Arms Race
The story of (in)-secure bootloaders

Lee Harrison,  Kang Li
Mobile Device Ecosystem

- Players
  - Device Manufacturers (Samsung, Apple, Qualcomm)
  - Service Providers (AT&T, Verizon)
  - Device “Owners”
The Need to Study Bootloaders

• Break the Lockdown
  ○ Do not have root access
    • Installing unofficial apps
  ○ Can not run non-stock kernels
    • Loading customized kernels
  ○ Can not replace boot loaders
    • Helping break the layers above
What this Talk is about

- Multiple case studies of

  - Do not have root access
  - Installing unofficial apps
    - Can not run non-stock kernel
    - Loading customized kernels
    - Can not replace bootloaders
  - Replacing bootloaders
Disclaimers

- No newly discovered bootloader flaws.
- All bugs have been patched.
- All bugs were found independently by us.
  - one was disclosed by others first.
- Then why make this talk?
  - Bootloader security is unique (and fun).
Why Make This Talk?

- Unique aspects of bootloader security
  - Severe time-pressure on product delivery
  - Threat source
    - From device owners
    - Physical access
  - Some mistakes found were unconventional
Boot Flow Overview

Power on, execute ROM

Load code from flash

Verify and execute code

Kernel

Sometimes many stages of code must be loaded
Bootloader Image Verification (simplified)

Flash
- Header
- Stage N+1 Bootloader Image
- Signature

Memory
- Stage-N Bootloader
- Public Key

Hash1 == Hash2
Case #1

**Samsung Galaxy S3**

- Released in mid-2012
- Based on the Qualcomm MSM8960 chipset

http://static.trustedreviews.com/94/000023989/3659/s3fronts.jpg
Goal: Boot a customized kernel image
S3 Boot Flow

BootROM (PBL) → SBL1 → SBL2 → SBL3 → ABOOT → Kernel

- Resource Power Manager
- TrustZone Kernel
Kernel image is loaded twice by ABOOT

First load the code for execution

Second load for verification
for (p in partitions) {
    if (strcmp(p.name, "boot") == 0) {
        load_partition_to_memory(p)
        break
    }
}

for (p in partitions) {
    if (stricmp(p.name, "boot") == 0) {
        if (load_and_verify_signature(p))
            boot_kernel()
        else show_error()
    }
}
for (p in partitions) {
    if (strcmp(p.name, "boot") == 0) {
        load_partition_to_memory(p)
        break
    }
}

for (p in partitions) {
    if (stricmp(p.name, "boot") == 0) {
        if (load_and_verify_signature(p))
            boot_kernel()
        else show_error()
    }
}
for (p in partitions) {
    if (strcpy(p.name, "boot") == 0) {
        load_partition_to_memory(p)
        break
    }
}

for (p in partitions) {
    if (stricmp(p.name, "boot") == 0) {
        if (load_and_verify_signature(p))
            boot_kernel()
        else show_error()
    }
}
Exploit:
Edit the GPT and add a “BOOT” partition before the “boot” partition partition.
Patched when S3 was updated to Android 4.1.

They altered the signature verification to only load the kernel image once.
Case #2

Samsung Galaxy Note II

Released in Fall 2012
Based on Samsung’s Exynos 4 Quad SoC
Goal: Boot a customized kernel image
Exynos 4 (Note II) Boot Flow

BootROM → BL1 → BL2 → BL3 → Kernel

TrustZone Kernel
// flash_read(mem_dst, disk_src, size)

flash_read(&boot_header, ptn_start, HEADER_SIZE);
if (memcmp(boot_header.magic, "ANDROID!") == 0) {
    // load kernel to 0x40008000, verify it
    // copy ramdisk to 0x42000000
}

Later...
jump to 0x40008000
flash_read(&boot_header, ptn_start, HEADER_SIZE);
if (memcmp(boot_header.magic, "ANDROID!")) == 0) {
    // load kernel to 0x40008000, verify it
    // copy ramdisk to 0x42000000
} else {
    flash_read(0x40008000, ptn_start, 0x800000);  
}
Later...
jump to 0x40008000
flash_read(&boot_header, ptn_start, HEADER_SIZE);
if (memcmp(boot_header.magic, "ANDROID!")) == 0) {
    // load kernel to 0x40008000, verify it
    // copy ramdisk to 0x42000000
} else {
    flash_read(0x40008000, ptn_start, 0x800000);
}
Later...
jump to 0x40008000
A small victory!

- That “else” branch lets us run unsigned code
- Custom kernel booting was made possible
A small victory!

- That “else” branch lets us run unsigned code
- Custom kernel booting was made possible
- But … the load action is very primitive

Need to develop a customized kernel loader
Case #2.1

Samsung Galaxy Note II

Can we modify the early bootloader stages?

Goal: to simplify custom kernel loading
Changing the bootloader

**Problem**: Bootloader is signed

Any modification breaks the signature

How to solve this?
BL1 Verification

- BL1 is verified by BootROM
- BootROM uses a fixed key inside the CPU
- How can this help?
BL1 Verification

- BL1 is verified by BootROM
- BootROM uses a *fixed key* inside the CPU.
- The same key was used across all CPUs of the same model.
Jackpot

- The ODROID-X hobby board shipped with a BL1 that didn't check the signature of the second stage (BL2).
Make a Custom Bootloader

- If we glue their BL1 with a patched BL2/3 then we can boot a custom kernel.

- Custom bootloader created!

The BootROM is still able to validate the BL1, and then the custom BL2/3 will run without checking the signature of future stages.
Another Challenge

- Bootloader partition is not exposed
  - Can’t read or write bootloader from Android
Bootloader on the eMMC

Bootloader is on the protected Boot 1 partition

Android can only see the blue partition

http://www.ureach-inc.com/global/Products-Flash-emmc.html
Writing the Bootloader

- How can we overwrite the Boot 1 partition?
Writing the Bootloader

● How can we overwrite the Boot 1 partition?

Update from Vendor requires flashing BL !!

Phone has a special “download mode” that allows flashing files to the eMMC.
One more problem!

Download mode checks file signatures when flashing.
So close...
How to disable the checking during flashing

The “else” branch!

We can write code to modify the signature check function in the running bootloader, and then load it through the else branch.

This lets us **flash the custom bootloader** to the device

Claim victory again!
Case #2.2
Samsung Galaxy Note II

- Bad News
  - The else branch was removed in firmware update
  - Added blacklisting of old bootloaders
We need a new mechanism for putting the custom bootloader on the eMMC Boot partition

Let’s investigate the firmware update mechanism
Flash Signature Checking

- What kind of checks are performed?
  - RSA-2048

- Under what condition is the check performed?
  - Partitions with certain names, such as “BOOT”

- Where does it get the partition information?
Partition Information Table

- Secondary partition table
- Additional partition metadata
  - id, filesystem
- Root user can edit this table
PIT Header

PIT Entry

part_id: 11
fs_type: raw

PIT

Pit Header

BOOT entry

RECOVERY entry

Flash

GPT

PIT

BOOT

RECOVERY
Bypassing the Flash Check

- Create a duplicate PIT entry for the bootloader partition

- Change the partition name to avoid checking

- The entries would point to the same location on the eMMC
Original PIT

- Pit Header
- BOOT entry
- BOOT data
Custom PIT

- Pit Header
- BOOT entry
- MYBOOT entry
- BOOT data
Custom PIT

Sig-checked during flash

Pit Header

BOOT entry

MYBOOT entry

Not sig-checked during flash

BOOT data
Success

- This was effective in bypassing the signature check.

- The custom bootloader from Case 2.1 was now flashable.
Case #2.3

Note II Lightning Round

Samsung added a signature check on the PIT area at boot time in a new bootloader revision.
• The PIT area could no longer be altered to add custom entries.
But... downgrade works

➔ ...the developers forgot to blacklist the previous bootloader file.
Case #3
Samsung Galaxy S4

- Released in April 2013
- Based on the Qualcomm Snapdragon 600
- Some service providers also requested to lock down their version of devices.
Goal: Boot a custom kernel
S4 Boot Flow

BootROM (PBL) → SBL1 → SBL2 → SBL3 → ABOOT → Kernel

- Resource Power Manager
- TrustZone Kernel
Execution reaches here
The diagram illustrates the process of reading data from a Flash memory to RAM. The Flash memory contains layers labeled as 'Hdr', 'Kernel_IMG', and 'Ramdisk_IMG'. These layers are read by a function called 'mmc_read' and the data is transferred to RAM. The RAM contains a header ('Hdr'), a kernel image ('Kernel_IMG'), a ramdisk image ('Ramdisk_IMG'), and an ABOOT section. The memory address 0x0 is at the top of RAM, and the address 0xFFFFFFFF is at the bottom, indicating the end of the RAM area.
How Sections Load Into Memory

// mmc_read(disk_src, mem_dst, size)

mmc_read( ptn_start, &hdr, PAGE_SIZE );
mmc_read( kernel_start, hdr.kernel_addr, hdr.ksize );
mmc_read( ramdisk_start, hdr.rd_addr, hdr.rdsise );
What can go wrong?
Who controls hdr?

// Load sections into memory
mmc_read( ptn_start, &hdr, PAGE_SIZE );
mmc_read( kernel_start, hdr.kernel_addr, hdr.ksize );
mmc_read( ramdisk_start, hdr.rd_addr, hdr.rdszie );
Who controls hdr?

// Load sections into memory
mmc_read( ptn_start, &hdr, PAGE_SIZE );
mmc_read( kernel_start, hdr.kernel_addr, hdr.ksize );
mmc_read( ramdisk_start, hdr.rd_addr, hdr.rdsise );

User controlled values
Spot the problem?

- No input validation
- Able to load code anywhere
- Even on top of the bootloader
Exploit

- Second mmc_read loads a combined kernel + ramdisk
- Third mmc_read overwrites the signature checking function
- First published by Dan Rosenberg
- Affected many other vendors (e.g. LG)
First mmc_read

Flash

- Hdr
- Kernel_IMG
- Ramdisk_IMG

Modified Flash

- Hdr
- Kernel_IMG
- Ramdisk_IMG
- Patch Code

RAM

- Hdr
- ABOOT

Address:
- 0x0
- 0xFFFFFFFF
Second mmc_read

Flash

- Hdr
- Kernel_IMG
- Ramdisk_IMG

Modified Flash

- Hdr
- Kernel_IMG
- Ramdisk_IMG
- Patch Code

RAM

- Hdr
- Kernel_IMG
- Ramdisk_IMG
- ABOOT

0x0

0xFFFFFFFF
Third mmc_read

Flash

- Hdr
- Kernel_IMG
- Ramdisk_IMG

Modified Flash

- Hdr
- Kernel_IMG
- Ramdisk_IMG
- Patch Code

RAM

- Hdr
- Kernel_IMG
- Ramdisk_IMG
- Patch Code

Hex values:
- 0x0
- 0xFFFFFFFF
Summary

- Six bootloader software errors
  - Case sensitive vs insensitive search
  - Additional debugging (else) branch
  - Failed to revocation signed modules
  - Forgot to blacklist old bootloader
  - Missing integrity check to PIT table
  - Failed to validate the image header

- Most of them are NOT conventional memory vulnerabilities.
Problems are not limited to a single vendor.
  ○ Samsung is very responsive to troubleshoot.
  ○ All bugs shown are patched in Samsung devices.
  ○ Some bugs are common across vendors.

Vendors are often open to allow loading customized kernels.

The “distrust” comes from service providers.
Thanks!

lee2704 @uga.edu
kangli @uga.edu