RSAConference2018
San Francisco | April 16 – 20 | Moscone Center

SESSION ID: MBS-R14

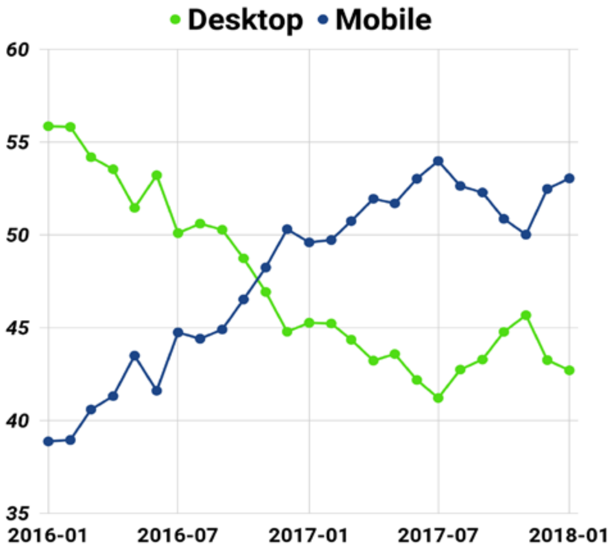
HOW AUTOMATED VULNERABILITY ANALYSIS DISCOVERED HUNDREDS OF ANDROID 0-DAYS

Giovanni Vigna

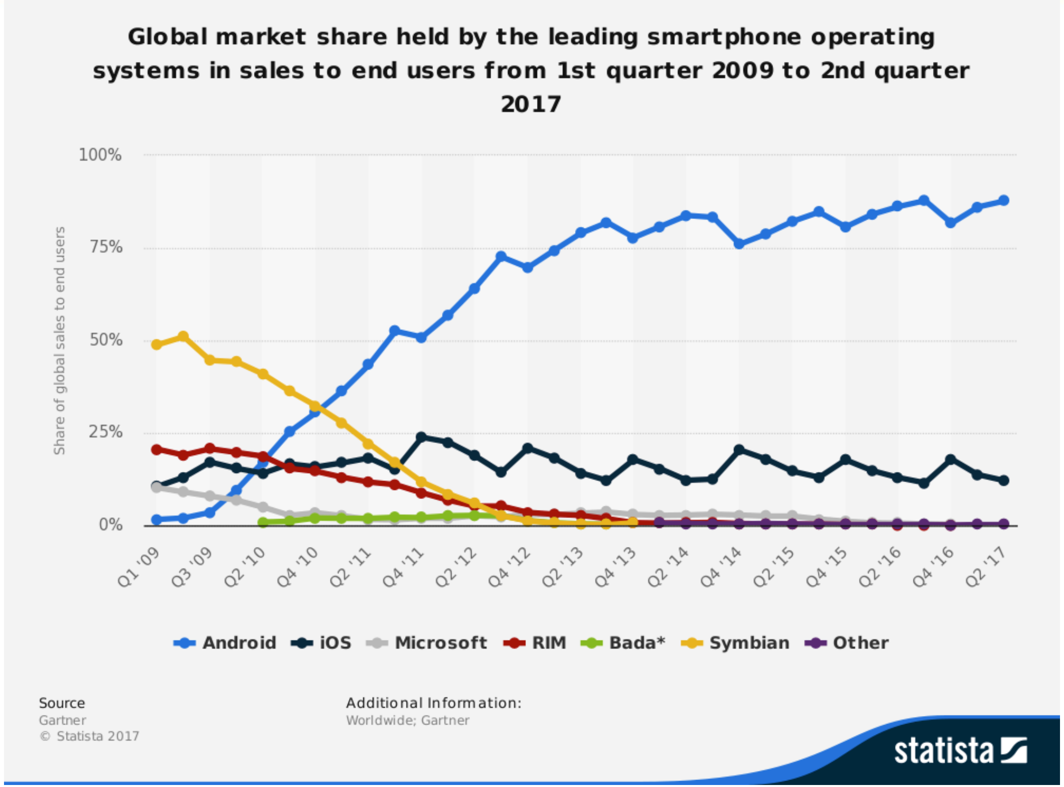
CTO, Lastline, Inc.
http://www.lastline.com

Professor of Computer Science, University of California Santa Barbara
http://www.cs.ucsb.edu/~vigna/

Smartphones Everywhere, All The Time



Why Android?


Global market share held by the leading smartphone operating systems in sales to end users from 1st quarter 2009 to 2nd quarter 2017
100%
75%
50%
25%
0%
Share of global sales to end users
Q1 '09
Q3 '09
Q2 '10
Q4 '10
Q2 '11
Q4 '11
Q2 '12
Q4 '12
Q2 '13
Q4 '13
Q2 '14
Q4 '14
Q2 '15
Q4 '15
Q2 '16
Q4 '16
Q2 '17
Android
iOS
Microsoft
RIM
Bada*
Symbian
Other
Source
Gartner
© Statista 2017
Additional Information:
Worldwide; Gartner
statista

Smartphone Vulnerabilities

Trend Micro Awards $515,000 at Mobile Pwn2Own2017

By: Sean Michael Kerner | November 02, 2017

(0) comments

The longest exploit chain in the history of the Pwn2own competition was demonstrated at the Mobile Pwn2Own 2017 event in Tokyo, with security researchers using 11 different bugs to get code execution on a Samsung Galaxy S8.

The second day of the mobile Pwn2Own hacking contest on Nov. 2 brought with it more exploits, including the longest exploit chain ever seen at a Pwn2own event.

Mobile Pwn2own 2017 ran from Nov.1-2 in Tokyo Japan and resulted in 32 different vulnerabilities being disclosed involving Apple, Samsung and Huawei mobile devices. At the end of the two-day event, Trend Micro's Zero Day Initiative (ZDI) awarded a grand total of $515,000 in prize money for the successfully demonstrated exploits. ZDI has privately disclosed all of the vulnerabilities to the impacted vendors so the issues can be patched.

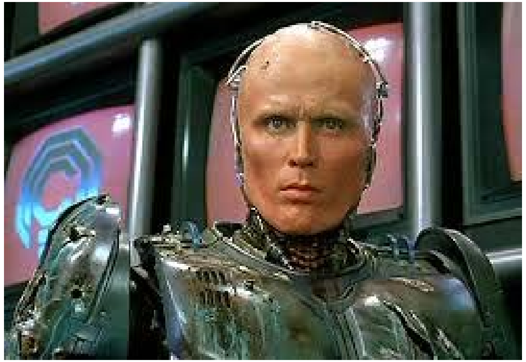
Smartphone Vulnerabilities

Severity	Complete Report* + PoC	Payment range (if report includes an exploit leading to Kernel compromise)**	Payment range (if report includes an exploit leading to TEE compromise)**
Critical	Required	Up to $150,000	Up to $200,000
High	Required	Up to $75,000	Up to $100,000
Moderate	Required	Up to $20,000	Up to $35,000
Low	Required	Up to $330	Up to $330

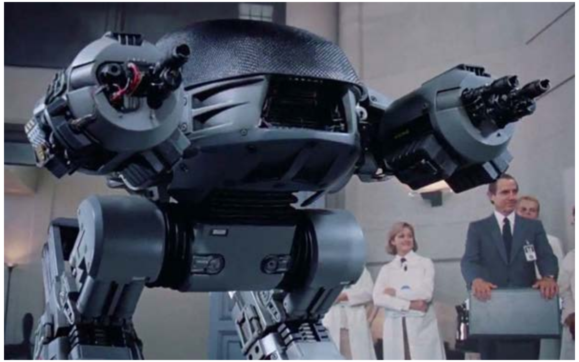
Apps vs. System

Automated Vulnerability Analysis

Human



Semi-Automated



Fully Automated

Many Weapons, Many Targets

Static Analysis

Concolic Execution

Dynamic Analysis

Fuzzing

Taint Analysis

Points-to Analysis

Symbolic Execution

Data Flow Analysis

Operating System

GUIs

Back-end Components

Drivers

Apps

Bootloaders

Trusted Execution Environments

Many Weapons, Many Targets

Static Analysis

Dynamic Analysis

Concolic Execution

Fuzzing

Taint Analysis

Points-to Analysis

Symbolic Execution

Data Flow Analysis

Operating System

GUIs

Back-end Components

Drivers

Apps

Bootloaders

Trusted Execution Environments

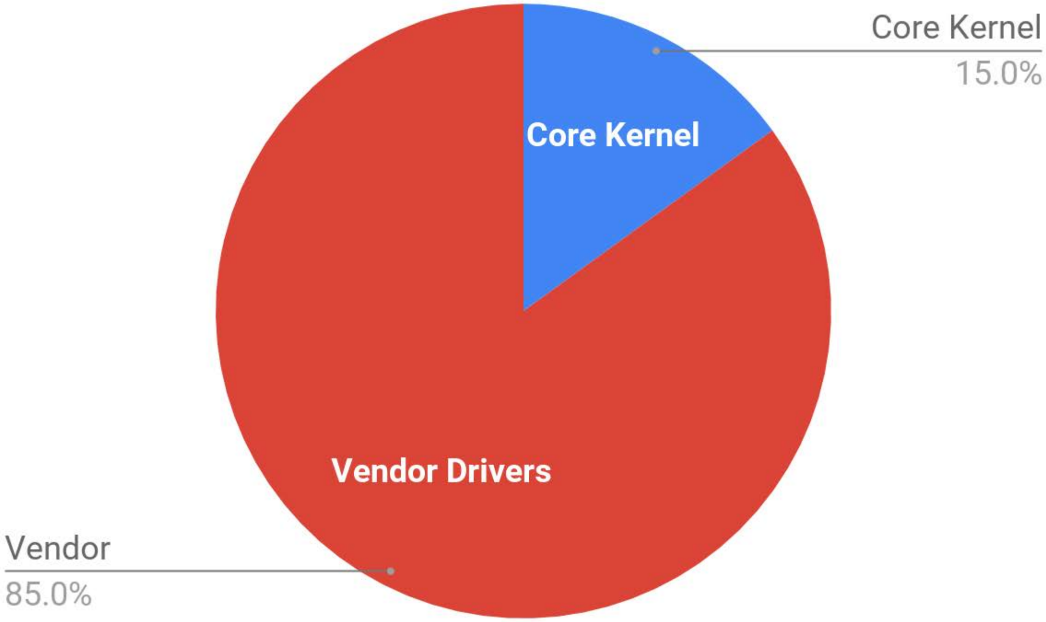
Find Vulnerabilities Before The Bad Guys Do

- Use scalable, automated algorithms to support large-scale analysis
- Share results with vendors to make systems more secure
- Improve the state-of-the-art in vulnerability analysis

Our Work

- Identify vulnerabilities in drivers using points-to and taint analysis
  - Found 158 bugs
- Identify vulnerabilities in drivers using interface-aware fuzzing
  - Found 36 bugs
- Identify vulnerabilities in bootloaders using taint analysis
  - Found 7 bugs

Where Are the Android Kernel Bugs?



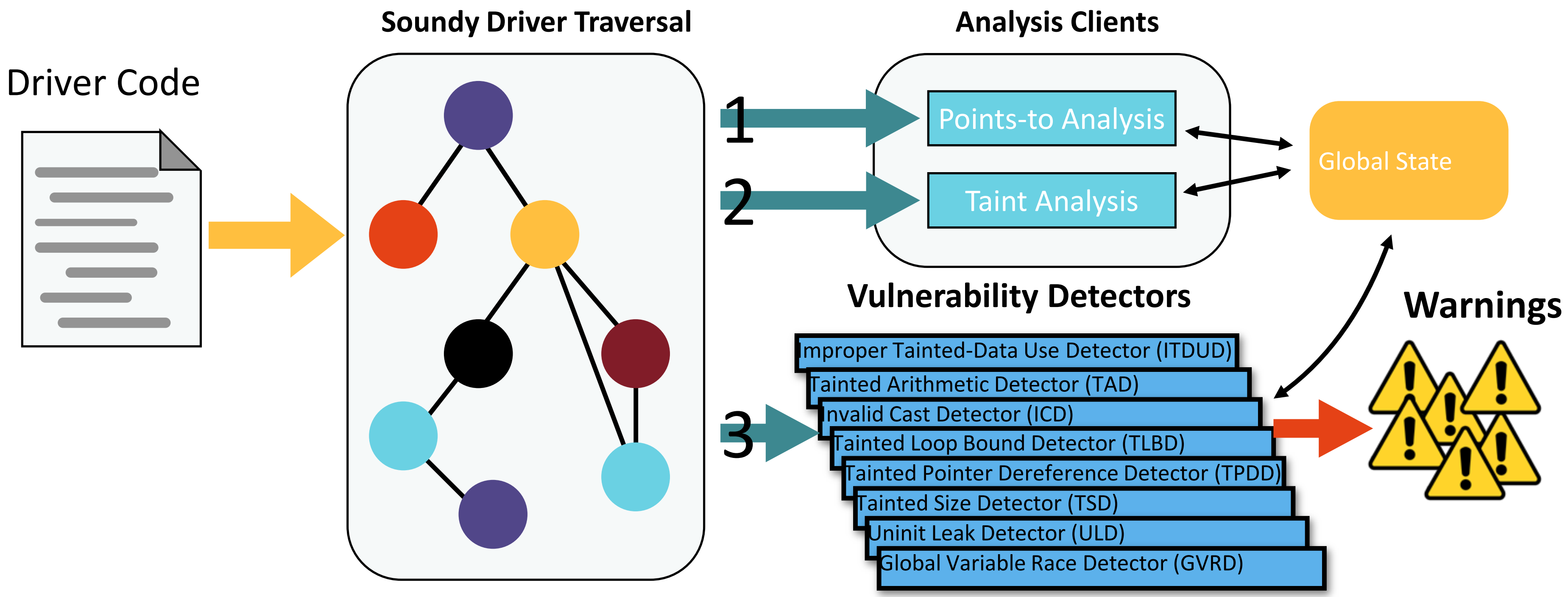
android: Protecting the kernel, Jeff Vander Stoep, Linux Foundation 2016

Program Analyses

- Points-to Analysis: Determines all storage locations that a pointer can point to
  - Example bug: Kernel code pointer to user-controlled memory

- Taint Analysis: Determines all of the locations that are affected by user-supplied (tainted) data
  - Example bug: User provided data used as length in copy_from_user()

Dr. Checker: Design



Soundy Driver Traversal

- **Context-sensitive:** Analysis for each function call is done in the context of the calling function
- **Field-sensitive:** The ability to differentiate between different fields in a memory structure
- **Flow-sensitive:** The ability to track data usage (e.g., taint) throughout a program, according to its control flow

Soundy Driver Traversal


```
struct kernel_obj ko;

void internal_function(int *ptr) {
   *ptr += 1;
}

void entry_point(void *user_ptr, int len) {
        curr_data->item = &ko;

        copy_from_user(&ko, user_ptr, len);

        for (int i = 0; i < ko.count; i++) {
          internal_function(&(ko.data[i]));
        }

        dangerous_function(curr_data->buf);
        dangerous_function(curr_data->item);
        kernel_function(curr_data->item);

}
```

Soundy Driver Traversal

Taint Analysis

user_ptr

len

```
struct kernel_obj ko;

void internal_function(int *ptr) {
  *ptr += 1;
}

void entry_point(void *user_ptr, int len) {
        curr_data->item = &ko;

        copy_from_user(&ko, user_ptr, len);

        for (int i = 0; i < ko.count; i++) {
          internal_function(&(ko.data[i]));
        }

        dangerous_function(curr_data->buf);
        dangerous_function(curr_data->item);
        kernel_function(curr_data->item);

}
```

Soundy Driver Traversal

Taint Analysis

user_ptr

len

```
struct kernel_obj ko;

void internal_function(int *ptr) {
  *ptr += 1;
}

void entry_point(void *user_ptr, int len) {
        curr_data->item = &ko;

        copy_from_user(&ko, user_ptr, len);

        for (int i = 0; i < ko.count; i++) {
          internal_function(&(ko.data[i]));
        }

        dangerous_function(curr_data->buf);
        dangerous_function(curr_data->item);
        kernel_function(curr_data->item);

}
```

Field-sensitive

Soundy Driver Traversal

Taint Analysis

user_ptr

len

ko

curr_data->item

Taint Source

```
struct kernel_obj ko;

void internal_function(int *ptr) {
  *ptr += 1;
}

void entry_point(void *user_ptr, int len) {
        curr_data->item = &ko;

        copy_from_user(&ko, user_ptr, len);

        for (int i = 0; i < ko.count; i++) {
          internal_function(&(ko.data[i]));
        }

        dangerous_function(curr_data->buf);
        dangerous_function(curr_data->item);
        kernel_function(curr_data->item);

}
```

Field-sensitive

Warning: Improper Tainted-Data Use

Soundy Driver Traversal

Taint Analysis

user_ptr

len

ko

curr_data->item

Taint Source

```
struct kernel_obj ko;

void internal_function(int *ptr) {
  *ptr += 1;
}

void entry_point(void *user_ptr, int len) {
        curr_data->item = &ko;

        copy_from_user(&ko, user_ptr, len);

        for (int i = 0; i < ko.count; i++) {
          internal_function(&(ko.data[i]));
        }

        dangerous_function(curr_data->buf);
        dangerous_function(curr_data->item);
        kernel_function(curr_data->item);

}
```

Field-sensitive

Warning: Improper Tainted-Data Use

Warning: Tainted Loop Bound

Soundy Driver Traversal

Taint Analysis

user_ptr

len

ko

curr_data->item

Taint Source

```
struct kernel_obj ko;

void internal_function(int *ptr) {
  *ptr += 1;
}

void entry_point(void *user_ptr, int len) {
      curr_data->item = &ko;

      copy_from_user(&ko, user_ptr, len);

      for (int i = 0; i < ko.count; i++) {
       internal_function(&(ko.data[i]));
      }

      dangerous_function(curr_data->buf);
      dangerous_function(curr_data->item);
      kernel_function(curr_data->item);

}
```

Field-sensitive

Warning: Improper Tainted-Data Use

Warning: Tainted Loop Bound

Soundy Driver Traversal

Taint Analysis

user_ptr

len

ko

curr_data->item

Taint Source

```
struct kernel_obj ko;

void internal_function(int *ptr) {
  *ptr += 1;
}

void entry_point(void *user_ptr, int len) {
        curr_data->item = &ko;

        copy_from_user(&ko, user_ptr, len);

        for (int i = 0; i < ko.count; i++) {
          internal_function(&(ko.data[i]));
        }

        dangerous_function(curr_data->buf);
        dangerous_function(curr_data->item);
        kernel_function(curr_data->item);

}
```

Warning: Tainted Arithmetic

Field-sensitive

Warning: Improper Tainted-Data Use

Warning: Tainted Loop Bound

Soundy Driver Traversal

Taint Analysis

user_ptr

len

ko

curr_data->item

Taint Source

Untainted Field

```
struct kernel_obj ko;

void internal_function(int *ptr) {
  *ptr += 1;
}

void entry_point(void *user_ptr, int len) {
        curr_data->item = &ko;

        copy_from_user(&ko, user_ptr, len);

        for (int i = 0; i < ko.count; i++) {
          internal_function(&(ko.data[i]));
        }

        dangerous_function(curr_data->buf);
        dangerous_function(curr_data->item);
        kernel_function(curr_data->item);

}
```

Warning: Tainted Arithmetic

Field-sensitive

Warning: Improper Tainted-Data Use

Warning: Tainted Loop Bound

Soundy Driver Traversal

Taint Analysis

user_ptr

len

ko

curr_data->item

Taint Source

Untainted Field

```
struct kernel_obj ko;

void internal_function(int *ptr) {
  *ptr += 1;
}

void entry_point(void *user_ptr, int len) {
      curr_data->item = &ko;

      copy_from_user(&ko, user_ptr, len);

      for (int i = 0; i < ko.count; i++) {
        internal_function(&(ko.data[i]));
      }

      dangerous_function(curr_data->buf);
      dangerous_function(curr_data->item);
      kernel_function(curr_data->item);

}
```

Warning: Tainted Arithmetic

Field-sensitive

Warning: Improper Tainted-Data Use

Warning: Tainted Loop Bound

Warning: Improper Tainted-Data Use

Soundy Driver Traversal

Taint Analysis

- user_ptr
- len
- ko
- curr_data->item

Taint Source

Untainted Field

Kernel Functions Ignored

```
struct kernel_obj ko;

void internal_function(int *ptr) {
  *ptr += 1;
}

void entry_point(void *user_ptr, int len) {
      curr_data->item = &ko;

      copy_from_user(&ko, user_ptr, len);

      for (int i = 0; i < ko.count; i++) {
        internal_function(&(ko.data[i]));
      }

      dangerous_function(curr_data->buf);
      dangerous_function(curr_data->item);
      kernel_function(curr_data->item);

}
```

Warning: Tainted Arithmetic

Field-sensitive

Warning: Improper Tainted-Data Use

Warning: Tainted Loop Bound

Warning: Improper Tainted-Data Use

Driver Entry Points

- File Operations
- Attribute Operations
- Socket Operations
- Wrapper Functions

Entry Type	Argument(s)	Taint Type
Read (File)	char *buf, size_t len	Direct
Write (File)	char *buf, size_t len	Direct
Ioctl (File)	long args	Direct
DevStore (Attribute)	const char *buf	Indirect
NetDevIoctl (Socket)	struct *ifreq	Indirect
V4Ioctl	struct v412_format *f	Indirect

Evaluation: Mobile Kernels

MEDIATEK

Amazon Echo (5.5.0.3)

Amazon Fire HD8 (6th Generation, 5.3.2.1)

HTC One Hima (3.10.61-g5f0fe7e)

Sony Xperia XA (33.2.A.3.123)

HUAWEI

Huawei Venus P9 Lite (2016-03-29)

SAMSUNG

Samsung Galaxy S7 Edge (SM-G935F NN)

QUALCOMM

HTC Desire A56 (a56uhl-3.4.0)

LG K8 ACG (AS375)

ASUS Zenfone 2 Laser (ZE550KL / MR5-21.40.1220.1794)

3.1 Million lines of driver code

Dr. Checker Results

Warnings per Kernel *(Count / Confirmed / Bug)*

Detector	Huawei	Qualcomm	Mediatek	Samsung	Total
TaintedSizeDetector	62 / 62/ 5	33 / 33 / 2	155 / 155 / 6	20 / 20 / 1	270 / 268 / 14
TaintedPointerDereferenceChecker	522 / 155 / 12	264 / 264 / 3	465 / 459 / 6	479 / 423 / 4	1,760 / 1,301 / 25
TaintedLoopBoundDetector	75 / 56 / 4	52 / 52 / 0	73 / 73 / 1	78 / 78 / 0	278 / 259 / 5
GlobalVariableRaceDetector	324 / 184 / 38	188 / 108 / 8	548 / 420 / 5	100 / 62 / 12	1,160 / 774 / 63
ImproperTaintedDataUseDetector	81 / 74 / 5	92 / 91 / 3	243 / 241 / 9	135 / 134 / 4	551 / 540 / 21
IntegerOverflowDetector	250 / 177 / 6	196 / 196 / 2	247 / 247 / 6	99 / 87 / 2	792 / 707 / 16
KernelUninitMemoryLeakDetector	9 / 7 / 5	1 / 1 / 0	8 / 5 / 5	6 / 2 / 1	24 / 15 / 11
InvalidCastDetector	96 / 13 / 2	75 / 74 / 1	9 / 9 / 0	56 / 13 / 0	236 / 109 / 3
	1,449 / 728 / 78	**901 / 819 / 19**	**1,748 / 1,607 / 44**	**973 / 819 / 24**	**5,071 / 3,973 / 158**

Precision: 78%

Zero-day Bug (Mediatek's Accdet driver)

#RSAC

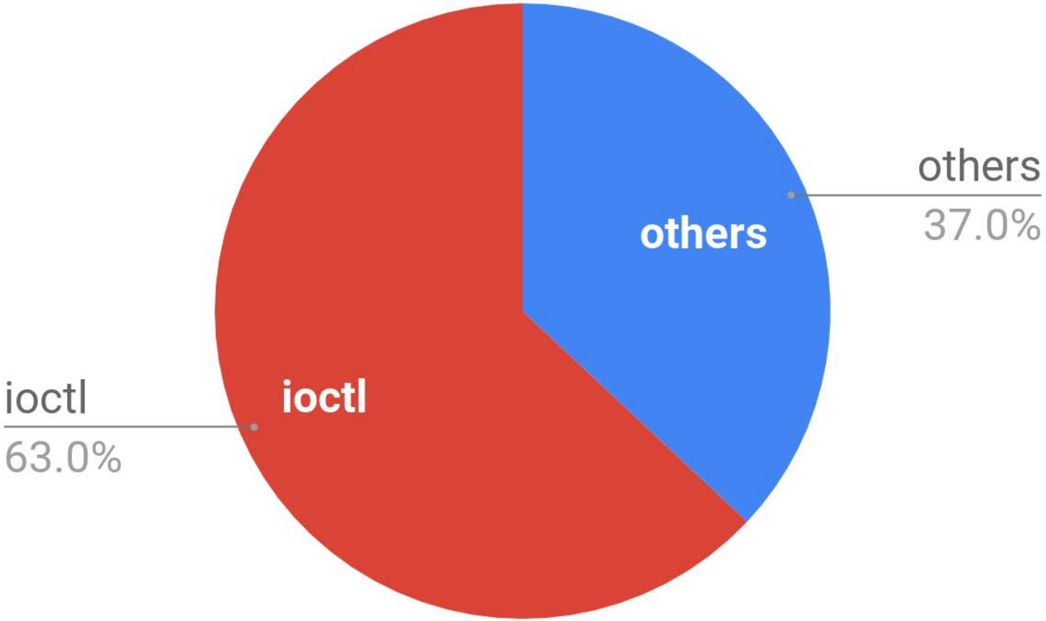
```
static char call status ;
...
static ssize_t accdet_store_call_state( struct device driver *ddri , const char *buf , size t count) {

  int ret = sscanf(buf, ”%s”, &call status);

  if (ret != 1) {
    ACCDETDEBUG(”accdet: Invalid values\n”);
    return -EINVAL;
  }
  ...
}
```

buf can contain more than one char !

Warning: Improper Tainted-Data Use

ret is checked, but it's too late

How Are Kernel Bugs Reached from User Space?



android: Protecting the kernel, Jeff Vander Stoep, Linux Foundation 2016

ioctl

- **I**nput/**O**utput **C**on**t**ro**l**

- System call to allow device operations that can't be well modeled as a "normal" system call

- Bound to a file, requires a valid file descriptor

ioctl

```
ioctl(
    int fd,
    unsigned long command,
    unsigned long param
);
```

ioctl


```
ioctl(
    int fd,
    unsigned long command,
    unsigned long param
);
```

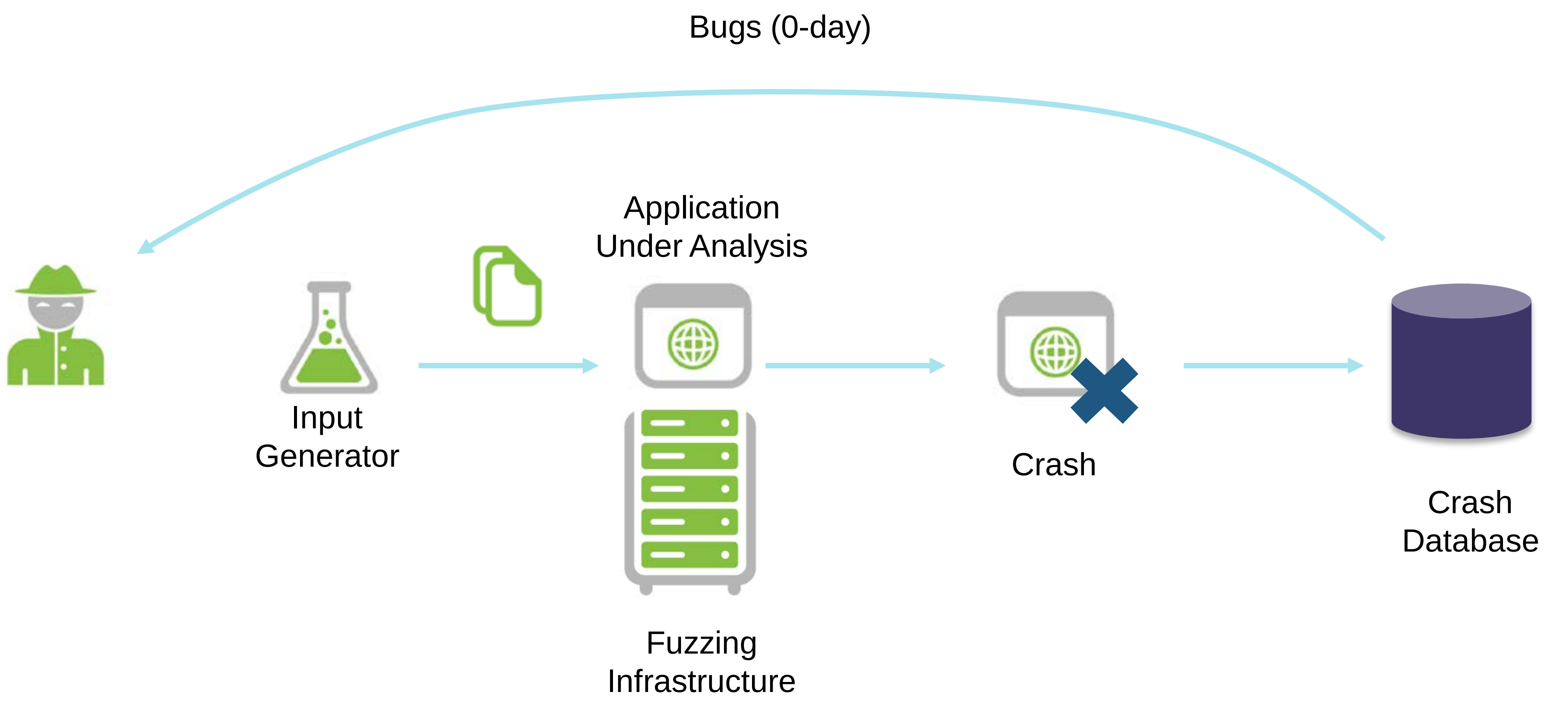
Valid file descriptor.

Unverified user data.

Fuzzing

- Fuzzing is an automated procedure to send inputs and record safety condition violations as crashes
  - Assumption: crashes are potentially exploitable
- Several dimensions in the fuzzing space
  - How to supply inputs to the program under test?
  - How to generate inputs?
  - How to generate more "relevant" crashes?
  - How to change inputs between runs?
- Goal: maximized effectiveness of the process

Fuzzing



Bugs (0-day)
Application Under Analysis
Input Generator
Crash
Crash Database
Fuzzing Infrastructure

Input Generation

- Inputs to programs under test depend on the program
  - Files, network, environment
- Input generation strategies
  - Random data
  - Mutated data
  - Data generated from a grammar

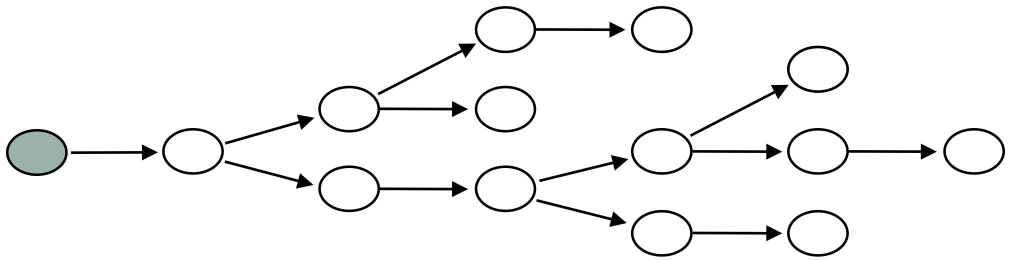
Random Inputs

- Inputs generated randomly
- Easy to write, many tools available, works well for (pathologically) buggy programs
- Many disadvantages
  - More crash analysis required
  - More duplication of results
  - Will not trigger hard-to-reach bugs

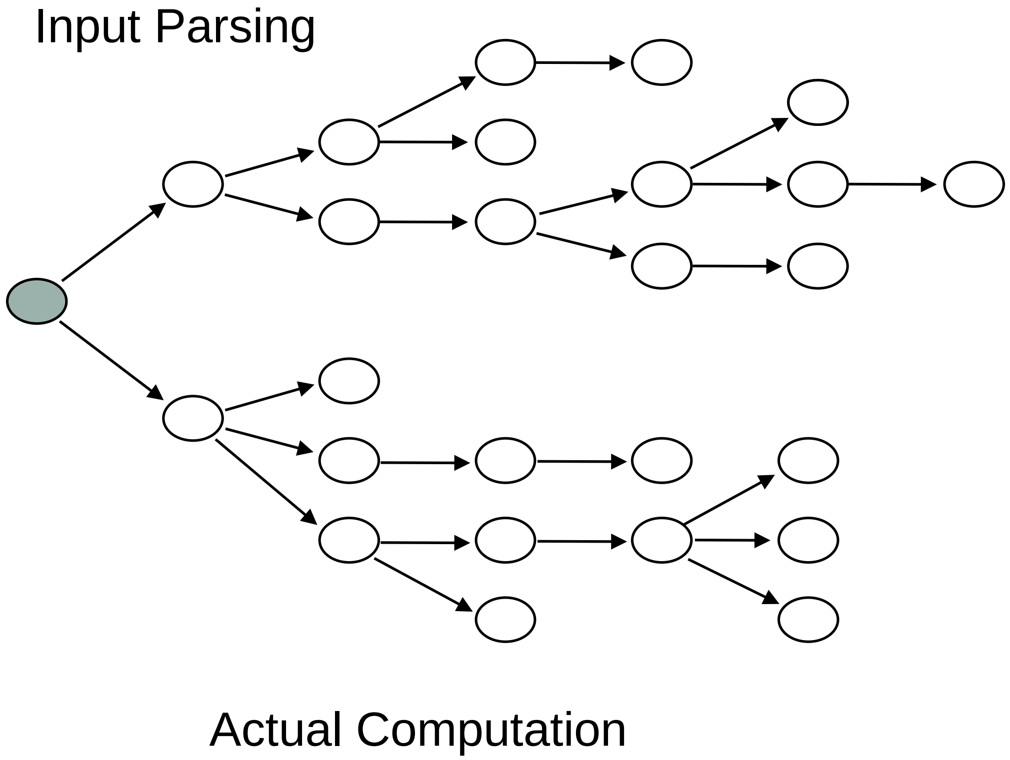
Random Fuzzing

Random Fuzzing

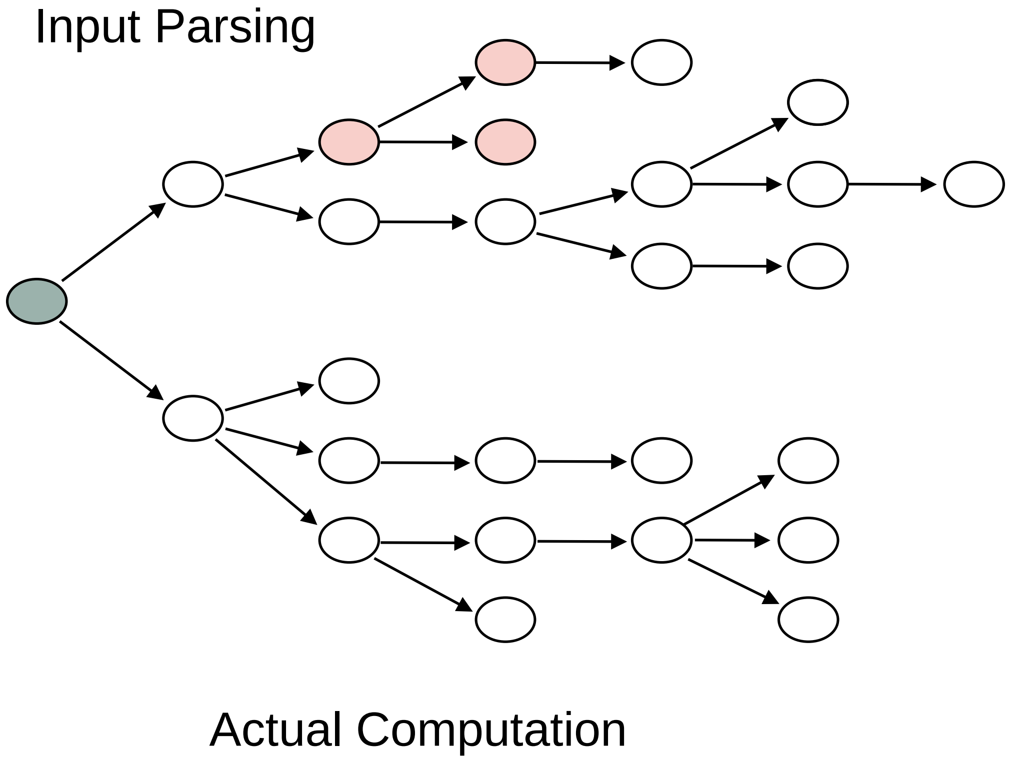
Input Parsing



Random Fuzzing

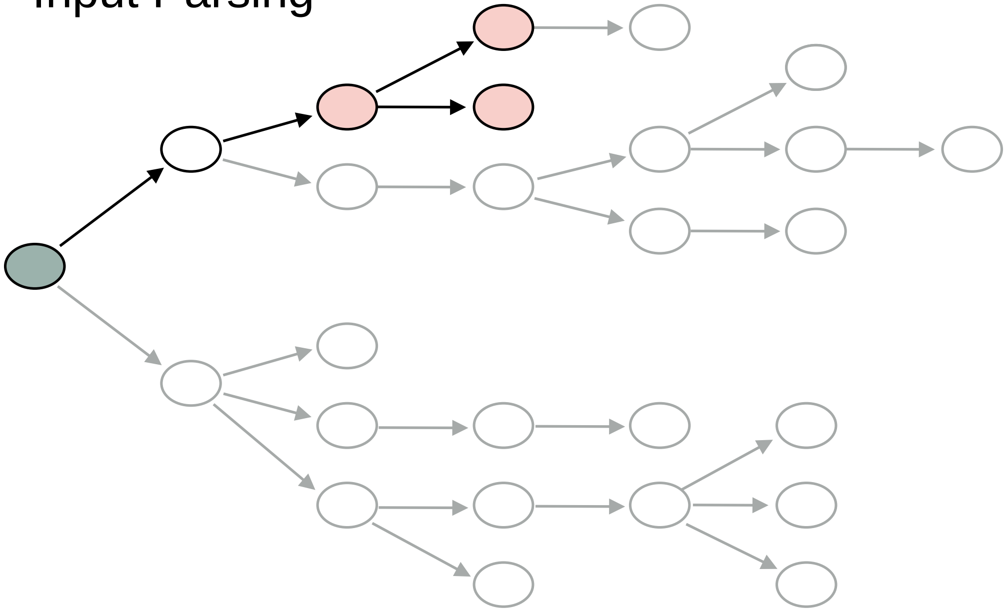


Random Fuzzing



Random Fuzzing

Input Parsing



Actual Computation

Generative Fuzzing

- Goal: Construct a grammar that produces “reasonable” inputs, and sample from the corresponding input space
- Another approach to exploring program beyond initial parsing and validation stages
- However, requires understanding the input space and constructing a grammar
  - More up-front work compared to random or mutational fuzzing
  - Not certain that grammar can trigger bugs

Fuzzing ioctl

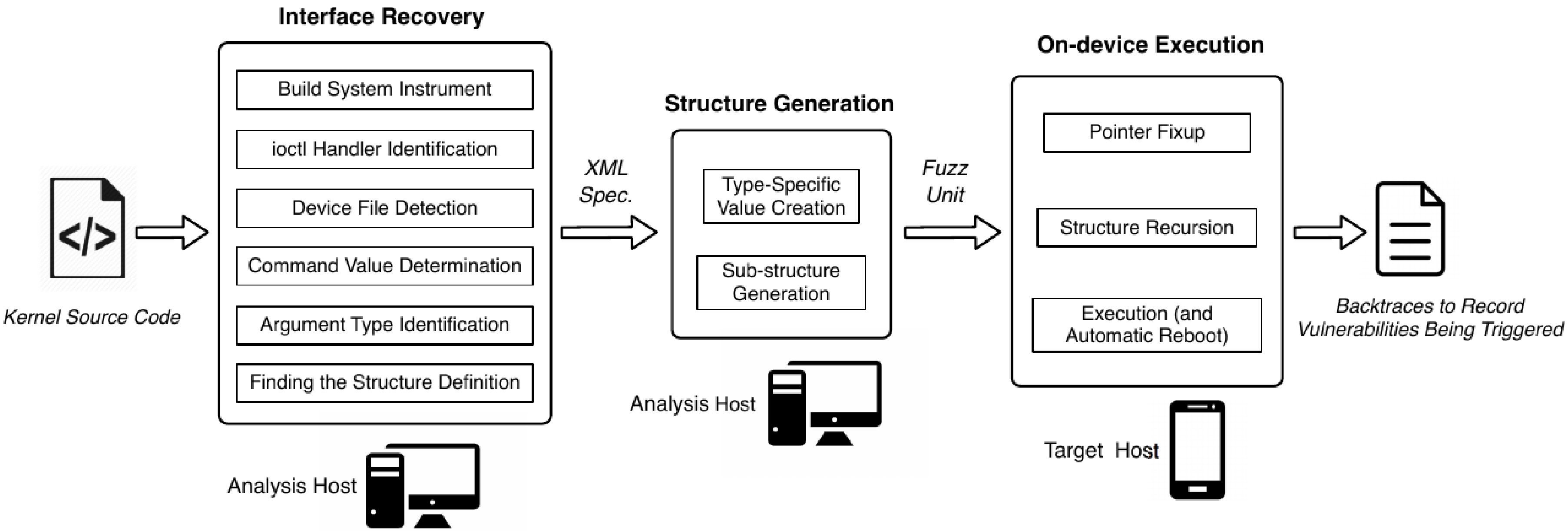
- ioctl routines have highly structured input
- Can we use the input grammar to support better fuzzing?
- Track all data type information associated with the destination of a copy_from_user operation where the source argument is param

DIFUZE: Interface Aware Fuzzing

- Recover all the command values and the corresponding param types automatically

- Use this information to reduce the state space and help in effective fuzzing

DIFUZE



Interface Recovery
Build System Instrument
ioctl Handler Identification
Device File Detection
Command Value Determination
Argument Type Identification
Finding the Structure Definition
Analysis Host
Kernel Source Code
XML Spec.
Structure Generation
Type-Specific Value Creation
Sub-structure Generation
Analysis Host
Fuzz Unit
On-device Execution
Pointer Fixup
Structure Recursion
Execution (and Automatic Reboot)
Target Host
Backtraces to Record Vulnerabilities Being Triggered

Evaluation

Manufacturer	Device	Chipset
Google	Pixel	Qualcomm
HTC	E9 Plus	Mediatek
HTC	One M9	Qualcomm
Huawei	P9 Lite	Huawei
Huawei	Honor 8	Huawei
Samsung	Galaxy S6	Samsung
Sony	Xperia XA	Mediatek

36 0-day Bugs Found

Crash Type	Count
Arbitrary Read	4
Arbitrary Write	4
Assert Failure	6
Buffer Overflow	2
Null Dereference	9
Out of Bound Index	5
Uncategorized	5
Writing to non-volatile memory	1

From Drivers to Bootloaders

- Bootloader
  - Initializes the device and its peripherals
  - Loads the kernel code from secondary storage
  - Jumps to it
- No standard (e.g., ARM gives guidelines)
- Booting through several stages
- Protect integrity of user's device and data
- Bootloader unlocking

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

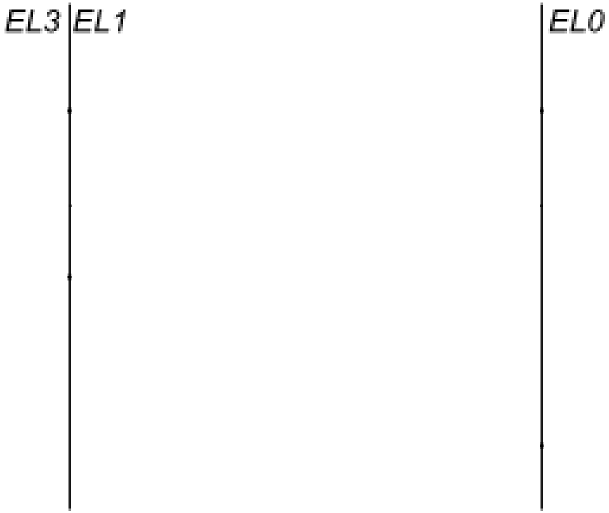
- Integrity of the booting process
  - Android OS is verifiably to be in a non-tampered state
  - A root process cannot interfere with peripherals setup
- Unlocking security mechanism
  - A root process cannot unlock the bootloader
  - Physical attacker cannot unlock the bootloader

Threat Model

- 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

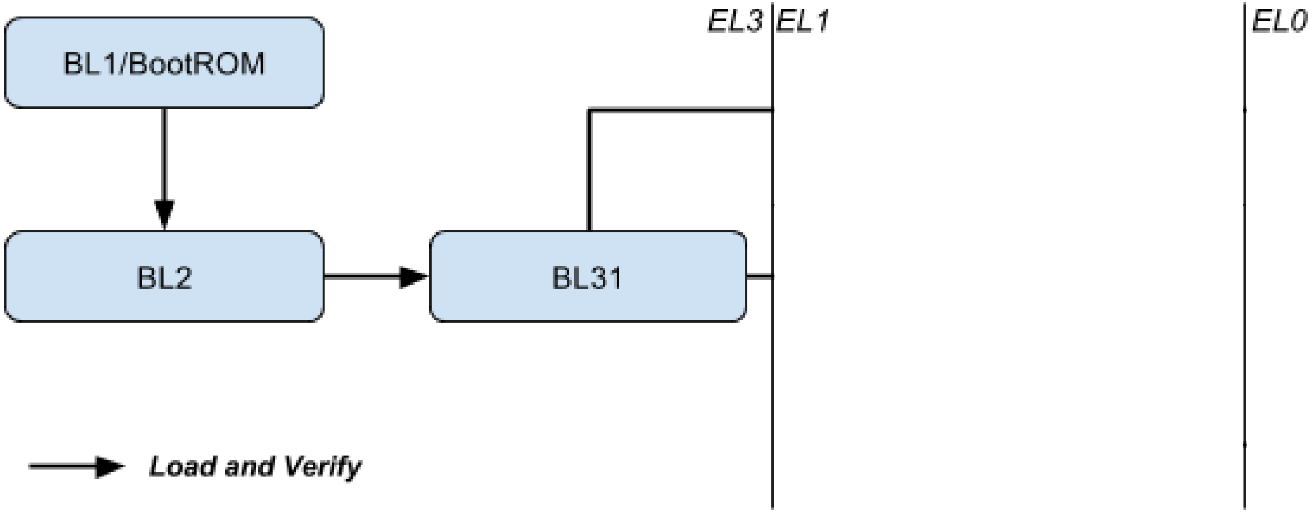
Booting Process

God mode | Kernel mode | User mode



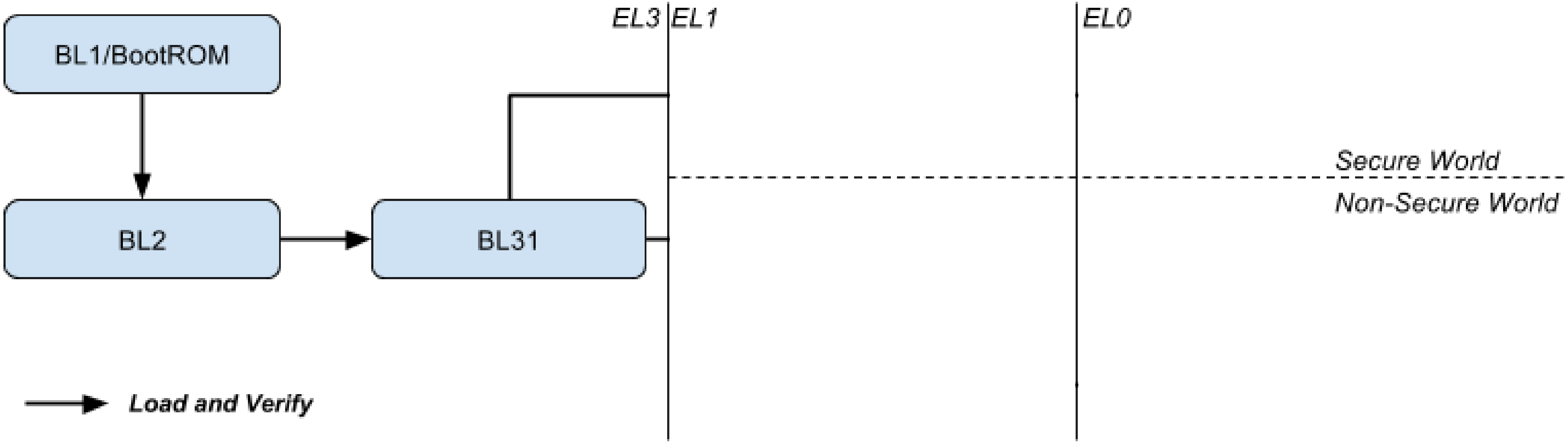
Booting Process

God mode | Kernel mode | User mode

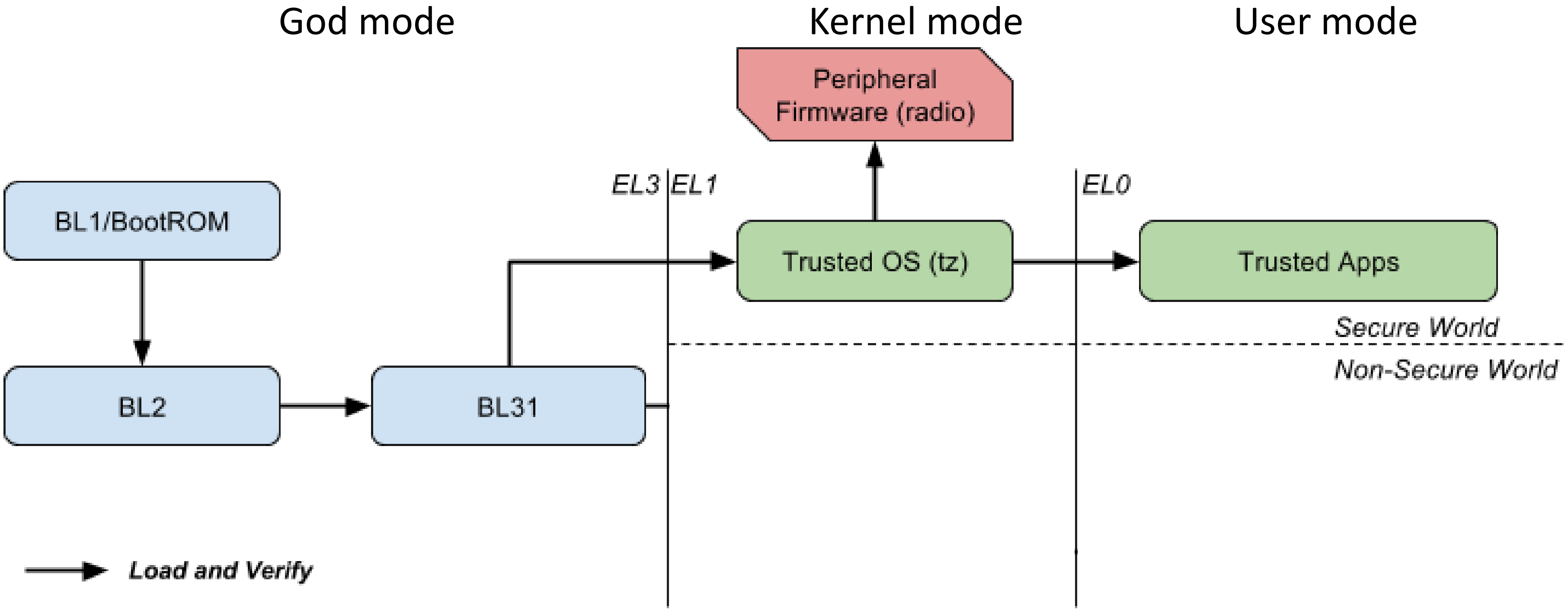


Booting Process

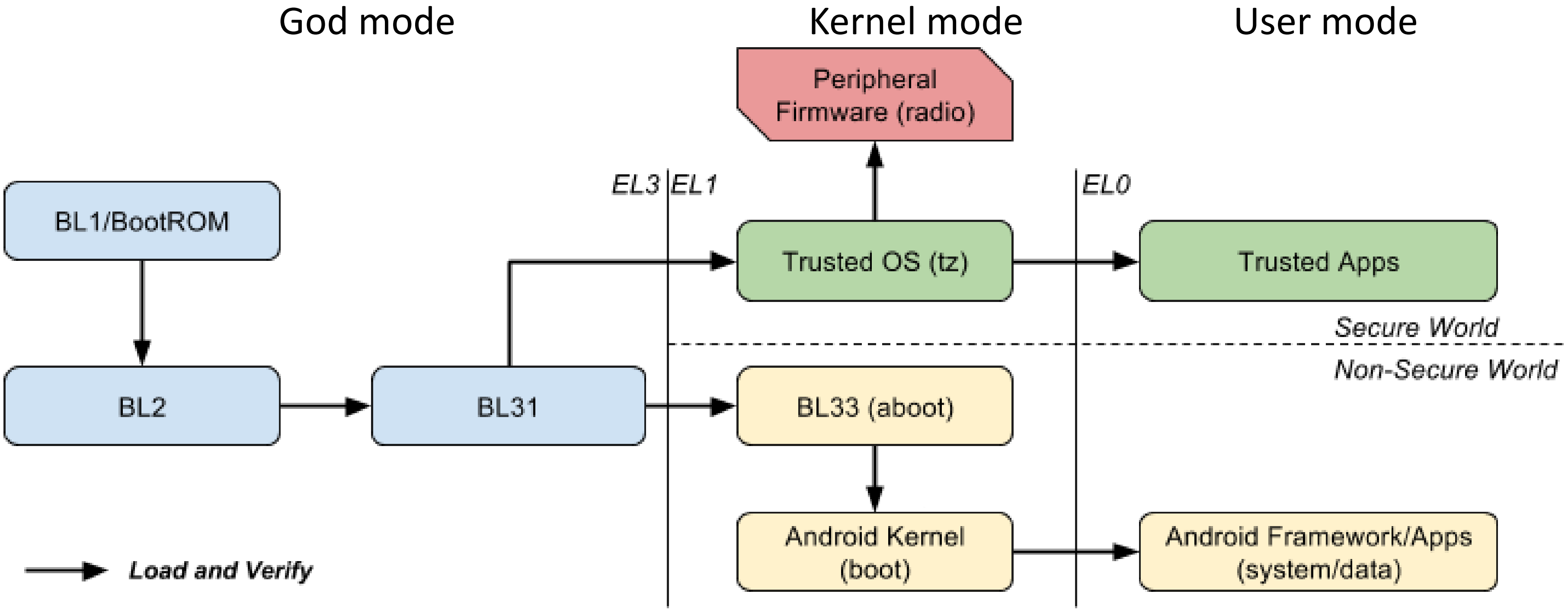
God mode | Kernel mode | User mode



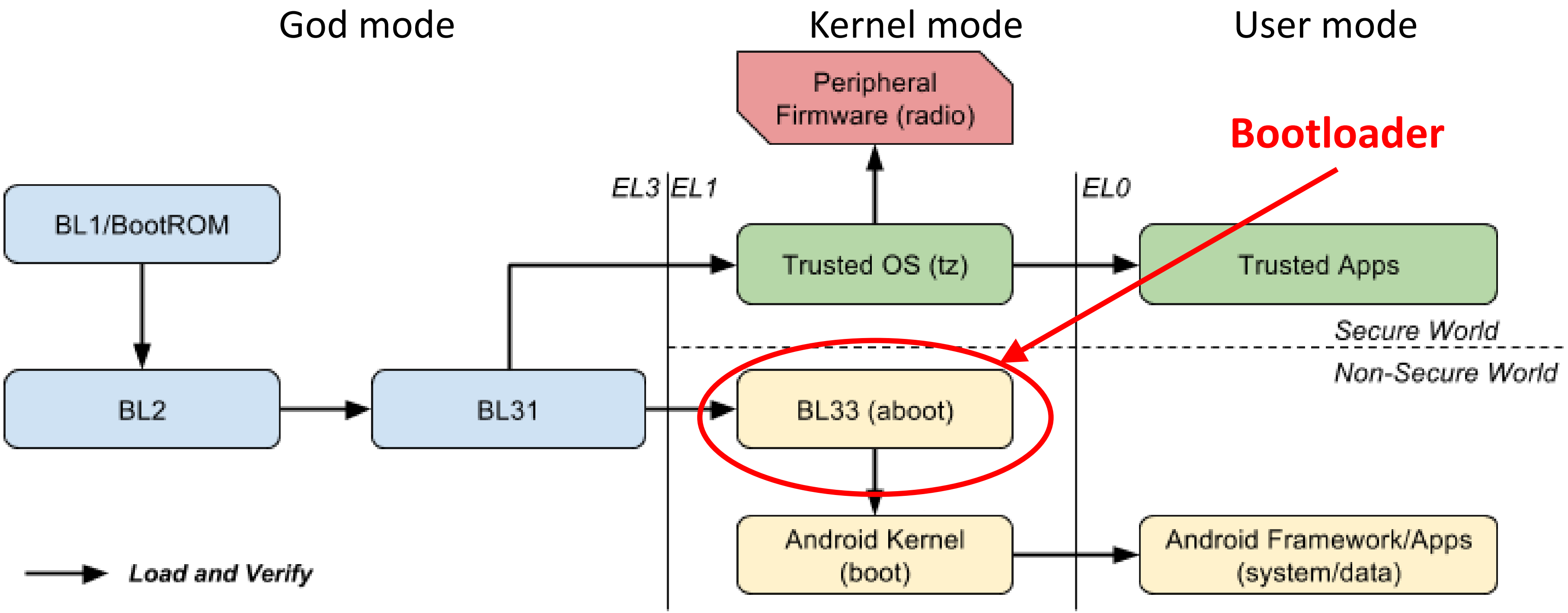
Booting Process



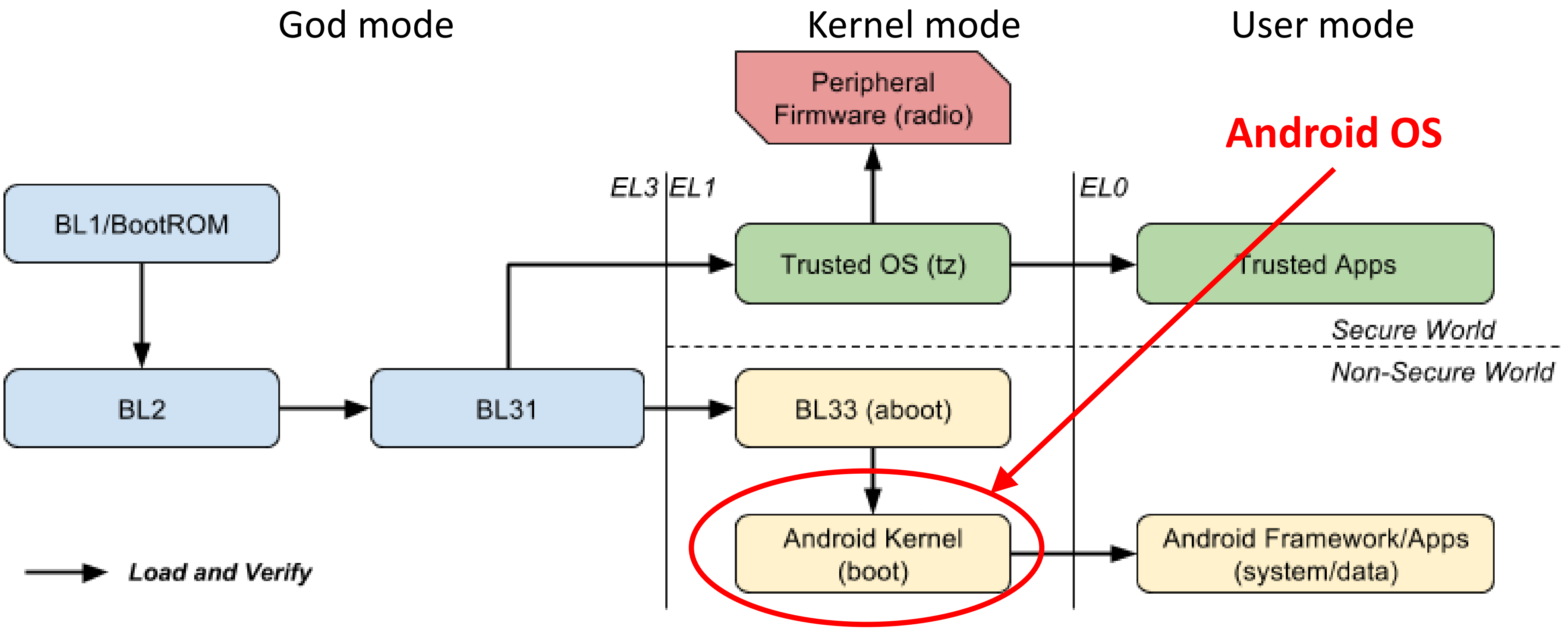
Booting Process



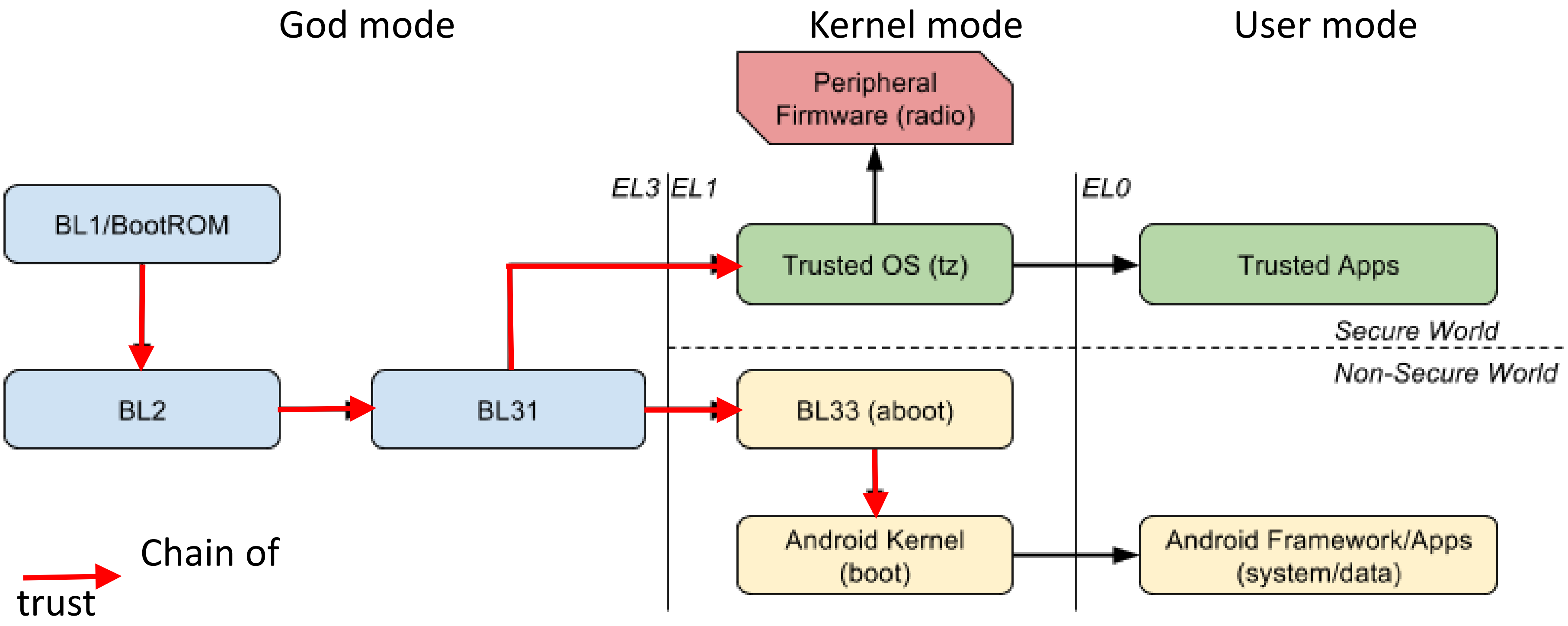
Booting Process



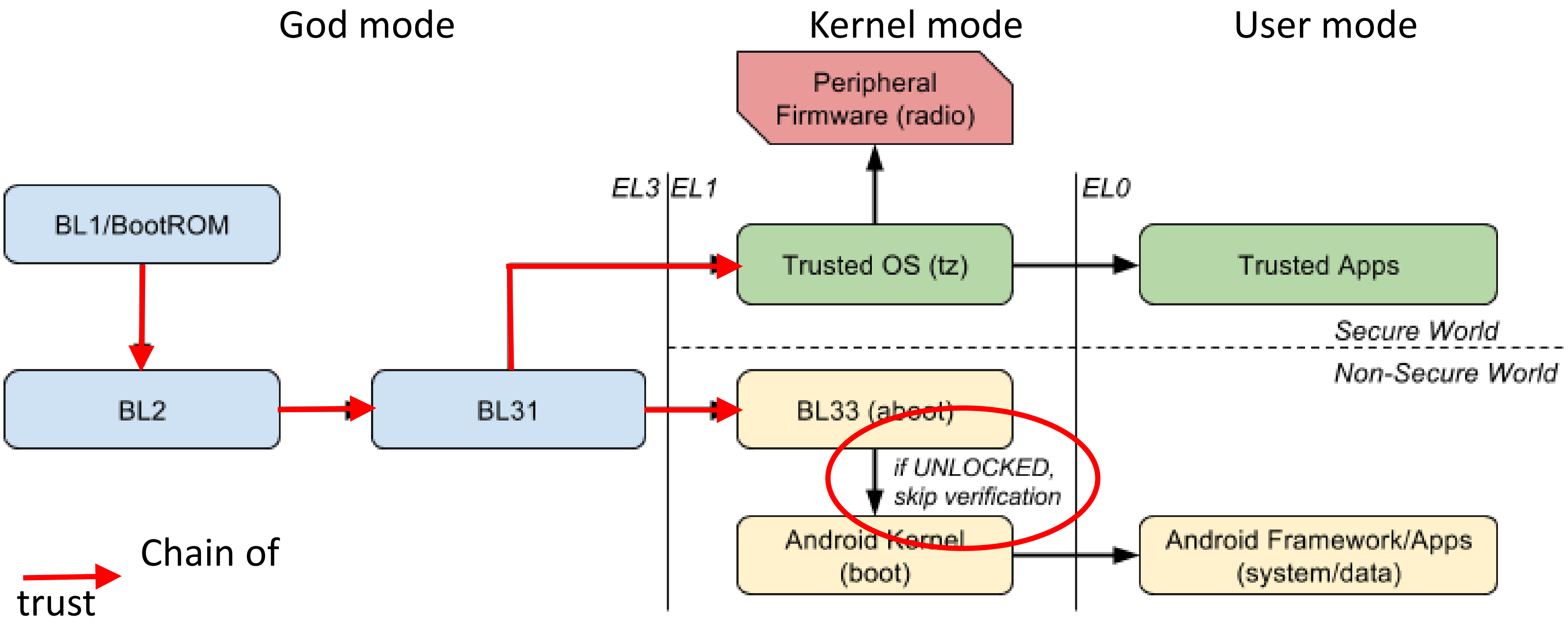
Booting Process



Booting Process

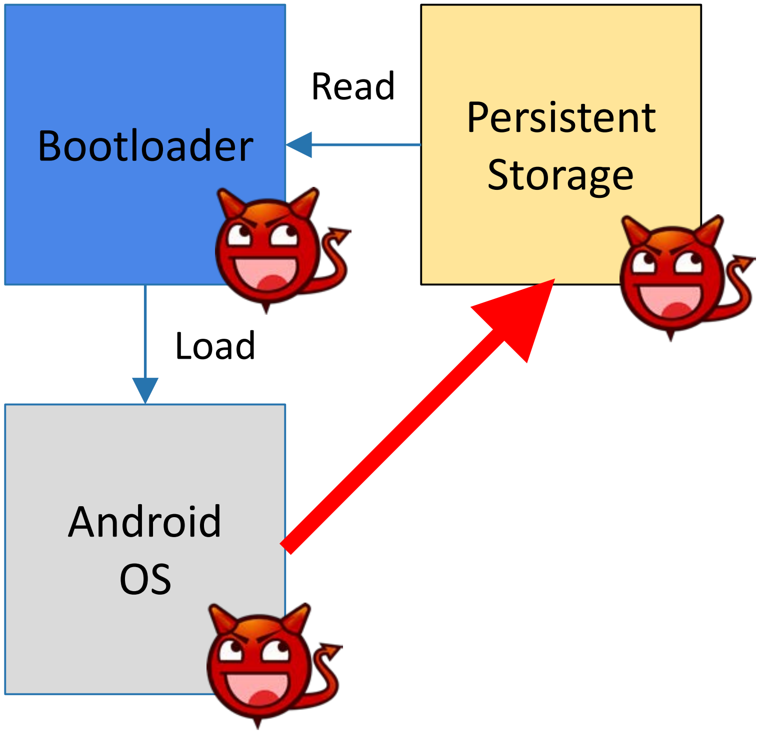


Booting Process



Can a compromised Android OS affect the booting process?

Yes!


Bootloader
Read
Persistent Storage
Load
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
    - There is no memcpy!

BootStomp: A Bootloader Analyzer

- Automatic static binary tool for finding security vulnerabilities in bootloaders
- Uses multi-tag taint analysis based on under-constrained dynamic symbolic execution
- Determines whether attacker-controlled data can influence the bootloader's intended behavior
- Traceable output
  - Verify generated alerts

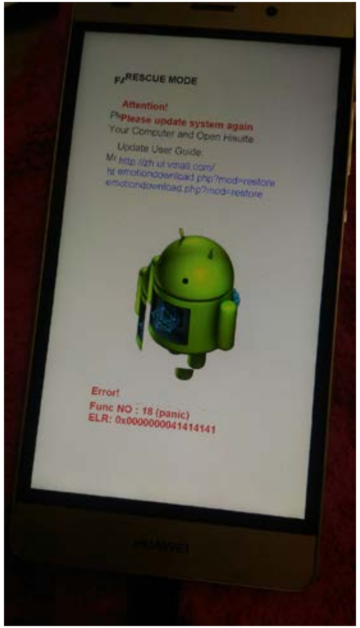
BootStomp: A Bootloader Analyzer

- Arbitrary memory writes
- Arbitrary memory reads
- Control over loop iterations
- Bypassing of the unlocking mechanism
  - Functions overwriting the security state on persistent storage

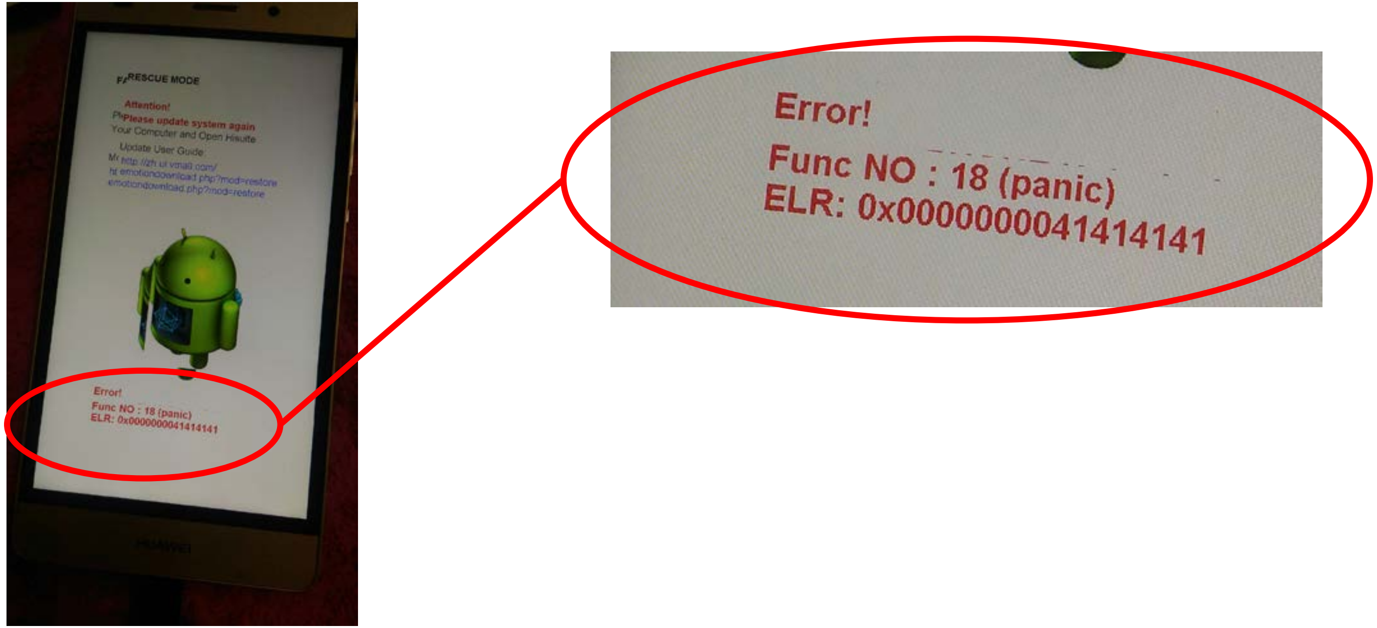
Evaluation: Bugs

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)

Evaluation: Bugs



Evaluation: Bugs



Responsible Disclosure

- All bugs reported, acknowledged and already fixed

Conclusions

- Android-based smartphones are part of our everyday life
- We need better automated tools to analyze the myriad of vendor drivers and bootloaders
- Find the bugs before the bad guys do!
- More techniques, more approaches, more targets are needed!

Continuous, Crowdsourced Innovation…

Help Make Drivers and Bootloaders Better!

- https://github.com/ucsb-seclab/dr_checker
- https://github.com/ucsb-seclab/difuze
- https://github.com/ucsb-seclab/bootstomp

lastline

QUESTIONS?

vigna@lastline.com

vigna@cs.ucsb.edu

UNIVERSITY OF CALIFORNIA · SANTA BARBARA · LET THERE BE LIGHT