; this helped a lot making sense of meaning of libc calls and from there figuring out what syscall implements what

T-BASE MICROKERNEL START UP: MAP LIB, SOCB

```
int __fastcall store_pte_0(int virt_addr, int virt_addr_high, int phys_addr_low, int phys_addr_high, int permissions)
{
  int virt_addr_; // r4@1
  int pt_entry; // r0@1

  virt_addr_ = virt_addr;
  pt_entry = create_pte(phys_addr_low, phys_addr_high, permissions, 0);
  return store_VA_to_pt(0, virt_addr_, pt_entry);
}

unsigned int __fastcall create_pte(int pa_low, int pa_hi, unsigned int perms, int dirtyflag)
{
  unsigned int pte_; // r0@1
  unsigned int v5; // r4@3
  unsigned int v6; // r0@4

  pte_ = pa_low & 0xFFFFF000 | 0x403;
  if ( perms & 0x80 )
    pte_ |= 0x20u;
  v5 = (perms >> 1) & 1;
  if ( dirtyflag )
  {
    v6 = pte_ | 0x800;
    if ( v5 )
      pte_ = v6 | 0x40;
    else
      pte_ = v6 | 0xC0;
  }
  else if ( !v5 )
  {
    pte_ |= 0x80u;
  }
  if ( perms & 0x100 )
    return pte_;
  if ( perms & 0x10 )
  {
    pte_ |= 4u;
  }
  else if ( perms & 8 )
  {
    pte_ |= 8u;
  }
  else
  {
    pte_ |= 0x30Cu;
  }
  return pte_;
}
```

```
void __fastcall __noreturn config_tbase_start_s0cb()
{
  process_struct_1 *new_proc; // r5@1
  unsigned int max_processes; // r1@1
  unsigned int v2; // r3@2
  int v3; // r0@4
  void *process_structs_area_addr; // [sp+0h] [bp-10h]@1
  int process_structs_area_length; // [sp+4h] [bp-Ch]@1

  memset_to_0_0(&tbase_conf_t, 60);
  tbase_conf_t.tbase_state = tbase_conf_t.tbase_state & 0xFFFFF8FF | 0x100;
  config_tbase_map_s0cb_to_0x7FFF0000_tell_atf_vbar(&process_structs_area_addr, &process_structs_area_length);
  sanity_check_S0CB_header();
  prepare_per_process_descriptors(
    process_structs_area_addr,
    process_structs_area_length,
    mapped_s0cb.max_processes,
    mapped_s0cb.max_sgs_allowed);
  dword_7F117B0 = -1;
  write_EL1_thread_id();
  reset_counter_something();
  tbase_conf_t.tbase_state = tbase_conf_t.tbase_state & 0xFFFFF8FF | 0x200;
  new_proc = init_new_process(1, 0, 0xFFFFF0F, (LOBYTE(mapped_s0cb.max_sgs_allowed) << 24) | 0xFFFFFF);
  max_processes = mapped_s0cb.max_processes;
  while ( max_processes > 1 )
  {
    v2 = max_processes--;
    new_proc->perm_flags[v2 >> 5] |= 1 << (v2 & 0x1F);
  }
  v3 = mmap_s0cb_for_itself(new_proc);
  if ( create_new_process(new_proc, mapped_s0cb.entry, v3) )
    j_exit_to_ATF(15, 0x7F00976);
  while ( 1 )                                   // actually ERET into S0CB
  {
    jump_to_R0(go_to_ATF_or_sW_EL0, 1, 0, 0);
    __mcr(15, 0, EL1_thread_struct, 13, 0, 4);
  }
}
```

T-BASE SYSCALLS

```
void __fastcall svc_D_mprotect(int r0_0)
{
  int v1; // r4@1
  int v2; // r11@1
  process_struct_1 *proc_t_map_into; // r4@3
  unsigned int arg1; // r9@4
  unsigned int arg2; // r7@4
  int arg3; // r10@4
  int arg4; // r8@4
  int v8; // r6@4
  process_struct_1 *proc_t_map_from; // r5@12
  int v10; // r0@18
  unsigned int id; // [sp+0h] [bp-10h]@1
  void *a1; // [sp+8h] [bp-8h]@1

  v1 = r0_0;
  a1 = r0_0;
  v2 = get_arg0(r0_0);
  id = get_arg0(a1) >> 16;
  if ( proc_id_check_1(id) && !proc_id_is_0(id) )
    proc_t_map_into = get_process_struct_by_id(id);
  else
    proc_t_map_into = *(v1 + 364);
  arg1 = get_arg1(a1);
  arg2 = get_arg2(a1);
  arg3 = get_arg3(a1);
  arg4 = get_arg4(a1);
  v8 = 1;
  if ( arg4 << 29
    && proc_id_check_1(v2)
    && (arg1 < arg2 || !arg2)
    && is_low_12_bits_0_address_alignment_check(arg1 | arg2 | arg3) )
  {
    if ( v2 )
    {
      proc_t_map_from = get_process_struct_by_id(v2);
      if ( (!proc_id_check_1(id) || proc_id_is_0(id)) && proc_t_map_from != proc_t_map_into )
      {
        v8 = verify_process_permission(proc_t_map_from, v2, proc_t_map_into);
        if ( v8 )
          goto LABEL_19;
      }
    }
    else
    {
      proc_t_map_from = proc_t_map_into;
    }
    execute_mprotect(proc_t_map_from, proc_t_map_into, arg1, arg2, arg3, arg4);//
                                                //
                                                // from the process struct, get the first level page table,
                                                // from there get the phys address of corresponding second
                                                // level page table and map that into a fix address for the kernel.
                                                //
                                                // then, now sitting in kernel virtual address, directly modify
                                                // that page table entry to actually create the mapping.
                                                //
                                                //
    v8 = v10;
```

```
syscall_table   DCD svc_0_nop+1          ; DATA XREF: invoke_syscall_from_table+40↑o
                                         ; invoke_syscall_from_table:syscall_table_ptr↑o
                DCD svc_1_init_process+1
                DCD svc_2_nop+1
                DCD svc_3_nop+1
                DCD svc_4+1              ; Did not find this invoked anywhere in {SOCB,tlLib}
                DCD svc_5_start_process+1
                DCD svc_exit+1
                DCD svc_mmap+1
                DCD svc_8_munmap+1
                DCD svc_9_start_thread+1
                DCD svc_A_stop_thread+1
                DCD svc_B_return_0xD+1
                DCD svc_C_modify_thread_registers+1
                DCD svc_D_mprotect+1
                DCD svc_E_resume_thread+1
                DCD svc_F+1
                DCD svc_10_set_thread_prio+1
                DCD svc_11_ipc+1
                DCD svc_12_int_attach+1
                DCD svc_13_int_detach+1
                DCD svc_14_sigwait+1
                DCD svc_15_signal+1
                DCD svc_16+1             ; Did not find this invoked anywhere in {SOCB,tlLib}
                DCD svc_tbase_smc_fastcall_input+1
                DCD svc_18_log_char+1
                DCD svc_19_get_secure_timestamp+1
                DCD svc_1A_control+1     ; includes a lot, such as:
                                         ; - driver shmem map/unmap
                                         ; - get/set exception info
                                         ; - get MCP queue info
                                         ; - get IPCH phys address values
                                         ; - cache control
                                         ; - virt2phys, phys2virt translation
                                         ; - set custom fastcall, call custom fastcall
```

REVERSE ENGINEERING S0CB

- S0CB is a trustlet also, with (almost) same MCLF loading format
- Start-up sequence is pretty complex, as it has to initialize various descriptors for maintaining state information of trustlet instances and it also has to load initial drivers (specifically, the crypto driver)
- The most important is starting three threads:
  - MCP handling
  - IPC handling
  - Trustlet notification handling
- To be able to handle things, S0CB needs to access the MCP and notification queues. It uses the control syscall (0x1A) to get the address information from the microkernel and then maps it in for itself.

REVERSE ENGINEERING SOCB: MCP

- MCP: we know the command types and definitions
- Follow OpenSession command, see how Trustlets and Drivers are loaded (permissions, where shared memory buffers are put, etc.)

```
void __fastcall process_MCP(char *mcp_cmd, int a2)
{
  int v2; // r4@1
  _DWORD *v3; // r0@3
  __int64 v4; // kr00_8@6
  __int64 v5; // kr08_8@6
  int v6; // r1@8
  int v7; // r2@8
  int v8; // r3@8
  int v13; // [sp+0h] [bp-68h]@3
  __int64 v14; // [sp+8h] [bp-60h]@3
  int v15; // [sp+10h] [bp-58h]@3
  int v16; // [sp+14h] [bp-54h]@3

  v2 = a2;
  switch ( *mcp_cmd )
  {
    case 1:                                             //
                                                        // MC_MCP_CMD_OPEN_SESSION
      mcpOpenSession(mcp_cmd, a2);
      break;
    case 3:                                             //
                                                        // MC_MCP_CMD_CLOSE_SESSION
                                                        //
      goto LABEL_4;
    case 4:                                             //
                                                        //
                                                        // MC_MCP_CMD_MAP
                                                        //
                                                        //
      v4 = *(mcp_cmd + 4);
      v13 = *(mcp_cmd + 4) >> 32;
      v5 = *(mcp_cmd + 2);
      v14 = *(mcp_cmd + 2);
      v15 = *(mcp_cmd + 3);
      v16 = *(mcp_cmd + 6);
      *(a2 + 8) = mcpMap(v4, &v13);
      break;
    case 5:                                             //
                                                        // MC_MCP_CMD_UNMAP
                                                        //
                                                        //
      mcpUnmap(*(mcp_cmd + 1), *(mcp_cmd + 2), *(mcp_cmd + 3), *(mcp_cmd + 4));
      break;
    case 9:                                             // mcpGetMobicoreVersion
      memset_to_0(&v13, 96);
      if ( tlApiGetMobicoreVersion(&v13, v6, v7, v8) )
      {
        assert_after_setting_exception_info();
        __asm { SVC             6 }
        while ( 1 )
          ;
      }
      memcpy((v2 + 8), &v13, 0x60u);
      break;
    case 0xA:
      mcpReset();                                       // MC_MCP_CMD_CLOSE_MCP
                                                        //
                                                        //
      break;
    case 0xB:                                           //
                                                        //
                                                        // MC_MCP_CMD_LOAD_TOKEN
                                                        //
      v13 = *(mcp_cmd + 1);
      v14 = *(mcp_cmd + 1);
      v15 = *(mcp_cmd + 4);
      v16 = *(mcp_cmd + 6);
      mcpLoadToken(&v13);
```

REVERSE ENGINEERING SOCB: IPC

- IPC: entirely proprietary, no initial info.
- The big help was identifying the tlApi/drApi callers of SVC 0x11, which gave away which command type is what.
- From there, connect more syscall knowledge (e.g. mmap, mprotect etc) to commands
- It was an iterative process, constantly back and forth among microkernel, S0CB, and trustlets
- In total: of 40 IPC commands, I was able to identify the purpose of 30

REVERSE ENGINEERING SOCB: IPC

```
case 1u:                                // MSG_RQ -> Request from client to server (ta->driver)
  ret.field_0 = ipc_cmd;
  ret.msg_len = 0;
  goto PROCESS_REQ;
case 2u:                                // MSG_RS -> Response
  MCP_process_IPC_0x2_MSG_RESP(peerIDs.caller_ID, peerIDs.called_ID, ipc_cmd);
  break;
case 3u:                                // MSG_RD -> Driver Ready: first IPCH call by a driver always
  ret.field_0 = ipc_arg;                // basically the very first time we call from a driver,
                                        // we will get assigned a (temp) thread id, this is essentially
                                        // our instance id used in IPC; unliked the (static)
                                        // driver ID that is the same for each instances.
  ret.msg_len = peerIDs.caller_ID;
  *a2 = ipc_cmd;
  a4 = peerIDs.called_ID;
  mcp_DRV_set_id_2(peerIDs.caller_ID);
  TA_instance_set_id_sg_driverId(ret.msg_len, ret.msg_len);
  drv = find_MCP_DRV_struct_slot(ret.msg_len);
  if ( drv )
  {
    marshalled_cmd.ta_id = a4;          // here the ipc_cmd_parsed_struct_t marshalled_cmd
                                        // struct is actually var_84, so that's really confusing
    v30 = *a2;
    *&marshalled_cmd.handler_id = ret;
    call_handler_by_id(drv, &v30);      // we are going to call the handler 3 here
    call_MSG_RSP_handlers_of_all_TAs();
  }
  break;
case 4u:                                // MSG_NOTIFY
  v2 = find_trustlet_by_driverId(peerIDs.caller_ID);
  if ( v2 != -1 )
    do_notify_0x1A_control_cmd(v2);
  break;
case 5u:                                // MSG_CLOSE_TRUSTLET
  v5 = peerIDs.called_ID;
  ret = *(&v0 - 1);
  marshalled_cmd.ipc_arg = peerIDs.caller_ID;
  marshalled_cmd.ta_id = ipc_arg;
  if ( sub_2E56(4, peerIDs.caller_ID) )
  {
    if ( marshalled_cmd.ta_id != -1 )
    {
      v30 = marshalled_cmd.ipc_arg;
      *&marshalled_cmd.handler_id = ret.field_0;
      *a2 = ret.msg_len;
      a4 = ret.field_0;
      v6 = find_mcp_TA_struct(ret.field_0);
      if ( v6 )
      {
        call_handler_by_id(v6, a2);
        dword_F22C = sub_379C(a2);
        if ( !dword_F22C )
          sub_2440();
      }
    }
  }
  break;
case 6u:                                // MSG_CLOSE_TRUSTLET_ACK
```

```
case 11u:                               // MSG_GET_DRIVER_VERSION
  ret = peerIDs;
  if ( TA_with_id_loaded(peerIDs.caller_ID) )
  {
    *a2 = 0;
    if ( ret.msg_len )
    {
      v12 = mcp_DRV_find_by_id_unk_3(ret.msg_len);
      if ( v12 )
        *a2 = v12->driver_version;
    }
    MCP_process_ipc_cmd_complete_response(ret.field_0, 0, 11, *a2);
  }
  break;
case 12u:                               // MSG_GET_DRAPI_VERSION
  ret.field_0 = peerIDs.caller_ID;
  memset_to_0(a2, 96);
  if ( sub_3632(ret.field_0) && !tlApiGetMobicoreVersion(a2, v13, v14, v15) )
    MCP_process_ipc_cmd_complete_response(ret.field_0, 0, 12, resp_data);
  break;
case 13u:                               // MSG_SET_NOTIFICATION_HANDLER
  ret.field_0 = peerIDs.called_ID;
  ret.msg_len = peerIDs.caller_ID;
  if ( (is_trustlet_type_a_driver_SECURITY_CHECK(peerIDs.caller_ID) || is_trustlet_systemTrustlet(ret.msg_len))
    && LOWORD(ret.msg_len) == LOWORD(ret.field_0) )
  {
    sub_8394(ret.msg_len, ret.field_0);
    MCP_process_ipc_cmd_complete_response(ret.msg_len, 0, 13, ret.field_0);
  }
  break;
case 14u:                               // MSG_GET_REGISTRY_ENTRY
  MCP_process_IPC_0xE(peerIDs.caller_ID, ipc_arg);
  break;
case 15u:                               // MSG_DRV_NOT -> drApiNotifyClient
  ret = __PAIR__(ipc_arg, peerIDs.caller_ID);
  *a2 = 0;
  if ( is_trustlet_type_a_driver_SECURITY_CHECK(peerIDs.caller_ID) || is_trustlet_systemTrustlet(ret.field_0) )
  {
    if ( TA_with_id_loaded(ret.msg_len) && (wrap_find_trustlet_instance_by_driverId(ret.msg_len), v16 == 2) )
      call_ipc_signal(ret.msg_len);
    else
      *a2 = 3847;
  }
  else
  {
    *a2 = 3843;
  }
  MCP_process_ipc_cmd_complete_response(ret.field_0, 0, 2, *a2);
  break;
case 16u:                               // MSG_SET_FASTCALL_HANDLER
                                        // (drApiInstallFc)
  ret.msg_len = peerIDs.called_ID;
  *a2 = ipc_arg;
  a4 = peerIDs.caller_ID;
  *&marshalled_cmd.handler_id = 0;
  if ( is_trustlet_type_a_driver_SECURITY_CHECK(peerIDs.caller_ID) )
  {
    if ( *a2 >= 0x100000u )
      *&marshalled_cmd.handler_id = 3842;
    v17 = find_trustlet_by_driverId(a4);
    if ( sub_7F8A(v17) )
    {
      v18 = syscall_0x1A_(0x94, 0, a4, *a2, ret.msg_len, 0);// SET CUSTOM FASTCALL HANDLER
      if ( v18 )
        *&marshalled_cmd.handler_id = ((v18 & 0xFFF) << 12) + 3848;
    }
    else
    {
      *&marshalled_cmd.handler_id = 3848;
    }
  }
  else
```

REVERSE ENGINEERING SOCB: TRUSTLET NOTIFY

```
void __noreturn attach_intr_init_mcp_queues_and_process_notifications_forever()
{
  _DWORD *v4; // r0@1
  int *v5; // r0@3

  __asm { SVC             0x12 }                  // call interrupt attach for 0x8001
                                                  // this jumps to the while(1) on success
                                                  // actually, tsk tsk tsk decompiler :)

  v4 = custom_log_reserve_sg(4);
  *v4 = 0x60100001;
  assert(v4, &custom_heap_something, 0);
  while ( 1 )
  {
    call_init_MCP_queues();                       // uses a control syscall (0x1A) to get the queue addresses
                                                  // that were set up when nSW called INIT fastcall,
                                                  // then it mmaps the queues into its own address space.
                                                  //
                                                  //

    do
    {
      while ( 1 )
      {
        v5 = get_noti_from_notification_queue();
        if ( !v5 )
          break;
        send_signal_to_ta(*v5);                   // svc 0x15
      }
      svc_0x11_to_0x8001FFFF();
    }
    while ( dword_18B50 );
    maps_or_unmaps_mcp();
  }
}
```

TRUSTLETS

TRUSTLETS AND DRIVERS

- similarities and differences
- all depends on tlApi and drApi calls
- Trustlet behavior: single threaded; input from WSM tciBuf, tlApi_callDriver if driver needed
- Driver behavior: often multi threaded (handle IRQs from devices)
  - input from drApiIpcCallToIPCH: command type + length + initial command pointer
  - passed in arguments (marshalled from tlApi_callDriver by S0CB)
  - then map in Trustlet memory to read actual command buffer(s) using more drApi calls that map to S0CB IPC commands
- so, the comm. b/w trustlets&drivers is marshalled by the previously seen S0CB
  - *no* “MAC” on IPCs at all; up to drivers to filter callers for commands based on caller uuid

LABELING TLLIB APIS

- Prior art + strings in Samsung TAs covered most tlApis, but few drApis; additional reversing of secure drivers helped fill in the gaps
- "GOT" approach is really trivial: one field in trustlet/driver headers gets filled in with tlLib address at load time; all library calls simply call to this address with R0 holding the API id
- Luckily, all such calls go through small stub functions, so we can label these nicely in the code.
- Wrote a script to automatically label the APIs in a trustlet/driver binary, to aid reverse engineering

LABELING TLLIB APIS

```
; int tlApi_syscall_table[180]
tlApi_syscall_table DCD tlApiNOP+1          ; 0 ; DATA XREF:
                                            ; ROM:off_7D0AA00↑o
                    DCD tlApiGetVersion+1   ; 1
                    DCD tlApiGetMobicoreVersion+1; 2
                    DCD tlApiGetPlatformInfo+1; 3
                    DCD tlApiExit+1         ; 4
                    DCD tlApiLogvPrintf+1   ; 5
                    DCD tlApiWaitNotification+1; 6
                    DCD tlApiNotify+1       ; 7
                    DCD tlApi_callDriver+1  ; 8
                    DCD tlApiWrapObjectExt+1; 9
                    DCD tlApiUnwrapObjectExt+1; 10
                    DCD tlApiGetSuid+1      ; 11
                    DCD tlApiSecSPICmd+1    ; 12
                    DCD tlApiCrAbort+1      ; 13
                    DCD tlApiRandomGenerateData+1; 14
                    DCD tlApiGenerateKeyPair+1; 15
                    DCD tlApiCipherInitWithData+1; 16
                    DCD tlApiCipherUpdate+1 ; 17
                    DCD tlApiCipherDoFinal+1; 18
                    DCD tlApiSignatureInitWithData+1; 19
                    DCD tlApiSignatureUpdate+1; 20
                    DCD tlApiSignatureSign+1; 21
                    DCD tlApiSignatureVerify+1; 22
                    DCD tpiApiMessageDigestInitWithData+1; 23
                    DCD tlApiMessageDigestUpdate+1; 24
                    DCD tlApiMessageDigestDoFinal+1; 25
                    DCD tlApiGetVirtMemType+1; 26
                    DCD tlApiDeriveKey+1    ; 27
                    DCD tlApiMalloc+1       ; 28
                    DCD tlApiRealloc+1      ; 29
                    DCD tlApiFree+1         ; 30


                    DCD tlApiRandomGenerateData_wrap+1; 83
                    DCD tlApiCrash+1        ; 84
                    DCD tlApiEndorse+1      ; 85
                    DCD tlApiTuiGetScreenInfo+1; 86
                    DCD tlApiTuiOpenSession+1; 87
                    DCD tlApiTuiCloseSession+1; 88
                    DCD tlApiTuiSetImage+1  ; 89
                    DCD tlApiTuiGetTouchEvent+1; 90
                    DCD tlApiTuiGetTouchEventsLoop+1; 91
                    DCD tlApiDrmProcessContent+1; 92
                    DCD tlApiDrmOpenSession+1; 93
                    DCD tlApiDrmCloseSession+1; 94
                    DCD tlApiDrmCheckLink+1 ; 95
                    DCD tlApiDeriveKey_wrapper+1; 96
                    DCD tlApiUnwrapObjectExt_wrapper+1; 97
                    DCD tlApiGetSecureTimestamp+1; 98
```

```
; int drApi_syscall_table[62]
drApi_syscall_table DCD drApiGetVersion+1   ; 0 ; DATA XREF: get_syscall_fn+30↑o
                                            ; ROM:off_7D0A9F8↑o
                    DCD drApiExit+1         ; 1
                    DCD drApiMapPhys+1      ; 2
                    DCD drApiUnmap+1        ; 3
                    DCD drApiMapPhysPage4KBWithHardware+1; 4
                    DCD drApiMapClient+1    ; 5
                    DCD drApiMapClientAndParams+1; 6
                    DCD drApiAddrTranslateAndCheck+1; 7
                    DCD drApiGetTaskid+1    ; 8
                    DCD drApiTaskidGetThreadid+1; 9
                    DCD drApiGetLocalThreadId+1; 10
                    DCD drApiStartThread+1  ; 11
                    DCD drApiStopThread+1   ; 12
                    DCD drApiResumeThread+1 ; 13
                    DCD drApiThreadSleep+1  ; 14
                    DCD drApiSetThreadPriority+1; 15
                    DCD drApiIntrAttach+1   ; 16
                    DCD drApiIntrDetach+1   ; 17
                    DCD drApiWaitForIntr+1  ; 18
                    DCD drApiTriggerIntr+1  ; 19
                    DCD drApiIpcWaitForMessage+1; 20
                    DCD drApiIpcCallToIPCH+1; 21
                    DCD drApiIpcSignal+1    ; 22
                    DCD drApiIpcSigWait+1   ; 23
                    DCD drApiNotify+1       ; 24
                    DCD drApiSyscallControl+1; 25
                    DCD drApiReadOemData+1  ; 26
                    DCD drApiVirt2Phys+1    ; 27
                    DCD drApiCacheDataCleanAll+1; 28
                    DCD drApiCacheDataCleanInvalidateAll+1; 29
                    DCD drApiNotifyClient+1 ; 30
                    DCD drApiThreadExRegs+1 ; 31
                    DCD drApiInstallFc+1    ; 32
                    DCD drApiIpcUnknownMessage+1; 33
                    DCD drApiIpcUnknownException+1; 34
                    DCD drApiGetPhysMemType+1; 35
                    DCD drApiGetClientRootAndSpId+1; 36
                    DCD drApiCacheDataCleanRange+1; 37
                    DCD drApiCacheDataCleanInvalidateRange+1; 38
                    DCD drApiMapPhys64+1    ; 39
                    DCD drApiMapPhys64Page4KBWithHaardware64+1; 40
                    DCD drApiVirt2Phys64+1  ; 41
                    DCD drApiGetPhysMemType64+1; 42
                    DCD drApiUpdateNotificationThread+1; 43
                    DCD drApiRestartThread+1; 44
                    DCD drApiGetSecureTimestamp+1; 45
                    DCD drApiFastCall+1     ; 46
                    DCD drApiGetClientUUID+1; 47
                    DCD sub_7D0A2AC+1       ; 48
                    DCD drApiMapVirtBuf+1   ; 49
                    DCD drApiUnmapPhysNew+1 ; 50
                    DCD drApiMapPhysNew+1   ; 51
                    DCD drApiUnmapVirtBufNew+1; 52
                    DCD sub_7D07D76+1       ; 53
                    DCD sub_7D0A558+1       ; 54
                    DCD sub_7D0A574+1       ; 55
                    DCD sub_7D0A5D6+1       ; 56
                    DCD sub_7D0A738+1       ; 57
                    DCD sub_7D0A1BA+1       ; 58
                    DCD sub_7D0A1C8+1       ; 59
                    DCD sub_7D0A5F4+1       ; 60
                    DCD sub_7D0A60C+1       ; 61
```

TRUSTLETS AND DRIVERS

- 0006…: Widewine trustlet (drm)
- 0701…: TlCm (certificate management) trustelt
- 0706…:TEE keymaster trustlet
- 0813…: TEE gatekeeper trustlet
- f…5: HDCP trustlet (drm)
- f…a: TIMA PKM trustlet
- f…b: TIMA LKM Auth trustlet
- f…c: Key Management trustlet
- f…d: ?
- f…e: Synaptics Fingerprint trustlet
- f…f: TIMA attestation trustlet

TRUSTLETS AND DRIVERS

- f...12: CCM trustlet
- f...13: Keystore trustlet
- f...14: TUI trustlet
- f...16: SKMM trsutlet (Secure Key Management Module, used by SPay trustlets)
- f...17: MLDAP trustlet (knox mdm integration https://www.samsungknox.com/en/solutions/knox-workspace/on-premise-mdm-integration)
- f...19: Dmverity trustlet
- f...1e: OTP trustlet
- f...1f: FIDO trustlet
- f...2e: Fingerprint trustlet
- f...38: ESECOMM trustlet
- f...3e: TEE Keymaster
- f...41: ICCC

TRUSTLETS AND DRIVERS

- f...1c: VISA Pay
- f...21: Mastercard Pay
- f...27: PLCC Pay
- f...28: KRCC Pay
- f...31: Discovery Pay
- f...3a: CHNCMM Pay
- f...33: JIC Pay
- f...32: CNCC Pay
- f...39: EURCOMM Pay

TRUSTLETS AND DRIVERS

- ffffff0....1: SRPMB driver
- ffffffd....4: Crypto driver
- ffffffd....a: TIMA driver
- ffffffd....e: SEC SPI driver #1 (fingerprint sensor)
- ffffffd....14: TUI driver
- ffffffd....17: SEC SPI driver #2 (eSE)

TLC IN ANDROID

T-BASE AND ANDROID

- Debugging info in filesystem?
  - /system/bin/tima_dump_log
    /system/tima_measurement_info (attestation)
    /system/kern_sec_info{1|2|3|4] (pkm)
    /proc/tima_secure_log
    /proc/tima_secure_rkp_log
    /proc/tima_debug_log
    /proc/tima_debug_rkp_log (rkp)
    /sys/kernel/debug/trustonic_tee/*
    /proc/sec_log

T-BASE AND ANDROID

- Looking around
  - strings -f * | grep onTransact
    strings -f * | grep mcNotify
  - lib names in /system/lib64/ & /system/vendor/lib64 with “tlc”
  - service list | grep {com.sec,finger,keystore} etc.
  - finally, settled on tlc_server. Exposes access to certain trustlets via Binder. (ESECOMM, CCM, DCM, TUI)

TLC SERVER

- one instance started for each TA (CCM, DCM, TUI, ESECOMM, PUF)
- uses dlopen/dlsym to get comm_data initialization function, e.g. esecomm_get_comm_data in libtlc_tz_esecomm.so; this identifies trustlet to load
- registers binder service, handler a simple switch of commands: OPEN, CLOSE, COMM, COMM_VIA_ASHMEM

TLC SERVER

- OPENSWCONN: create_comm_cxt()
- COMM, COMM_VIA_ASHMEM: parse arguments out of Binder parcel and send via libMcClient.so
  - COMM: uses SEAMS to validate caller based on SEAndroid policy
  - COMM_VIA_ASHMEM: no authentication! **COMPLETELY OPEN**

```
comm_cxt = create_comm_ctx(                 // TLC_COMM_TYPE:
                                            // 0 - proxy
                                            // 1 - direct
                                            //
                                            // ==> we create direct
                                            //
                                            //
                                            // direct_comm_cxt()
                                            // - instantiate directCommImpl with these parameters
                                            //    - includes root (== device id, switched out to 0) and process (==uuid)
                                            // - call tlc_open
                                            //    - mcOpenDevice(0) and mcOpenSession to uuid,
                                            // after it mmaps the TLC buffer, to sendmsglen+recvmsglen length
        TLC_COMMUNICATION_TYPE_DIRECT,
        comm_data_root,
        comm_data_root_strlen,
        comm_data_process,
        comm_data_process_strlen,
        comm_data_max_sendmsg_size,
        comm_data_max_recvmsg_size);
```

VULNERABILITIES

ATTACK SURFACES

- unpriviliged Android > TLC
- privileged Android > Trustlet (tci)
- privileged Android > IPCH (MCP implementation)
- privileged Android > tbase kernel (SMC)
- privileged Android > custom tbase fastcalls (SMC) and ATF SMCs
- Trustlet > Driver (IPC)
- Trustlet > tbase kernel (SVC)
- Trustlet > IPCH (IPC)

FOUND VULNERABILITIES

- Authentication bypass in tlc_server
- Memory corruption in tlc_server
- Plenty of memory corruption vulnerabilities in trustlets (buffer overflows, arbitrary writes, integer overflow-to-buffer overflows)
- Session hijack logic vulnerability in CCM Trustlet
- Memory corruption and race condition vulnerabilities in TIMA driver
- Feature abuse in TIMA driver leading to arbitrary Android kernel read (KASLR bypass)

STACK BOF IN ESECOMM

- ESECOMM: implements interface to eSE (NFC enabled for Samsung Pay)
- smart card communication: ISO7816
  - http://www.cardwerk.com/smartcards/smartcard_standard_ISO7816-4_4_abrev_and_notation.aspx
  - Uses SEC SPI driver to talk to eSE
  - SPI driver: there is also a Linux kernel driver (see /drivers/spi/spi-s3c64xx.c), this explains meaning of memory mapped I/O
- Implements “SCP03 Global Platform Secure Channel Protocol”
  - http://csrc.nist.gov/groups/ST/ssr2016/documents/presentation-mon-traore.pdf
  - uses an APDU based protocol with TLV parameters to set up secure channel cryptographic information (Diffie-Hellman keys)

STACK BOF IN ESECOMM

```
signed int __fastcall process_ScpInstallCaCert(tci_msg_add_ca_payload_t *reqmsg, tci_rsp_payload_t *rspmsg)
{
  tci_msg_add_ca_payload_t *reqmsg_; // r4@1
  tci_rsp_payload_t *rspmsg_; // r5@1
  char *reqmsg_payload_; // r7@1
  signed int result; // r0@2
  char pubkey[512]; // [sp+Ch] [bp-244h]@2
  char caid[32]; // [sp+20Ch] [bp-44h]@2
  char curveid; // [sp+22Ch] [bp-24h]@2
  int pubkey_len; // [sp+230h] [bp-20h]@2
  unsigned int caid_len; // [sp+234h] [bp-1Ch]@2

  reqmsg_ = reqmsg;
  rspmsg_ = rspmsg;
  reqmsg_payload_ = reqmsg->payload;
  if ( validate_input_len(reqmsg->payload, reqmsg->total_len, reqmsg->payload, (char *)&reqmsg->total_len) )  //
ONLY verifies the total input len, not related to TLV lengths
  {
    result = parse_ca_cert(reqmsg_payload_, reqmsg_->total_len, caid, (int *)&caid_len, &curveid, pubkey,
&pubkey_len);
    (…)
```

STACK BOF IN ESECOMM

```
signed int __fastcall parse_ca_cert(char *msg_payload, int total_req_payload_len, void *caid, int *caid_len, char *curveid,
void *pubkey_buf, int *pubkey_buf_len)
{
  (…)

  v7 = msg_payload;
  total_req_payload_len_ = total_req_payload_len;
  caid_ = caid;
  v10 = caid_len;
  v11 = -1;
  memset_to_0(&out_buf, 0x44u);
  if ( parse_tlvs_from_APDU(&out_buf, v7, 0, total_req_payload_len_) < 0 )
  {
    snprintf(logbuffer, 119, "%s:%d :: Error, %s\n", "parse_ca_cert", 73, "failed to parse TLV");
    logbuffer[119] = 0;
    tlApiLogPrintf_0("%s\n", logbuffer);
    return 3;
  }
  tlv_caid = find_tlv_obj_in_parsed_tlvs(&out_buf, (char *)&caid_tag_value);
  tlv_caid_ = tlv_caid;
  if ( tlv_caid )
  {
    memcpy_w(caid_, &tlv_caid->tlv_value_OR_num_extra_tlvs, tlv_caid->tlv_length);
```

CCM SESSION HIJACK

- CCM: Client Certificate Management
  - supposed to secure private certs, basically
  - https://seap.samsung.com/api-references/android-premium/reference/com/sec/enterprise/knox/ccm/TZ_CCM_PKCS11_Guide.pdf
- Keys never leave TZ, CCM is an encryption/decryption oracle.
- To secure this, at CA install time certs are secured with some authentication method (password, TUI)

CCM SESSION HIJACK

- So, how does login exactly work?
- First, session must be opened with C_OpenSession
  - 64 bit random session id generated
- Then, session must login, using C_Login
  - pass session id to identify session
  - if successful, session->logged_in bit flipped
- After this, until C_CloseSession, all operations will be allowed where session id matches

CCM SESSION HIJACK

- That creates a race condition window:
  - IF legitimate user logs a session in,
  - ANYBODY can impersonate the session that knows the session id
- 64 bit randomness from crypto engine though, so that seems solid
- Except…

CCM SESSION HIJACK

- The session ids are printed into the /proc/sec_log …
- Even if an app context would be restricted: the actual secure log implementation is the same buffer for ALL trustlets: by hijacking ESECOMM, we can read the sec_log even if we can't read /proc/sec_log itself b/c of SEAndroid

```
v16 = TZ_get_rand_data(rand, &a2);
if ( v16 || a2 != 8 )
{
  snprintf(logbuff, 119, "TL_TZ_CCM: C_OpenSession rand gen failed with ret = 0x%x", v16, v20, v21, v22);
  logbuff[119] = 0;
  tlApiLogvPrintf2("%s\n", logbuff);
  v4 = 5;
}
else
{
  snprintf(logbuff, 119, "TL_TZ_CCM: OpenSession ID: 0x%llx, len: %d", 0, *(_DWORD *)rand, v25, a2);
  logbuff[119] = 0;
  tlApiLogvPrintf2("%s\n", logbuff);
```

TIMA DRIVER KASLR BYPASS

- Once we hijack ESECOMM, we can talk to the TIMA driver
- Implements many different things:
  - TIMA measurement golden copy area read/write (trustlets use this to verify/set warranty info)
  - nSW kernel hashing (for TIMA PKM) (every UUID!)
  - MST driver (for Samsung Pay) (only whitelisted UUIDs)
  - SCrypto (a full openSSL stack based FIPS compliant crypto library… in addition to the Crypto Driver, hmmm.) (every UUID!)
  - modify UUID whitelists (restricted to CNCC trustlet)

TIMA DRIVER KASLR BYPASS

- measurement info read/write:
  - this would allow arbitrary physical address read/write
  - Gal Beniamini described this
  - however, in practice, range checks limit what can be written to the intended areas
  - Nonetheless - you could STILL replace the golden measurement/warranty info (don't try this at home…)

TIMA DRIVER KASLR BYPASS

```
signed int __fastcall is_addr_range_valid_for_phys_access(unsigned int addr_start, int len, int access_type)
{
  unsigned int addr_end; // r3@1
  const char *v4; // r2@2
  unsigned int tima_unk_phys_addr_end; // r6@3
  int v6; // r3@15
  const char *v7; // r2@15

  addr_end = addr_start + len;
  if ( addr_start + len >= addr_start )
  {
    tima_unk_phys_addr_end = addr_from_tima_sfr_start_of_SW + 0x100000;
    if ( access_type )
    {
      if ( access_type != 1 )
      {
        v4 = "OMG!!! non-supported action";
        goto LABEL_6;
      }
      if ( addr_start >= addr_from_tima_sfr_start_of_SW + 0xFFC00 && addr_end <= tima_unk_phys_addr_end
        || addr_start >= addr_from_tima_sfr_start_of_SW + 0x164 && addr_end <= addr_from_tima_sfr_start_of_SW + 0x360 )
                                                //
                                                // all right, so quite clearly we are NOT
                                                // allowed to just write to any physical address,
                                                // unfortunately, because we are limited to this
                                                // TIMA range.

      {
        return 0;
      }
      v6 = addr_start;
      v7 = "OMG!!! physical address %x is not allowed for writing";
    }
    else
    {
      if ( addr_start >= addr_from_tima_sfr_start_of_SW && addr_end <= tima_unk_phys_addr_end
        || addr_start >= 0x80000000 && addr_end <= 0x80200000
        || addr_start >= 0x2023800 && addr_end <= 0x2024000
        || addr_start >= 0x80000000 && addr_end <= addr_from_tima_sfr_start_of_SW )
      {
        return 0;
      }
      v6 = addr_start;
      v7 = "OMG!!! physical address %x is not allowed for reading";
    }
    snprintf((int)logbuff, 119, v7, v6);
    goto LABEL_23;
  }
  v4 = "addr_pa + len overflow";
LABEL_6:
  snprintf((int)logbuff, 119, v4, addr_end);
LABEL_23:
  logbuff[119] = 0;
  drApiLogvPrintf_Wrap("%s\n", logbuff);
  return 1;
}
```

TIMA DRIVER KASLR BYPASS

- What's left in TIMA?
  - SCrypto: I actually found several memory corruption vulnerabilities in here. Disclosure not finished yet.
  - PKM MD5 hashing command
- Hash command is open to all UUIDs! (Should have been restricted to TIMA PKM trustlet)
  - address range allows all of physical memory within which nSW lives
    - to break Android KASLR: invoke from hijacked ESECOMM to hash first page of kernel physical address space: this is an empty page with just the value of the KASLR slide in it
    - from the hash, it is a <32 bit bruteforce to recover the slide.

TAKEAWAYS

- even an almost entirely black box TEE OS implementation can be reversed fairly comprehensively
- t-base architecture solid for isolation, but lacks modern exploit mitigations
- driver api design less susceptible to exploitability of input parsing memory corruption vulns, much worse for trustlets
  - still, architecture should find a way to sandbox secure drivers, e.g. filter what areas can be mapped by drivers, what drivers can make fastcalls etc.
  - improve driver whitelists for allowed commands!
- altogether, t-base provides a huge attack surface to an elevated Android privilege context and it does not help vendors to eliminate or deal with unprivileged Android attack surface

QUESTIONS?