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

















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!



We need a tool to automatically verify the safety properties

Bootloaders are hard to analyze:

• The source code is hardly available

Bootloaders are hard to analyze:

• The source code is hardly available \rightarrow Binary (blob)

- The source code is hardly available \rightarrow Binary (blob)
- Dynamic execution is impractical

- The source code is hardly available \rightarrow Binary (blob)
- Dynamic execution is impractical \rightarrow Hardware is required

- The source code is hardly available \rightarrow Binary (blob)
- Dynamic execution is impractical \rightarrow Hardware is required
- Execute before the Android OS

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

Bootloaders are hard to analyze:

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

Outline

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

Automatic static binary tool for finding security vulnerabilities in bootloaders

Automatic static binary tool for finding security vulnerabilities in bootloaders

- Determine whether **<u>attacker-controlled data</u>** can influence the bootloader intended behavior
- Traceable output
 - Verify generated alerts

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

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

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

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

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

BootStomp must find these functions

Automatic detection of functions:

- Identify the functions based on the "log" strings
- Analysis to identify the arguments to 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





Memory

ty

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
 - \circ $\,$ Can control the number of iterations of a loop
 - Gets written to a persistent storage
- A traceable output is produced

• Limited function traversal

- Limited function traversal
 - Tainted arguments and call stack size < threshold ?

- Limited function traversal
 - Tainted arguments and call stack size < threshold ?
 - Yes \rightarrow step into
 - $\bullet \quad No \quad \rightarrow step \ over$

- Limited function traversal
- Limited loop iterations

- Limited function traversal
- Limited loop iterations
 - Threshold used

- Limited function traversal
- Limited loop iterations
- Timeout

Outline

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

BootStomp has been evaluated against 4 different bootloaders

Bootloader	Total Alerts	Bugs
Qualcomm (Latest)	4	0
Qualcomm (Old)	8	1 (already known)
NVIDIA	7	1
HiSilicon	17	5
MediaTek	_	_
Total	36	7 (6 0days)

(Further details in the paper)

Ok good, but how bad are them?





Great, but what can you do with it?

Great, but what can you do with it?

• A lot! Example: some bootloaders work in EL3

Evaluation: Unlocking Bypass

Bootloader	Writes to flash?	Potentially vulnerable?
Qualcomm (Latest)	6	YES*
Qualcomm (Old)	4	YES*
NVIDIA	9	NO
HiSilicon	17	YES*
MediaTek	1	NO

(Yes means BootStomp found a write to a persistent storage)


Overview

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

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

- Google approach
 - The key used to encrypt/decrypt user data contains the security state (locked/unlock)
 - If the state changes, the key changes \rightarrow user's data cannot be decrypted

- Google approach
 - The key used to encrypt/decrypt user data contains the security state (locked/unlock)
 - If the state changes, the key changes \rightarrow user's data cannot be decrypted
- Our proposal
 - Security state stored in the eMMC's Replay Protected Memory Block (RPMB)

- Google approach
 - The key used to encrypt/decrypt user data contains the security state (locked/unlock)
 - If the state changes, the key changes \rightarrow 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

✓ <u>https://github.com/ucsb-seclab/bootstomp</u>

That's All

Questions?



```
// 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);
```

```
// 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);
```

```
// 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);
```

```
// 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