BootStomp: On the Security of Bootloaders in Mobile Devices

Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi, Eric Gustafson, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna

University of California, Santa Barbara

Presented at USENIX 2017
What is a Bootloader?
What is a Bootloader?

Software module which:

- Initializes the device and its peripherals
- Loads the kernel code from secondary storage
- Jumps to it
We focused on Android bootloaders
Android Bootloaders Overview

- No standard (e.g., ARM gives guidelines)
- Booting through several stages
- Protect integrity of user's device and data:
  - Trusted boot
- Bootloader unlocking
Why attacking bootloaders?
Attacking Bootloaders

An attacker controlling the bootloader might:

- Boot custom Android OS (*bootloader unlocking*)
  - Persistent rootkit
- Brick the device
- In some cases, achieve controls over peripherals
Safety Properties

Integrity of the booting process

- Android OS is verifiably to be in a non-tampered state
- A root process cannot interfere with peripherals setup

Unlocking security mechanism

- A root process cannot unlock the bootloader
- Physical attacker cannot unlock the bootloader
Threat Model
Threat Model

- Attacker has control over the Android OS
  - Root privileges
Threat Model

- Attacker has control over the Android OS
  - Root privileges
- If an attacker has root privileges is game over, why even bother?
  - The safety properties should hold anyway
Outline

- Booting Process
- Bootloader Unlocking
- BootStomp
- Evaluation
- Mitigations
- Conclusions
Outline

- Booting Process
- Bootloader Unlocking
- BootStomp
- Evaluation
- Mitigations
- Conclusions
Booting Process

God mode  Kernel mode  User mode

EL3 | EL1 | EL0
Booting Process

God mode

Kernel mode

User mode

---

BL1/BootROM

BL2

BL31

Load and Verify
Booting Process

God mode                                     Kernel mode                     User mode

BL1/BootROM

BL2

BL31

EL3 | EL1

EL0

Secure World

Non-Secure World

Load and Verify
Boot Process

God mode

Kernel mode

User mode

Peripheral Firmware (radio)

 Trusted OS (tz)

 Trusted Apps

Secure World

Non-Secure World

Load and Verify
# Booting Process

**God mode**

**Kernel mode**

- Peripheral Firmware (radio)
- Trusted OS (tz)
- Android Kernel (boot)

**User mode**

- Trusted Apps
- Android Framework/Apps (system/data)

---

**Load and Verify**

**Secure World**

**Non-Secure World**

**: EL3 EL1 EL0**
Booting Process

God mode

BL1/BootROM

BL2

BL31

BL33 (aboot)

Kernel mode

Peripheral Firmware (radio)

Trusted OS (tz)

Android Kernel (boot)

User mode

Trusted Apps

Android Framework/Apps (system/data)

Bootloader
Booting Process

- **God mode**
- **Kernel mode**
- **User mode**

*Android OS*
Booting Process

God mode

Kernel mode

User mode

Chain of trust
Booting Process

God mode

- BL1/BootROM
- BL2
- BL31

Kernel mode

- Peripheral Firmware (radio)
- Trusted OS (tz)
- BL33 (aboot)
- Android Kernel (boot)

User mode

- Trusted Apps
- Android Framework/Apps (system/data)

Chain of trust
Outline

- Booting Process
- **Bootloader Unlocking**
- BootStomp
- Evaluation
- Mitigations
- Conclusions
Bootloader Unlocking

Two steps
Bootloader Unlocking

Against an attacker with physical access

Against root process
Bootloader Unlocking

The unlocking state (*device’s security state*) saved on persistent storage

- It should be writable only by high privileged components (e.g., bootloader or secure OS)
Can a compromised Android OS affect the booting process?
Can a compromised Android OS affect the booting process?

Yes!
Bootloader

Read

Load

Persistent Storage

Android OS
We need a tool to automatically verify the safety properties
Towards a Bootloader Analyzer

Bootloaders are hard to analyze:

- The source code is hardly available
Towards a Bootloader Analyzer

Bootloaders are hard to analyze:

- The source code is hardly available → Binary (blob)
Towards a Bootloader Analyzer

Bootloaders are hard to analyze:

- The source code is hardly available → Binary (blob)
- Dynamic execution is impractical
Towards a Bootloader Analyzer

Bootloaders are hard to analyze:

- The source code is hardly available → Binary (blob)
- Dynamic execution is impractical → Hardware is required
Towards a Bootloader Analyzer

Bootloaders are hard to analyze:

- The source code is hardly available → Binary (blob)
- Dynamic execution is impractical → Hardware is required
- Execute before the Android OS
Towards a Bootloader Analyzer

Bootloaders are hard to analyze:

- The source code is hardly available → Binary (blob)
- Dynamic execution is impractical → Hardware is required
- Execute before the Android OS → Known library/syscall are not in use
Towards a Bootloader Analyzer

Bootloaders are hard to analyze:

- The source code is hardly available → Binary (blob)
- Dynamic execution is impractical → Hardware is required
- Execute before the Android OS → Known library/syscall are not in use
  - There is no memcpy!
Outline

● Booting Process

● Unlocking Mechanism

● **BootStomp**

● Evaluation

● Mitigations

● Conclusions
BootStomp: A Bootloader Analyzer

Automatic static binary tool for finding security vulnerabilities in bootloaders
BootStomp: A Bootloader Analyzer

Automatic static binary tool for finding security vulnerabilities in bootloaders

- Determine whether **attacker-controlled data** can influence the bootloader intended behavior

- Traceable output
  - Verify generated alerts
BootStomp: A Bootloader Analyzer

BootStomp uses multi-tag taint analysis based on under-constrained dynamic symbolic execution
BootStomp: A Bootloader Analyzer

BootStomp uses multi-tag taint analysis based on under-constrained dynamic symbolic execution

- It uses a **fully symbolic** taint analysis engine to trace attacker-controlled data
BootStomp: A Bootloader Analyzer

BootStomp uses multi-tag taint analysis based on under-constrained dynamic symbolic execution

- Arbitrary memory writes
- Arbitrary memory reads
- Attacker can control loops iterations
- Bypass unlocking mechanism
  - Functions overwriting the security state on persistent storage
BootStomp: A Bootloader Analyzer

Mobile Device → Mobile Bootloader (Binary BLOB) → Manual Extraction → BootStomp

BootStomp: CFG → Hex-Rays Decompiler → CFG → Seed Finder Module → Sink Finder Module → Seed Analysis Engine → angr

Alerts
BootStomp: A Bootloader Analyzer

BootStomp uses multi-tag taint analysis based on under-constrained dynamic symbolic execution

- Seeds of taint
- Taint propagation and removal
- Sinks of taint
- Taint checking
BootStomp: Seeds of Taint

- Data read from persistent storage
- Data used by the unlocking procedure
BootStomp: Seeds of Taint

- Data read from persistent storage
- Data used by the unlocking procedure

BootStomp must find these functions
BootStomp: Seeds of Taint

Automatic detection of functions:

- Identify the functions based on the “log” strings
- Analysis to identify the arguments to taint

```c
LABEL_5:  
v16 = sub_7002D00(a3, v11, v12, 1i64); // emmc read function

t = v16 << 9;

v23 = v16;

  sub_70032BC(0xB3u, 8u);
result = 0164;

if ( v23 )
{
  sub_705F924(__int64)"emmc read error = "\n", v23, v17, v18, v19, v20, v21, v22, v45); // logging function 1
  sub_7001F04(__int64)"emmc read error = "\n", v23, v39, v40, v41, v42, v43, v44, v45); // logging function 2
  result = 0xFFFFFFFF64;
}
return result;
```
BootStomp: Seeds of Taint

Optionally, provided by the security analyst

• Useful for finding the unlocking function
  ○ Several do not contain log messages
BootStomp: Taint Propagation and Removal

- Taints are symbolic expressions encoding how the value is computed

- Propagated and removed implicitly during the dynamic symbolic execution traversal
BootStomp: Taint Propagation and Removal

**Code**

```
ty = seed_func();
x = ty + 5;
....
x = 0xdeadbeef;
```

**Symbolic expressions**

```
ty = TAINT_ty
x = TAINT_ty + 5
....
x = 0xdeadbeef
```
BootStomp: Sinks of Taint

- Memcpy-like functions
- Dereference of a tainted variable
- Comparisons of tainted variables in loops’ conditions
- Write to a persistent storage of a tainted variable
BootStomp: Sinks of Taint

- Memcpy-like functions
  - Small functions with loop copying data between two buffers
  - Many callers (a threshold is used)
- Dereference of a tainted variable
- Comparisons of tainted variables in loops’ conditions
- Write to a persistent storage of a tainted variable
BootStomp: Sinks of Taint

- Memcpy-like functions
- Dereference of a tainted variable
- Comparisons of tainted variables in loops’ conditions
- Write to a persistent storage of a tainted variable
BootStomp: Taint Checking

- An alert is raised when a tainted variable:
  - Reaches a memcpy-like function
  - Gets dereferenced
  - Can control the number of iterations of a loop
  - Gets written to a persistent storage
BootStomp: Taint Checking

- An alert is raised when a tainted variable:
  - Reaches a memcpy-like function
  - Gets dereferenced
  - Can control the number of iterations of a loop
  - Gets written to a persistent storage

- A traceable output is produced
Limitation: Path Explosion Problem

- Limited function traversal
Limitation: Path Explosion Problem

- Limited function traversal
  - Tainted arguments and call stack size < threshold?
Limitation: Path Explosion Problem

- Limited function traversal
  - Tainted arguments and call stack size < threshold?
    - Yes → step into
    - No → step over
Limitation: Path Explosion Problem

- Limited function traversal
- Limited loop iterations
Limitation: Path Explosion Problem

- Limited function traversal

- Limited loop iterations
  - Threshold used
Limitation: Path Explosion Problem

- Limited function traversal
- Limited loop iterations
- Timeout
Outline

- Booting Process
- Unlocking Mechanism
- BootStomp
- Evaluation
- Mitigations
- Conclusions
BootStomp has been evaluated against 4 different bootloader
## Evaluation: Bugs

<table>
<thead>
<tr>
<th>Bootloader</th>
<th>Total Alerts</th>
<th>Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qualcomm (Latest)</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Qualcomm (Old)</td>
<td>8</td>
<td>1 (already known)</td>
</tr>
<tr>
<td>NVIDIA</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>HiSilicon</td>
<td>17</td>
<td>5</td>
</tr>
<tr>
<td>MediaTek</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>36</strong></td>
<td><strong>7 (6 0days)</strong></td>
</tr>
</tbody>
</table>

(Further details in the paper)
Ok good, but how bad are them?
Evaluation: Bugs
Evaluation: Bugs
Evaluation: Bugs

Great, but what can you do with it?
Evaluation: Bugs

Great, but what can you do with it?

- A lot! Example: some bootloaders work in EL3
## Evaluation: Unlocking Bypass

<table>
<thead>
<tr>
<th>Bootloader</th>
<th>Writes to flash?</th>
<th>Potentially vulnerable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qualcomm (Latest)</td>
<td>6</td>
<td>YES*</td>
</tr>
<tr>
<td>Qualcomm (Old)</td>
<td>4</td>
<td>YES*</td>
</tr>
<tr>
<td>NVIDIA</td>
<td>9</td>
<td>NO</td>
</tr>
<tr>
<td>HiSilicon</td>
<td>17</td>
<td>YES*</td>
</tr>
<tr>
<td>MediaTek</td>
<td>1</td>
<td>NO</td>
</tr>
</tbody>
</table>

*(Yes means BootStomp found a write to a persistent storage)*
Bootloader Unlocking Bypass

```c
memcpy(&expected_digest, &from_flash, 32);
compute_sha(oem_key, input_len, &key_digest);
if (memcmp(&key_digest, &expected_digest, 32)) {
    // Log the result
    return 1;
}

hash_func("bonaciao", &key_digest, &hash_output);
if (write_to_flash(hash_output, 16) & 0x80000000) {
    // Log the result
    return 0;
}
```
Overview

- Booting Process
- Unlocking Mechanism
- BootStomp
- Evaluation
- Mitigations
- Conclusions
Mitigations

- Google approach
  - The key used to encrypt/decrypt user data contains the security state (locked/unlock)
Mitigations

- **Google approach**
  - The key used to encrypt/decrypt user data contains the security state (locked/unlock)
    - If the state changes, the key changes → user’s data cannot be decrypted
Mitigations

● Google approach

  ○ The key used to encrypt/decrypt user data contains the security state (locked/unlock)
    ■ If the state changes, the key changes → user’s data cannot be decrypted

● Our proposal

  ○ Security state stored in the eMMC’s *Replay Protected Memory Block* (RPMB)
Mitigations

● Google approach

  ○ The key used to encrypt/decrypt user data contains the security state (locked/unlock)

    ■ If the state changes, the key changes → user’s data cannot be decrypted

● Our proposal

  ○ Security state stored in the eMMC’s Replay Protected Memory Block (RPMB)

    ■ Modify the trusted OS to allow only the bootloader to modify it
Outline

- Booting Process
- Unlocking Mechanism
- BootStomp
- Evaluation
- Mitigations
- Conclusions
Responsible Disclosure

All bugs reported, acknowledged and already fixed
Conclusions

- First study to explore Android bootloaders
- Automated technique to analyze bootloaders with traceable alerts
- Found 6 zero days in various bootloaders

https://github.com/ucsb-seclab/bootstomp
That’s All

Questions?
Buffer overflow

// oem_get_info function

oem_read(block, block_len);
buf = malloc(block[0]); // size block

// .. additional code ..
number_or_blocks = block[1];
block_id = block[2];

if (number_of_blocks == 1 || block_id == number_of_blocks) {
    return;
}

memcpy(buf + off, block[3], 0x300);
Buffer overflow

// oem_get_info function

oem_read(block, block_len);
buf = malloc(block[0]); // size block

// .. additional code ..
number_or_blocks = block[1];
block_id = block[2];

if (number_of_blocks == 1 || block_id == number_of_blocks) {
    return;
}

memcpy(buf + off, block[3], 0x300);
Buffer overflow

// oem_get_info function

oem_read(block, block_len);
buf = malloc(block[0]); // size block

// . additional code ..
number_or_blocks = block[1];
block_id = block[2];

if (number_of_blocks == 1 || block_id == number_of_blocks) {
    return;
}

memcpy(buf + off, block[3], 0x300);
Buffer overflow

// oem_get_info function

oem_read(block, block_len);
buf = malloc(block[0]); // size block

// .. additional code ..
number_or_blocks = block[1];
block_id = block[2];

if (number_of_blocks == 1 || block_id == number_of_blocks) {
  return;
}

memcpy(buf + off, block[3], 0x300); // buffer overflow!
If the bootloader only loads the Android O.S., how can an attacker harm the device?
If the bootloader only loads the Android O.S., how can an attacker harm the device?

Bootloaders are very diverse
BL33 in practice

Qualcomm and NVIDIA’s:

- BL33 conforms very closely to Google’s Verified Boot guidelines,
- BL33 runs in EL1
BL33 in practice

Qualcomm and NVIDIA’s Huawei HiSilicon:

- BL33 is also responsible for initializing modem and peripherals
- BL33 runs in EL3.
BL33 in practice

Qualcomm and NVIDIA’s

Huawei HiSilicon

MediaTek:

- BL33 is also responsible for initializing modem
- BL33 runs in EL1