EXPLORING THE GALAXY

Building emulators to find vulnerabilities in modern phones

Alexander Tarasikov

About me (@astarasikov)

- In previous life
  - Porting Linux to smartphones
  - Operating Systems development
  - Virtualisation and emulation for ARM
- Currently: Product Security at a Qualcomm
  - Here only privately as an independent speaker
  - Disclaimer: This talk represents results of my own research done in free time and is not based on information I could access due to my employment with Qualcomm

Where it all started

- Me and an unnamed researcher taking a walk in the nature
- We gotta look for bugs in those Exynos phones, huh?

Black Mountain, San Diego, CA

Later that day

- Found an unsigned partition with graphics/splash screen
  - Nothing that looks like a signature at the end of it
  - It can't be *that* easy?
- Reported to Samsung
  - They acknowledge the risks but won't change anything unless there is a real bug reported.
- A weighted and a reasonable response. But we'll see about that later.

Attempt 1

- Tried to add enough support to run the bootloader
- Had lots of funny ideas
  - Taint tracking to simplify reverse-engineering MMIO?
  - Similar project(s): Check out AFL++ with QEMU mode and CMPCOV for fuzzing
- Ultimately gave up because I got bored of single-stepping everything in GDB

When conditional instruction:
Print latest address/value pair for the reg

When checking the flags via cmp:
Save operands (immediate or reg val)
Look up last load address for registers

When ldr:
Save source address for register index

When assigning register:
Reset source address or copy from operand

One year later

- S10 with Exynos9820 comes out
  - first non-iPhone ARM phone with high single-thread performance
  - I initially got one to do performance tests
- I decided to definitely find some bugs and score some CVEs
  - I mean, I've been telling this to myself when buying every device in the last couple years
  - Time to recover that money

Reverse-engineering

- Updates are TAR files with partition binaries
- I focused primarily on the bootloader
  - Parts of bootloader are signed
  - We can identify them by entropy
  - Blocks of zeroes followed by garbage

- 0x0: probably EPBL (early primitive bootloader) with some USB support
- 0x13C00: ACPM (Access Control and Power Management?)
- 0x27800: some PM-related code
- 0x4CC00: some tables with PM parameters
- ... -> either charger mode code or PMIC firmware
- 0xA4000: BL2, the actual s-boot
- 0x19E000: TEEGRIS SPKG (CSMC)
- 0x19E02B: TEEGRIS SPKG ELF start
  This handles some SMC calls from the bootloader
- 0x1ACE00: TEEGRIS SPKG (FP_CSMC)
- 0x1ACE2B: TEEGRIS FP_CSMC (ELF header).

- 0x264000: TEEGRIS kernel, relocate to 0xffffffff0000000
- 0x29e000: EL1 VBAR for TEEGRIS kernel. ffffffff0041630: syscall table,
- 0x2D4000: startup_loader package
- 0x2D4028: startup_loader ELF start.

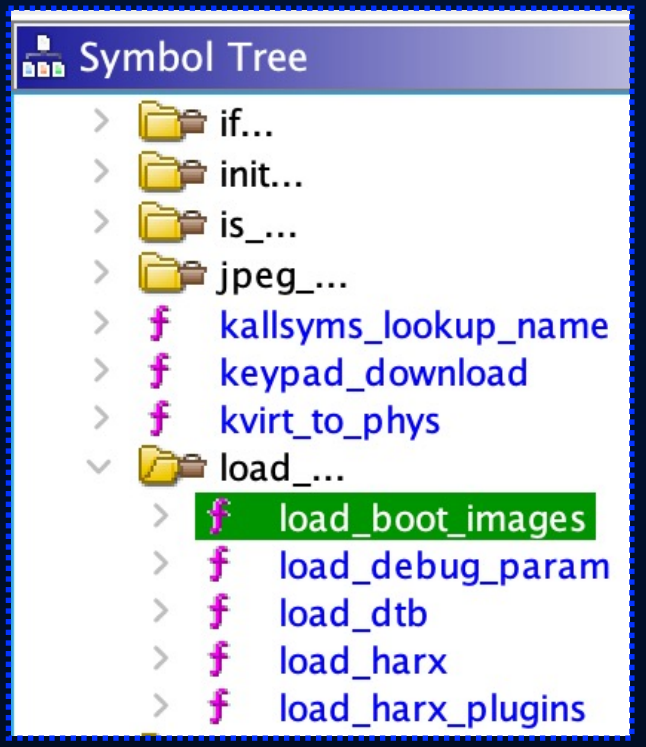
Reverse-engineering

- We can make an educated guess about the RAM addr from DTB
  - One can also use debug log menu (*#9900#) to get S-Boot log
- The actual address does not matter much
  - Once disassembled, look around image start and end
  - Start is exception handler code, end is data
  - Both contain hardcoded pointers which allow to determine base addr

Reverse-engineering

- Embedded binaries often contain debug strings
- We can leverage this to identify function names
  - A script to rename functions from debug prints
  - https://gist.github.com/astarasikov/1a67b948f3ca61f348b8e3eccf963a42

```
printf_debug(s_Could_not_do_normal_boot._(inval_e81193f8);
add_odininfo(s_%s:_Could_not_do_Narmal_Boot_(In_e8119428,s_load_boot_images_e81197
```



Reverse-engineering

- Looked around aimlessly and found something that looks like “memcpy"
  - You know it when you spend too much time with ARM assembly
  - Hmm, let's examine the callers. A lot of them. Luckily, some look weird.
  - Perhaps it’s not the best way to find bugs
- Figured out Samsung deployed a new TEE OS - TEEGRIS
  - New code, must be buggy

building emulators with QEMU: Bootloader

- Need to emulate basic peripherals: RAM controller, timer, UFS
  - 16 registers needed to be emulated in total
- Enough to have UART output and fallback into recovery/crash mode
- Need to implement basic TEE calls in QEMU
  - Can leverage PSCI code for that
- At this point, we can use QEMU to develop our exploits
- Can inject code with GDB and single-step afterwards

building emulators with QEMU: Bootloader

1. Move RAM region to match Exynos HW

```
-     [VIRT_MEM] =                { 0x40000000, RAMLIMIT_BYTES },
+     //[VIRT_PLATFORM_BUS] =       { 0x2c000000, 0x02000000 },
+     [VIRT_SECURE_MEM] =         { 0x2e000000, 0x01000000 },
+     //[VIRT_PCIE_MMIO] =          { 0x10000000, 0x2eff0000 },
+     //[VIRT_PCIE_PIO] =           { 0x3eff0000, 0x00010000 },
+     //[VIRT_PCIE_ECAM] =          { 0x3f000000, 0x01000000 },
+     [VIRT_MEM] =                { 0x80000000, RAMLIMIT_BYTES },
```

2. Add peripheral region @addr 0

```
#define FAKE_PERIPH_ADDR 0x00000000
#define FAKE_PERIPH_SIZE 0x20000000
```

3. Register peripheral emulation device

```
static void create_fake_periph(const VirtMachineState *vms, qemu_irq *pic)
{
    char *nodename;
    hwaddr base = vms->memmap[VIRT_FAKE_PERIPH].base;
    hwaddr size = vms->memmap[VIRT_FAKE_PERIPH].size;
    DeviceState *dev = sysbus_create_simple("fake_periph", base, 0);
```

4. Implement register emulation until it boots

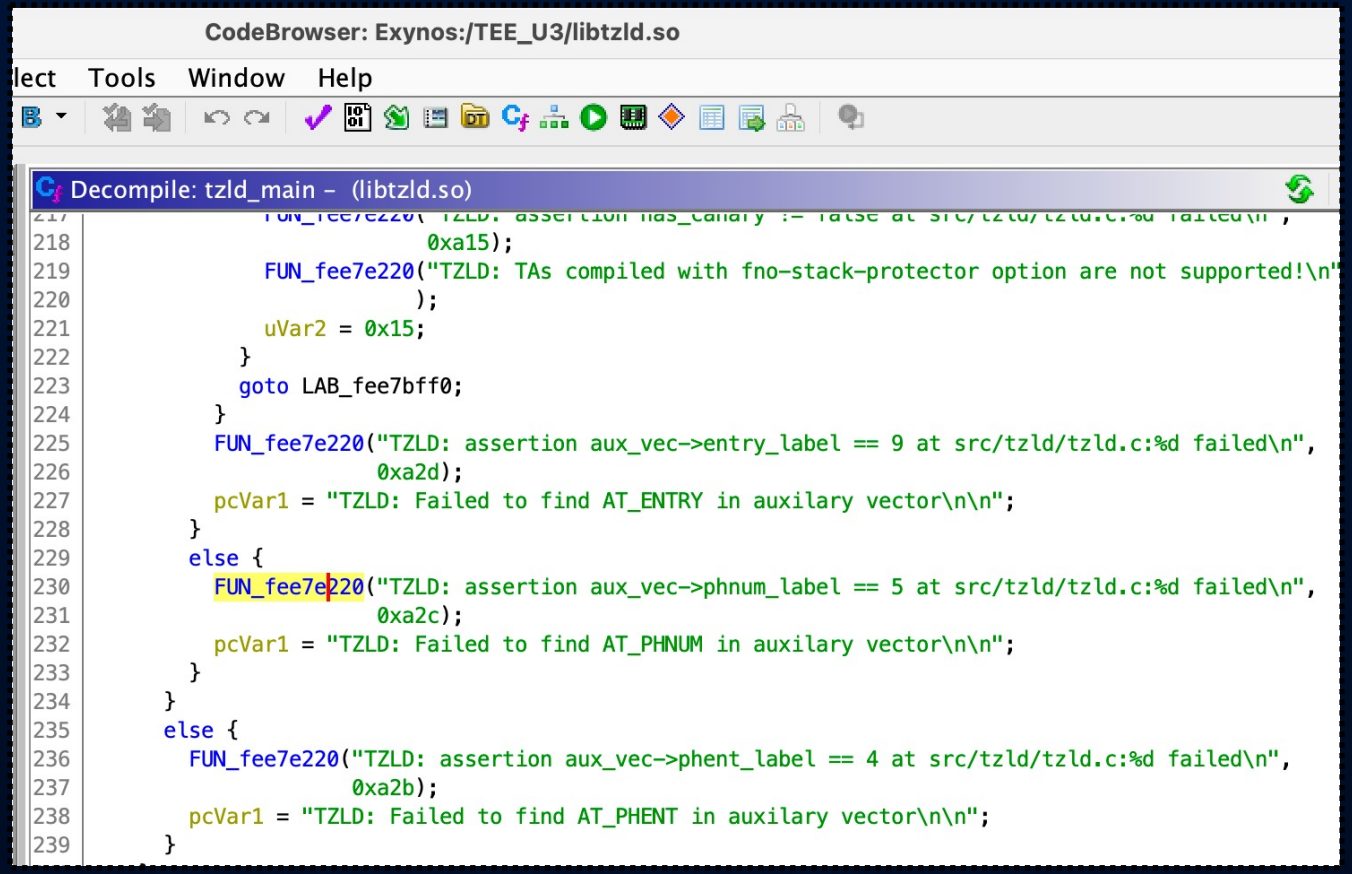
```
static uint64_t fake_periph_read(void *opaque, hwaddr offset,
                                 unsigned size)
{
    FAKE_PERIPHState *s = (FAKE_PERIPHState *)opaque;
    int i;
      uint64_t ret, *cache;

      //use the original offset to hit the cache of the current qperiph instance
      cache = &s->cache[offset / sizeof(uint64_t)];
      ret = *cache;

      //for the switch below, use absolute physical addresses for simplicity
      offset += s->iomem.addr;

    switch (offset) {
              case 0:
                      ret = 0; //some placeholder
                      break;

              case 0x108300a4:
                      ret = *cache;
                      break;

              /***** UFS *****/
```

building emulators with QEMU: TrustZone

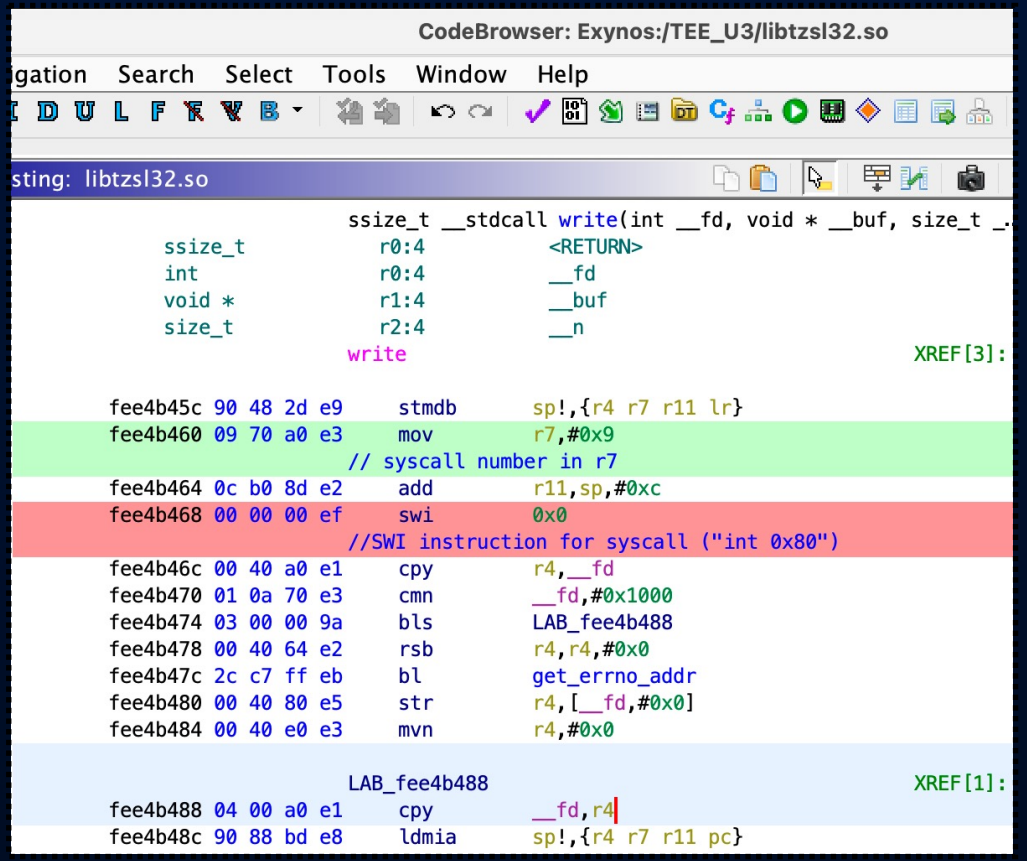
- QEMU has “Linux-user” target to run Linux binaries for another arch
- Somewhat easy to leverage for fuzzing TEEs
  - Need to modify the ELF loader to setup correct AUX vectors (ELF parameters, like argc/argv for main, but earlier)
  - Need to implement TEEGRIS syscalls
    - I knew half of them from disassembly of the kernel.
    - Libtzsl.so just exports C wrappers for all syscalls too, much easier.
- with AFL++ and QEMU mode, we can get coverage-guided fuzzing
  - All fine and good, but have found much better bugs with code review

building emulators with QEMU: TrustZone

reverse-engineering ELF AUX vals - just look at the “entry” function



Syscalls: not difficult either



S-Boot bug #1

- Stumbled upon the USB download protocol, also known as ODIN
  - Normally, PC sends “ODIN”, phone replies “LOKE” for download mode
  - There’s also upload mode
  - Previous research in this area http://hexdetective.blogspot.com/2017/02/exploiting-android-s-boot-getting.html
- From time to time, new commands are added there
- In S10, compared to S9, there’s a new command “SECCMD”

S-Boot bug #1

- Something looks off here
- I had no experience exploiting heap corruptions
- Wrote a POC to fill memory with “AAAAAA” but it did not crash
- Samsung initially rated as DoS

```
iVar5 = memcmp(inputBuffer,s_SECCMD_8f0da260,6);
puVar3 = UNK_RX_Buf;
if (iVar5 == 0) {
  dynbuf = (char *)malloc1(0x80);
  inp_len = strlen((char *)inputBuffer);
  memcpy(dynbuf,inputBuffer,inp_len);
  clear_emc_token(dynbuf);
  goto strlen_and_tx_resp;
}
```

S-Boot bug #1

- I knew that any heap corruption is a potential RCE
  - I set off to exploit this one
- S-Boot heap allocator uses inline metadata and linked lists
  - Good candidate for exploitation
  - Heap header followed by payload
  - Upon alloc, splits heap in two
  - OOB write corrupts next heap hdr

```
typedef struct HeapStruct_ {
    uint32_t heap_size;//0
    uint32_t is_free;//4
    uint64_t prev_heap;//8
    uint64_t next_heap;//0x10
} __attribute__((packed)) HeapStruct;
```

S-Boot bug #1

- Allocation algorithm: iterate all chunks and find the smallest that fits
- If it's still too big, split it into two

```
do {
  foundGoodHeap = prevGoodHeap;
  if ((((*(char *)&tmp_heap->is_free != 0) &&
       (foundGoodHeap = prevGoodHeap, alloc_size <= tmp_heap->heap_size)) &&
      (foundGoodHeap = tmp_heap, prevGoodHeap != (HeapStruct *)0x0)) &&
     (foundGoodHeap = prevGoodHeap, tmp_heap->heap_size < prevGoodHeap->heap_size))
    foundGoodHeap = tmp_heap;
  }
  tmp_heap = (HeapStruct *)tmp_heap->next_heap;
  prevGoodHeap = foundGoodHeap;
} while (curHeapStart != tmp_heap);
```

S-Boot bug #1

- Heap header is followed by data
  - Data is followed by the next heap header
  - We can link a new heap by writing OOB data
- Sometimes, we get a crash in free/malloc afterward
  - Remember, allocator always chooses the largest heap
  - We can mark the chunk as free but with a huge size

S-Boot bug #1

- So we can link our own chunk into the heap
- But where do we put our payload?
- Hard to use USB stack - almost always got a crash if corrupting more than 16 bytes
- Remember the unsigned splash screen partition?
- Can we put it into the splash JPEG?

S-Boot bug #1

- Not so fast
- Upon allocation, we get a controlled write when heap is split
- Unfortunately, heap_size is 32 bit & splash memory is past S-Boot range

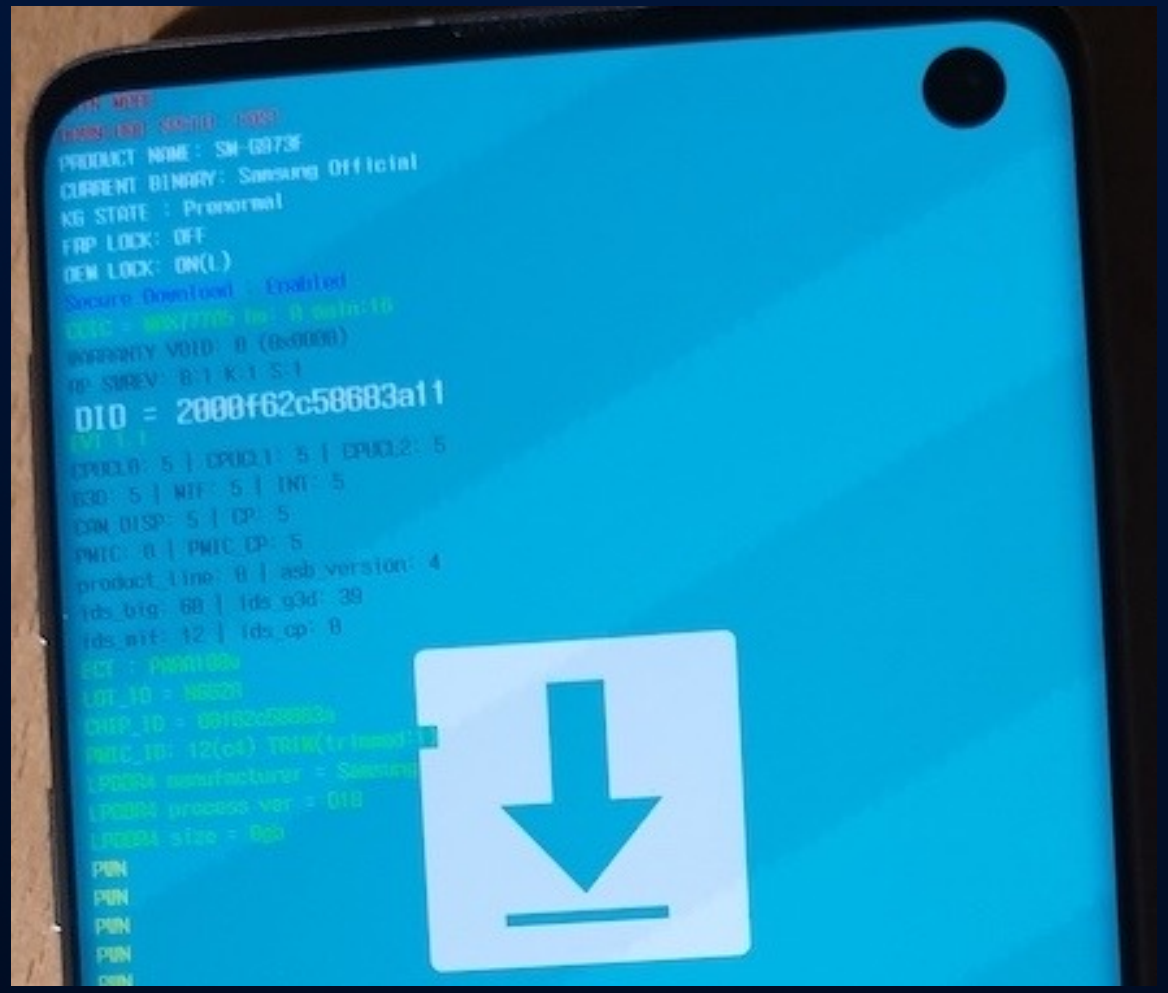
```
newSplitHeap = (uint *)((longlong)foundGoodHeap +
                        ((ulonglong)foundGoodHeap->heap_size - (ulonglong)alloc_
*newSplitHeap = alloc_size;
*(undefined *)(newSplitHeap + 1) = 0;
*(HeapStruct **)(newSplitHeap + 2) = foundGoodHeap;
*(uint **)(newSplitHeap + 4) = foundGoodHeap->next_heap;
*(uint **)(foundGoodHeap->next_heap + 2) = newSplitHeap;
foundGoodHeap->next_heap = newSplitHeap;
malloc_return = (uint **)(newSplitHeap + 6);
foundGoodHeap->heap_size = (foundGoodHeap->heap_size - 0x18) - alloc_size;
```

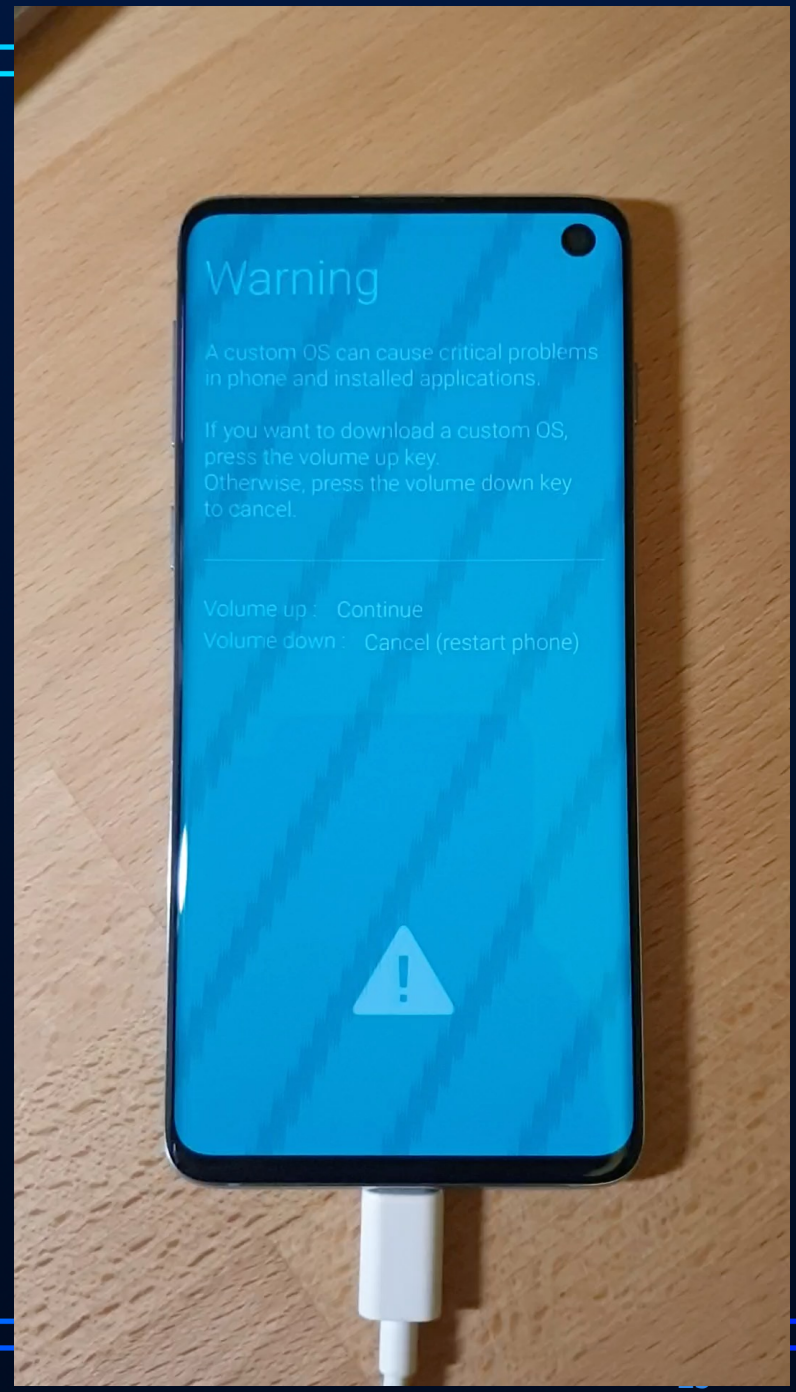
```
typedef struct HeapStruct_ {
    uint32_t heap_size;//0
    uint32_t is_free;//4
    uint64_t prev_heap;//8
    uint64_t next_heap;//0x10
} __attribute__((packed)) HeapStruct;
```

S-Boot bug #1

- Fortunately, there's an address before s-boot memory which we can use
- Download buffer for ODIN is at a low address in memory
- Unfortunately, once you enter ODIN mode there's no way to go to command mode and execute "SECCMD"
- Fortunately, DDR retains its contents for a while
  - Let's put data via the ODIN mode and reset the phone while holding the bootloader key combination

S-Boot bug #1


PRODUCT NAME: SM-G973F
CURRENT BINARY: Samsung Official
KG STATE : Prenormal
FRP LOCK: OFF
OEM LOCK: ON(L)
Secure Download : Enabled
WARRANTY VOID: 0 (0x0000)
DIO = 2000f62c58683a11


Warning
A custom OS can cause critical problems in phone and installed applications.
If you want to download a custom OS, press the volume up key.
Otherwise, press the volume down key to cancel.
Volume up : Continue
Volume down : Cancel (restart phone)

S-Boot bug #1

- How to defeat W^X and ASLR?
  - There's no ASLR
- But did you have to use ROP?
  - ODIN download buffer is RWX…

S-Boot bug #2

- Remember, splash partition (“up_param”) is not signed
  - Splash screen is a JPEG among many in the TAR file
  - I spent days trying to find a bug in libJPEG or its glue code
- Eventually, found something funny.

```
drawimg_buf = malloc_internal(0x100000);
g_drawimg_buf_1M_ptr = g_drawimg_buf_1M;
*(void **)g_drawimg_buf_1M = drawimg_buf;
if (drawimg_buf == (void *)0x0) {
  printf(s_%s:_img_buf_alloc_fail_c90d39e8,s_drawimg_c90c2998);
  uVar5 = 0xffffffff;
}
else {
  memset(drawimg_buf,0,0x100000);
  uVar5 = find_param_file_wrap(__s,*(undefined8 *)g_drawimg_buf_1M_ptr,0);
```

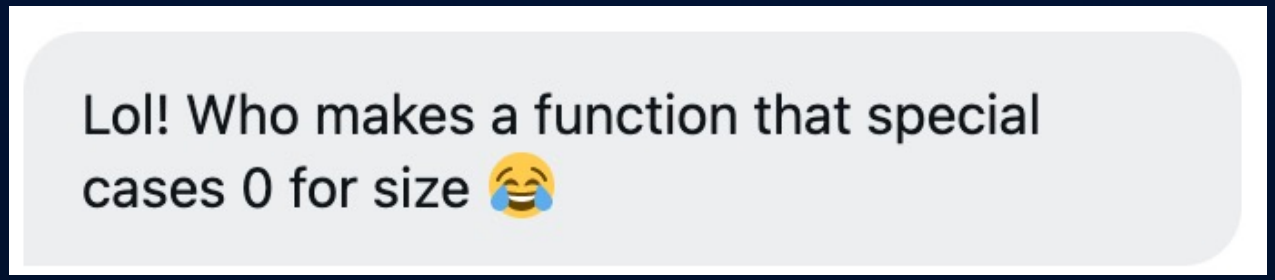
S-Boot bug #2

- Typically, one needs both pointer and size to check for OOB safely
- Here, the caller passes pointer and zero instead of size

```
iVar2 = strcmp(local_200,FILENAME);
if (iVar2 == 0) {
  size_from_TAR = tar_read_file_start__ParseIntWithBase(auStack388,8);
  if ((max_size < size_from_TAR) && (max_size != 0)) {
    printf(s_%s:_read_fail!_(%d_<_%d)_c90da760,s_find_param_file_c90c6ac0,max_size,
           size_from_TAR);
    return 0;
  }
  iVar2 = blk_bread_ufs_or_mmc_wrap(partition,(int)uVar4 + 1,size_from_TAR & 0xffffffff,OUT);
```

S-Boot bug #2

- Reactions from fellow researchers who diffed the patch



  Exploitation should have been similar in principle.
- Unfortunately, s-boot crashed upon using the old attack
- Perhaps, heap allocation pattern is very different

S-Boot bug #2

- What does s-boot even do upon a crash?
- Turns out, it re-initializes heap but leaves all DATA/BSS intact
- So I just corrupted a function pointer in DATA
- Upon a crash, during re-init, our function gets executed

S-Boot bug #2

- Bonus: remember using ODIN buffer and a hot reboot to put heap data?
- I found another location
  - Examined all calls to “blk_read”
  - Turns out, “keystorage” partition is half a megabyte in size, it’s read to RAM but AVB only validates a few kilobytes
  - We can overwrite unused data after AVB footer with our payload
- And that’s how you get persistent code-exec in bootloader

RKP hypervisor

- RKP is a hypervisor from Samsung
- Also known as UH - micro hypervisor
- Its main job is to enforce RO/RX memory mapping for Linux
- Also implements helpers for ROP mitigation
- Previously researched extensively by Google Project Zero

RKP hypervisor

- The binary is ELF, but obfuscated - no debug strings
- No problem, let’s find the exception handler and research from there
- Eventually mapped HVC call handlers
- They all follow a pattern of using a function to validate EL1 addresses
- You don’t want hypervisor to corrupt its own memory. I wouldn’t want it.

RKP hypervisor

- If addr looks like Linux VA address, just add some offset
- Otherwise, use MMU ("AT" insn) to translate
- The latter case also checks the addr against the restricted list

```
el1_paddr_t VirtToPhysWrap_MAYBE_EL1_TO_EL2(ulonglong addr)

{
  el1_paddr_t eVar1;

  eVar1 = 0;
  if (addr != 0) {
    if ((addr & 0xffffffc000000000) == 0xffffffc000000000) {
      if ((addr >> 0x26 & 1) == 0) {
        eVar1 = addr - EL1_user_offset;
      }
      else {
        eVar1 = (addr & 0x3fffffffff) + 0x80000000;
      }
    }
    else {
      eVar1 = virt_to_phys_el1_wrap(addr);
      if (eVar1 == 0) {
        if ((addr >> 0x26 & 1) == 0) {
          eVar1 = addr - EL1_user_offset;
        }
        else {
          eVar1 = (addr & 0x3fffffffff) + 0x80000000;
        }
        pa_restrict_sth(eVar1);
      }
    }
  }
  return eVar1;
}
```

RKP hypervisor

- At first, I thought I got scooped because Samsung posted a fix with a similar description
- Turns out, the “fix” was checking the address for NULL
- Some RKP calls effectively perform “write-where” after validating (or not validating) EL1 addresses
- Bonus bug: some validated base address but not the (untrusted) offset added to the base

RKP hypervisor

- I anyway started this project with one real goal
- I am disappointed EL2 is not available to end-users because OEMs used it for DRM purposes
- So…?
  - We have an exploit to go EL1->EL2
  - Tried to boot Linux in EL2
  - Mostly works, on 7 cores out of 8
  - Got KVM running Windows 10 with native speed

HArx

- Well anyway I accidentally updated my phone
- My precious RKP bugs were gone
- I was furious, but I also just got the latest S20 phone
- Motivation to continue research

HArx

- HArx is the new hypervisor on Galaxy S20 series
- It loads other hypervisors as plugins
  - Probably in anticipation of Hyper-V
- Besides that, it sets up SMMU
- Largely went unnoticed by the RE community so far…
  - A topic for another talk maybe?

A bit about TEE

- TEEGRIS kernel looks solid when it comes to validating input
- TEE apps core is also solid - parsers for certificates, TLVs
- However, there are dozens of bugs in entrypoints
- My impression is that TAs had already been researched/fuzzed
  - But got ported in a rush without understanding the trust boundary

TEEGRIS trusted apps

- Not checking input/output buffer length
  - Rather unlikely to corrupt TA memory, just other EL1 buffers
- Leftover debug code with string parsing
  - Classic buffer overflows. At least there are stack canaries in most TAs
- Camera and biometrics TAs need to access physical memory
  - Initially there were no allow-lists per TA
  - EL1 could ask TA to “mmap” TA’s memory range

TEEGRIS trusted apps

Entrypoint: processing untrusted arguments from Linux.
OOB write past the arguments buffer.

```
undefined4
TA_InvokeCommandEntryPoint
          (undefined4 uParm1,tz_bio_input_arg_t *ptParm2,undefined4 uParm3,int *piParm4)

{
  undefined4 uVar1;
  byte p4_copy;
  int at_p4;

  p4_copy = (byte)piParm4;
  PAL_DbgLog("TA_InvokeCommandEntryPoint commandID is %d in\n",ptParm2,uParm3,piParm4);
  at_p4 = *piParm4;
  uVar1 = bio_subModule_bf_cmd_handler((byte)ptParm2,(byte)at_p4,(byte)uParm3,p4_copy);
  *(tz_bio_input_arg_t **)(at_p4 + 0x104) = ptParm2;
  *(undefined4 *)(at_p4 + 0x72cf8) = uVar1;
```

TEEGRIS trusted apps

MMAP arbitrary address with TrustZone privileges

```
pgoff_field_0x10 = *(uint *)(val_from_parm4 + 0x10) & 0xfff;
uVar1 = (uint)(*(int *)(val_from_parm4 + 0xc) * *(int *)(val_from_parm4 + 8) * 3) >> 1;
uVar5 = pgoff_field_0x10 + uVar1 >> 0xc;
if ((uVar1 + pgoff_field_0x10 & 0xfff) != 0) {
  uVar5 = uVar5 + 1;
}
__addr = mmap((void *)0x0,uVar5 << 0xc,3,8,__fd,local_44);
if (__addr == (void *)0xffffffff) {
  puVar4 = (undefined4 *)get_errno_addr();
  printf("PHYS_DEV : failed to mmap phys memory. errno: %d\n",*puVar4);
  close(__fd);
  return 0xffff0000;
}
uVar2 = get_preview_frame_NV21
                  ((void *)((int)__addr + pgoff_field_0x10),*(int *)(val_from_parm4 + 8),
                   *(int *)(val_from_parm4 + 0xc),*(int *)(val_from_parm4 + 0x18),
                   *(int *)(val_from_parm4 + 0x1c),(void *)(val_from_parm4 + 0x110));
```

Related Research

- TeamT5 S-Boot bug (duplicate with mine) https://teamt5.org/en/posts/blackhat-s-talk-breaking-samsung-s-root-of-trust-exploiting-samsung-secure-boot/
- https://census-labs.com/news/2020/07/22/emulating-hypervisors-a-samsung-rkp-case-study-offensivecon-2020/
- https://blog.longterm.io/samsung_rkp.html

Conclusions for defenders. What could have prevented these bugs?

- SECCMD memcpy in S-Boot
  - Static analysis: possibly, but not guaranteed
    - Need to manually annotate taint sources
  - Fuzzing: possibly, but quite costly
    - Need to port S-Boot to run on Linux desktop
    - Need a good dictionary of inputs
  - Using memory safe language: yes

Conclusions for defenders. What could have prevented these bugs?

- “drawimg” heap OOB write in S-Boot
  - Static analysis: same. Possibly, but not guaranteed
  - Fuzzing: same. Possibly, but quite costly
  - Code review: maybe. However, it got exploitable due to refactoring
  - Using memory safe language: not sure
    - JPEG parser is a C library
    - Easy to introduce a bug at the FFI boundary between safe and unsafe

Conclusions for defenders. What could have prevented these bugs?

- RKP/UH issues
  - Static analysis: unlikely
  - Fuzzing: not with a random one, needs some domain knowledge
    - Existing research has not identified this particular bug
  - Memory safe language: probably not
    - The bug is related to MMU and a different address space
    - Most languages don’t model system with such precision
  - Enabling ASLR and stack cookies wouldn’t hurt though

Conclusions for defenders

- Android userland and Linux got more secure with the use SELinux
  - Focus is shifting to firmware and boot chain
- Firmware world is lacking in both mitigations and development tools
  - Static and dynamic analysis tools help a bit, but require dev support
- Bugs tend to be in fresh code or the one which is 3 years and older
- Researchers focus on the overall system, not one area of responsibility
- I believe in re-reviewing stuff randomly from time to time
  - These last two points are harder to formalise in a corporate env

Recap. Where do security issues happen?

- In newly-developed code that has not been reviewed by attackers
- Where someone has already looked but that was 3 years ago
- Where analysis tools/languages don't model the system at the required detail level (MMU, hardware component interaction)

Research tips

- You need to dump Firmware from the device
  - Firmware is encrypted
    - Of all phone/SoC vendors, only two encrypt most of FW
  - You need an existing bug to dump it
    - Most of it is not encrypted and can be downloaded online
- You need a fancy bug and a working exploit to participate in bug bounty
  - A bug is needed, but don’t be shy if it’s not too fancy

Bug bounty tips

- Submit early to allow the company to root cause and fix the bug ASAP
- Submit follow-up info for a better reward
- A good PoC increases payout a few times
  - at least with some vendors