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?
Smartphone Vulnerabilities

Trend Micro Awards $515,000 at Mobile Pwn2Own2017

By: Sean Michael Kerner  |  November 02, 2017

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

<table>
<thead>
<tr>
<th>Severity</th>
<th>Complete Report* + PoC</th>
<th>Payment range (if report includes an exploit leading to Kernel compromise)**</th>
<th>Payment range (if report includes an exploit leading to TEE compromise)**</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical</td>
<td>Required</td>
<td>Up to $150,000</td>
<td>Up to $200,000</td>
</tr>
<tr>
<td>High</td>
<td>Required</td>
<td>Up to $75,000</td>
<td>Up to $100,000</td>
</tr>
<tr>
<td>Moderate</td>
<td>Required</td>
<td>Up to $20,000</td>
<td>Up to $35,000</td>
</tr>
<tr>
<td>Low</td>
<td>Required</td>
<td>Up to $330</td>
<td>Up to $330</td>
</tr>
</tbody>
</table>
Apps vs. System

- Facebook
- Walmart
- Twitter
- Pinterest
- Google Maps
- Skype
- YouTube
- Bank of America
- Tesla
- Venmo
- PayPal
- SMS
- Nest
- Dropbox

RSA Conference 2018
Automated Vulnerability Analysis

Human

Semi-Automated

Fully Automated
Many Weapons, Many Targets

Static Analysis
- Fuzzing
- Symbolic Execution
- Data Flow Analysis

Dynamic Analysis
- Taint Analysis
- Concolic Execution
- Points-to Analysis

Bootloaders
- Operating System
- GUIs
- Back-end Components

Drivers
- Apps
- Trusted Execution Environments

RSAC Conference 2018
Many Weapons, Many Targets

Static Analysis
- Fuzzing
- Dynamic Analysis
- Taint Analysis
- Symbolic Execution
- Data Flow Analysis

Concolic Execution

Operating System
- GUIs
- Back-end Components
- Apps

Drivers

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?

- Vendor: 85.0%
- Core Kernel: 15.0%

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

Driver Code

Analysis Clients

Points-to Analysis

Taint Analysis

Global State

Vulnerability Detectors

Improper Tainted-Data Use Detector (ITDUD)
Tainted Arithmetic Detector (TAD)
Invalid Cast Detector (ICD)
Tainted Loop Bound Detector (TLBD)
Tainted Pointer Dereference Detector (TPDD)
Tainted Size Detector (TSD)
Uninit Leak Detector (ULD)
Global Variable Race Detector (GVRD)

Warnings
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
```c
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);
}
```
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);
}
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);
}
```c
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);
}
```
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);
}
Taint Analysis

user_ptr

len

ko

curr_data->item

Warning: Improper Tainted-Data Use

Warning: Tainted Loop Bound

Field-sensitive

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);"
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);
}
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);
}
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);
}
```c
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);
}
```
Driver Entry Points

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

<table>
<thead>
<tr>
<th>Entry Type</th>
<th>Argument(s)</th>
<th>Taint Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read (File)</td>
<td>char *buf, size_t len</td>
<td>Direct</td>
</tr>
<tr>
<td>Write (File)</td>
<td>char *buf, size_t len</td>
<td>Direct</td>
</tr>
<tr>
<td>ioctl (File)</td>
<td>long args</td>
<td>Direct</td>
</tr>
<tr>
<td>DevStore (Attribute)</td>
<td>const char *buf</td>
<td>Indirect</td>
</tr>
<tr>
<td>NetDevloctl (Socket)</td>
<td>struct *ifreq</td>
<td>Indirect</td>
</tr>
<tr>
<td>V4loctl</td>
<td>struct v412_format *f</td>
<td>Indirect</td>
</tr>
</tbody>
</table>
Evaluation: Mobile Kernels

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 Venus P9 Lite (2016-03-29)

HTC Desire A56 (a56uhl-3.4.0)
LG K8 ACG (AS375)
ASUS Zenfone 2 Laser (ZE550KL / MR5-21.40.1220.1794)

Samsung Galaxy S7 Edge (SM-G935F NN)

3.1 Million lines of driver code
# Dr. Checker Results

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

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

**Total:** 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)

static char call status;
...
static ssize_t accdet_store_call_state(struct device *ddev, const char *buf, size_t count) {
    int ret = sscanf(buf, "%s", &call status);
    Warning: Improper Tainted-Data Use

    if (ret != 1) {
        ACCDETDEBUG("accdet: Invalid values\n");
        return -EINVAL;
    }
    ...
}
How Are Kernel Bugs Reached from User Space?

- ioctl: 63.0%
- Others: 37.0%

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

- **Input/Output Control**

- 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(int fd, unsigned long command, unsigned long param);
ioctl(  
  int *fd,  \[\text{Valid file descriptor.}\]
  unsigned long *command,  
  unsigned long *param
  );

Unverified user data.
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
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

Input Parsing

Actual Computation
Random Fuzzing

Input Parsing

Actual Computation
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

Kernel Source Code

Interface Recovery
- Build System Instrument
- ioctl Handler Identification
- Device File Detection
- Command Value Determination
- Argument Type Identification
- Finding the Structure Definition

XML Spec.

Structure Generation
- Type-Specific Value Creation
- Sub-structure Generation

Fuzz Unit

Analysis Host

On-device Execution
- Pointer Fixup
- Structure Recursion
- Execution (and Automatic Reboot)

Backtraces to Record Vulnerabilities Being Triggered

Target Host
## Evaluation

<table>
<thead>
<tr>
<th>Manufacturer</th>
<th>Device</th>
<th>Chipset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google</td>
<td>Pixel</td>
<td>Qualcomm</td>
</tr>
<tr>
<td>HTC</td>
<td>E9 Plus</td>
<td>Mediatek</td>
</tr>
<tr>
<td>HTC</td>
<td>One M9</td>
<td>Qualcomm</td>
</tr>
<tr>
<td>Huawei</td>
<td>P9 Lite</td>
<td>Huawei</td>
</tr>
<tr>
<td>Huawei</td>
<td>Honor 8</td>
<td>Huawei</td>
</tr>
<tr>
<td>Samsung</td>
<td>Galaxy S6</td>
<td>Samsung</td>
</tr>
<tr>
<td>Sony</td>
<td>Xperia XA</td>
<td>Mediatek</td>
</tr>
</tbody>
</table>
## 36 0-day Bugs Found

<table>
<thead>
<tr>
<th>Crash Type</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arbitrary Read</td>
<td>4</td>
</tr>
<tr>
<td>Arbitrary Write</td>
<td>4</td>
</tr>
<tr>
<td>Assert Failure</td>
<td>6</td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>2</td>
</tr>
<tr>
<td>Null Dereference</td>
<td>9</td>
</tr>
<tr>
<td>Out of Bound Index</td>
<td>5</td>
</tr>
<tr>
<td>Uncategorized</td>
<td>5</td>
</tr>
<tr>
<td>Writing to non-volatile memory</td>
<td>1</td>
</tr>
</tbody>
</table>
**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

<table>
<thead>
<tr>
<th>God mode</th>
<th>Kernel mode</th>
<th>User mode</th>
</tr>
</thead>
</table>

EL3 | EL1 | EL0 |
Booting Process

God mode                                      Kernel mode                                      User mode

BL1/BootROM → BL2 → BL31 → EL3 → EL1

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
Booting Process

God mode

Kernel mode

User mode

BL1/BootROM
BL2
BL31

Peripheral Firmware (radio)

Trusted OS (tz)

BL33 (aboot)

Android Kernel (boot)

Trusted Apps

Android Framework/Apps (system/data)

Load and Verify
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)

Load and Verify
Booting Process

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

Android OS
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)

Secure World

Non-Secure World

Chain of trust
Booting Process

God mode
- BL1/BootROM
- BL2
- BL31

Kernel mode
- Peripheral Firmware (radio)
- Trusted OS (tz)
- BL33 (aboot)
- Android Kernel (boot)
- if UNLOCKED, skip verification

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

Chain of trust
Can a compromised Android OS affect the booting process?

Yes!
Bootloader → Read

Load

Persistent Storage → 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

<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 (60 days)</strong></td>
</tr>
</tbody>
</table>
Evaluation: Bugs
Evaluation: Bugs

Error!
Func NO: 18 (panic)
ELR: 0x000000041414141
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
QUESTIONS?

vigna@lastline.com
vigna@cs.ucsb.edu