Mind the Gap –
Uncovering the Android patch gap through binary-only patch analysis
HITB conference, April 13, 2018

Jakob Lell <jakob@srlabs.de>
Karsten Nohl <nohl@srlabs.de>
Allow us to take you on two intertwined journeys

This talk in a nutshell

- **Research journey**
  - Wanted to understand how fully-maintained Android phones can be exploited
  - Found surprisingly large patch gaps for many Android vendors
  - Also found Android exploitation to be unexpectedly difficult

- **Engineering journey**
  - Wanted to check thousands of firmwares for the presence of hundreds of patches
  - Developed and scaled a rather unique analysis method
  - Created an app for your own analysis
Android patching is a known-hard problem

### Patching challenges

- Computer OS vendors regularly issue patches
- Users “only” have to confirm the installation of these patches
- Still, enterprises consider regular patching among the most effortful security tasks

### Patch ecosystems

**OS vendor**
- Microsoft
- Apple
- Linux distro

**Endpoints & servers**

**OS patches**

**Endpoints & servers**

### The nature of Android makes patching so much more difficult

- Patches are handed down a long chain of typically four parties before reaching the user
- Only some devices get patched (2016: 17% [2]). We focus our research on these “fully patched” phones

### Our research question – How many patching mistakes are made in this complex Android ecosystem? That is: how many patches go missing?
Vendor patch claims can be unreliable; independent verification is needed

How do we determine whether an Android binary has a patch installed, without access to the corresponding source code?

<table>
<thead>
<tr>
<th>Trust vendor claims?</th>
<th>Try exploiting the corresponding vulnerability?</th>
<th>Apply binary-only patch heuristics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image" alt="Perform test" /></td>
<td><img src="image" alt="Find evidence in the binary itself on whether a patch is installed" /></td>
</tr>
<tr>
<td>Claimed patch level: 2017-09-01</td>
<td><img src="image" alt="No exploits publicly available for most Android bugs" /></td>
<td><img src="image" alt="Scale to cover hundreds of patches and thousands of phones" /></td>
</tr>
<tr>
<td>2017-05</td>
<td><img src="image" alt="A missing patch also does not automatically imply an open vulnerability (It’s complicated. Let’s talk about it later)" /></td>
<td><img src="image" alt="The topic of this presentation" /></td>
</tr>
<tr>
<td>2017-06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017-07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017-08</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017-09</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Important distinction: A missing patch is *not* automatically an open security **vulnerability**. We’ll discuss this a bit later.
### Patching is necessary in the Android OS and the underlying Linux kernel

<table>
<thead>
<tr>
<th>Android OS patching (&quot;userland&quot;)</th>
<th>Linux kernel patching</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Responsibility</strong></td>
<td><strong>Same kernel that is used for much of the Internet</strong></td>
</tr>
<tr>
<td>Android Open Source Project (AOSP) is maintained by Google</td>
<td><strong>Maintained by a large ecosystem</strong></td>
</tr>
<tr>
<td>In addition, chipset and phone vendors extend the OS to their needs</td>
<td><strong>Chipset and phone vendors contribute hardware drivers, which are sometimes kept closed-source</strong></td>
</tr>
<tr>
<td><strong>Security relevance</strong></td>
<td><strong>Attackable mostly from within device</strong></td>
</tr>
<tr>
<td>Most exposed attack surface: The OS is the primary layer of defense for remote exploitation</td>
<td><strong>Relevant primarily for privilege escalation (&quot;rooting&quot;)</strong></td>
</tr>
<tr>
<td><strong>Patch situation</strong></td>
<td><strong>Large number of vulnerability reports, only some of which are relevant for Android</strong></td>
</tr>
<tr>
<td>Monthly security bulletins published by Google</td>
<td><strong>Tendency to use old kernels even with latest Android version; e.g., Kernel 3.18 from 2014, end-of-life: 2017</strong></td>
</tr>
<tr>
<td>Clear versioning around Android, including a patch level date, which Google certifies for some phones</td>
<td>We focus our attention on userland patches</td>
</tr>
</tbody>
</table>

---

We focus our attention on userland patches.
Agenda

- Research motivation

Spot the Android patch gap

- Try to exploit Android phones
We want to check hundreds of patches on thousands of Android devices.

**Android userland patch analysis**
- Android’s 2017 security bulletins list ~280 bugs (~CVEs) with Critical or High severity
- Source code is available for ~240 of these bugs
- Of these userland bugs, ~180 originate from C/C++ code (plus a few Java)
- So far, we implemented heuristics for 164 of the corresponding patches

**Out-of-scope (for now)**
- ~700 kernel and medium/low severity userland patches
- The remaining bugs are in closed-source vendor-specific components
- We do not yet support most Java patches

The heuristics would optimally work on hundreds of thousands of Android firmwares:
- 60,000 Android variants [3]
- Regular updates for many of these variants
The patch gap: Android patching completeness varies widely for different phones

<table>
<thead>
<tr>
<th>Device</th>
<th>Android version</th>
<th>Patch level</th>
<th>2016</th>
<th>2017</th>
<th>Patches &quot;missing&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
<td>Critical</td>
</tr>
<tr>
<td>Google Pixel 2</td>
<td>8.1</td>
<td>Feb 2018</td>
<td>9</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>Samsung J5 (2016)</td>
<td>7.1.1</td>
<td>Aug 2017</td>
<td>11</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>Samsung J3 (2016)</td>
<td>5.1.1</td>
<td>Jan 2018</td>
<td>12</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Wiko Freddy</td>
<td>6.0.1</td>
<td>Sep 2017</td>
<td></td>
<td></td>
<td>18</td>
</tr>
</tbody>
</table>

Legend:
- Green: Not affected
- Light green: Patch found applied as claimed
- Medium green: Patched found above claimed level
- Red: Patch not found within claimed level
- Orange: Patch not found outside claimed level
- Black: Claimed patch level
- Dashed black: Android version release date
- Not tested

Security Research Labs
Binary-only analysis: Conceptually simple

1. Prepare patch test set
   - Vulnerable source code
     - Apply patch
   - Compile with different compilers, compiler configurations, CPU options
   - Mask volatile information (e.g. call destinations)
   - Collection of unpatched binaries

2. Test for patch presence
   - Binary file
   - Mask volatile information
   - Compare to collections: Find match with patched or unpatched sample
A bit more background: Android firmwares go from source code to binaries in two steps.

### Source code
contains placeholders that are filled in during preprocessing

```c
#include <limits.h>
#include <string.h>
void foo(char* fn){
    char buf[PATH_MAX];
    strncpy(buf, fn, PATH_MAX);
}
```

### Compiler
preprocessors and compiles source code into object files that are then fed into the linker

```assembly
stp   x28, x27, [sp, #-32]!  
      […]
orr   w2, wzr, #0x1000
mov   x1, x8
bl    0 <strncpy>
      […]
ret
```

### Linker
combines the object files into an executable firmware binary.

```assembly
stp   x28, x27, [sp, #-32]!  
      […]
orr   w2, wzr, #0x1000
mov   x1, x8
bl    11b3e8 <strcpy@plt>
      […]
ret
```
The basic idea: Signatures can be generated from reference source code

Compile reference source code (before and after patch)

Parse disassembly listing for relocation entries

Disassembly of object file, after compiler but before linker

```
0000000000000000 <impeg2d_api_reset>:
  0:  a9bd7bfd   stp    x29, x30, [sp, #-48]!
  4:  910003fd   mov    x29, sp
[...]
 20:  f9413e60   ldr    x0, [x19, #632]
 24:  52800042   mov    w2, #0x2                        // #2
 28:  b9402021   ldr    w1, [x1, #32]
 2c:  94000000   bl     0
```

Instruction format of the bl instruction

```
100 01  ii iiii iiii iiii iiii iiii iiii iiii
```

Sanitize instructions
Toss out irrelevant destination addresses of the instruction

Create hash of remaining binary code

Generate signature containing function length, position/type of relocation entries, and hash of the code
At scale, three compounding challenges need to be solved

- **Too much source code**
  - There is too much source code to collect
  - Once collected, there is too much source code to compile

- **Too many compilation possibilities**
  - Hard to guess which compiler options to use
  - Need to compile same source many times

- **Hard to find code “needles” in binary “haystacks”**
  - Without symbol table, whole binary needs to be scanned
  - Thousands of signatures of arbitrary length
Signature generation would require huge amounts of source code

---

### One Android source code tree is roughly 50 GiB in size

Source code trees are managed in a manifest, which lists git repositories with revision and path in a source code tree

```xml
...<project name="platform/external/zxing" revision="d2256df36df8778a3743e0a71eab0cc5106b98c9"/>
<project name="platform/frameworks/av" revision="330d132dfab2427e940cfaa2184a2e549579445d"/>
<project name="platform/frameworks/base" revision="85838f4eaa8c8c8d38c4262e749171e59a275d02"/>
...+
```

~500 MORE REPOSITORIES

---

### Signature generation requires many source code trees

- **Hundreds of different Android revisions** (e.g. android-7.1.2_r33)
- **Device-specific source code trees** (From Qualcomm Codeaurora CAF)

Currently ~1100 source code trees are used in total (many more exist!)

1100 x 50 GiB = 55 TiB

Would require huge amount of storage, CPU time, and network traffic to check out everything.
We leverage a FUSE (filesystem in userspace) to retrieve files only on demand.

**Insight: The same git repositories are used for many manifests.**

<table>
<thead>
<tr>
<th>Manifest 1</th>
<th>Manifest 2</th>
<th>Manifest 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>platform/frameworks/av rev 330d132d</td>
<td>platform/frameworks/base rev 85838fea</td>
<td>platform/frameworks/av rev deadbeef</td>
</tr>
<tr>
<td>platform/frameworks/base rev 85838fea</td>
<td>platform/frameworks/av rev d43a8fe2</td>
<td>platform/frameworks/base rev cafebabe</td>
</tr>
</tbody>
</table>

**How this can be leveraged**

- **Filesystem in userspace (FUSE)**
  - Store each git repository only once (with `git clone --no-checkout`)
  - Extract files from git repository on demand when the file is read
  - Use database for caching directory contents

- Reduces storage requirement by >99%:
  55 TiB => 300 GiB

- Saves network bandwidth and time required for checkout
- Prevents IP blocking by repository servers
Using our custom FUSE, we can finally generate a large collection of signatures

<table>
<thead>
<tr>
<th>Prepare patch test set</th>
<th>Mount source code tree</th>
<th>Run source-code analysis</th>
<th>Generate build log</th>
<th>Preprocess source files</th>
<th>Recompile with variants</th>
<th>Generate signatures</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>▪ Read manifest</td>
<td>▪ Source-code patch analysis is much easier than binary analysis</td>
<td>▪ Run build system in dry-run mode, don’t compile everything</td>
<td>▪ Use command line from saved build log</td>
<td>▪ &gt;50 different compiler binaries</td>
<td>▪ Evaluate relocation entries and create signatures for each compiler variant</td>
</tr>
<tr>
<td></td>
<td>▪ Use FUSE filesystem to read files on demand</td>
<td>▪ Determines whether a signature match means that the patch is applied or not</td>
<td>▪ Save log of all commands to be executed</td>
<td>▪ Save preprocessor output in database</td>
<td>▪ All supported CPU types</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>▪ Various hacks/fixes to build system required</td>
<td></td>
<td>▪ Optimization levels (e.g. -O2, -O3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>▪ 3897 combinations in total, 74 in our current optimized set</td>
<td></td>
</tr>
</tbody>
</table>

Next question: How many different compiler variants do we need?
Brute-forcing 1000s of compiler variants finds 74 that produce valid signatures for all firmwares tested to date

- Our collection includes 3897 compiler configuration variants, only 74 of which are required for firmwares tested to date.
- To ensure a high rate of conclusive tests, test results are regularly checked for success.
- The test suite is amended with additional variants from the collection as needed.
- The collection itself is amended with additional compiler configuration variants as they become relevant.

Tests are regularly optimized

- For 224 tested 64-bit firmwares, signatures from the first 74 compiler config variants provide full test coverage
- 74 variants → 6,944 signatures → 3MB
- We tried 3,897 variants → 775,795 signatures → 34MB

Just two variants account for 60% of successful sub-tests:
- gcc version 4.9.x-google 20140827 (prerelease)
- Android clang version 3.8.256229
Both were run with each git’s default configuration
Finding needles in a haystack: What do we do if there is no symbol table?

<table>
<thead>
<tr>
<th>Test for patch presence</th>
<th>Challenge</th>
<th>Insight</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function found in symbol table</td>
<td>Checking signature at each position is computationally expensive</td>
<td>Similar problem already solved by rsync</td>
<td>Take advantage of rsync rolling checksum algorithm</td>
</tr>
<tr>
<td>Function not in symbol table</td>
<td>Relocation entries are not known while calculating checksum</td>
<td>Relocation entries are only used for certain instructions</td>
<td>Guess potential relocation entries based on instruction type and sanitize args before checksumming</td>
</tr>
<tr>
<td></td>
<td>32bit code uses Thumb encoding, for which instruction start is not always clear</td>
<td>Same binary code is often also available in 64bit version based on same source code</td>
<td>Only test 64bit code</td>
</tr>
</tbody>
</table>

Simply compare function with pre-computed samples
Using improved rolling signatures, we can efficiently search the binary ‘haystack’ for our code ‘needles’

**Sanitize arguments before checksumming**
- Potential relocation entries are detected based on instruction.
- Zero-out volatile bits

**Match signatures of arbitrary lengths using sliding windows**
- Two overlapping sliding windows
- Only needs powers of 2 as window sizes to match arbitrary function lengths
- Allows efficient scanning of a binary for a large number of signatures

**Process step**
- Hex dump of instruction
- Assembly code / instructions

<table>
<thead>
<tr>
<th>Process step</th>
<th>Hex dump of instruction</th>
<th>Assembly code / instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sanitize arguments before checksumming</td>
<td>97fee7a2</td>
<td>bl e7c40 <a href="mailto:strncpy@plt">strncpy@plt</a></td>
</tr>
<tr>
<td>Match signatures of arbitrary lengths using sliding windows</td>
<td>94000000</td>
<td>bl 0</td>
</tr>
<tr>
<td></td>
<td>f10002ff</td>
<td>cmp x23, #0x0</td>
</tr>
<tr>
<td></td>
<td>1a9f17e8</td>
<td>cset w8, eq</td>
</tr>
<tr>
<td></td>
<td>b40000b6</td>
<td>cbz x22, 10ddbc</td>
</tr>
<tr>
<td></td>
<td>3707fde8</td>
<td>tbnz w8, #0, 10dd6</td>
</tr>
<tr>
<td></td>
<td>f1006d6</td>
<td>subs x22, x22, #0x1</td>
</tr>
<tr>
<td></td>
<td>54ffff42</td>
<td>b.cs 10dd9c</td>
</tr>
<tr>
<td></td>
<td>35fffd48</td>
<td>cbnz w8, 10dd64</td>
</tr>
<tr>
<td></td>
<td>36000255</td>
<td>tbz w21, #0, 10de08</td>
</tr>
<tr>
<td></td>
<td>394082e8</td>
<td>ldrb w8, [x23,#32]</td>
</tr>
<tr>
<td></td>
<td>35000208</td>
<td>cbnz w8, 10de08</td>
</tr>
<tr>
<td></td>
<td>52adad21</td>
<td>mov w1, #0x6d690000</td>
</tr>
<tr>
<td></td>
<td>320003e8</td>
<td>orr w8, wzr, #0x1</td>
</tr>
<tr>
<td></td>
<td>728daca1</td>
<td>movk w1, #0x6d65</td>
</tr>
</tbody>
</table>

To avoid false positives (due to guessed relocation entries), signature is matched from the first window to the end of the overlapping window.
Putting it all together: With all three scaling challenges overcome, we can start testing

### Prepare patch test set

**Mount source code tree**
- Read manifest
- Fuse filesystem to read files on demand

**Run source-code analysis**
- Source-code patch analysis is much easier than binary analysis
- Determines whether a signature match means that the patch is applied or not

**Generate build log**
- Run build system in dry-run mode, don’t compile everything
- Save log of all commands to be executed
- Various hacks/fixes to build system required

### Preprocess source files

**Recompile with variants**
- >50 different compiler binaries
- All supported CPU types
- Optimization levels (e.g. -O2, -O3)
- 3897 combinations in total, 74 in our current optimized set

**Generate signatures**
- Evaluate relocation entries and create signatures for each compiler variant

### Test for patch presence

- Find and extract function (using symbol table or rolling signature)
- Mask relocation entries from signature
- Calculate and compare hash of remaining code
### Vendors differ in how many patches are missing from their phones

<table>
<thead>
<tr>
<th>Missed patches</th>
<th>Vendor</th>
<th>Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 1</td>
<td>Google</td>
<td>Lots</td>
</tr>
<tr>
<td></td>
<td>Sony</td>
<td>Few</td>
</tr>
<tr>
<td></td>
<td>Samsung</td>
<td>Lots</td>
</tr>
<tr>
<td></td>
<td>Wiko</td>
<td>Few</td>
</tr>
<tr>
<td>1 to 3</td>
<td>Xiaomi</td>
<td>Many</td>
</tr>
<tr>
<td></td>
<td>OnePlus</td>
<td>Many</td>
</tr>
<tr>
<td></td>
<td>Nokia</td>
<td>Few</td>
</tr>
<tr>
<td>3 to 4</td>
<td>HTC</td>
<td>Few</td>
</tr>
<tr>
<td></td>
<td>Huawei</td>
<td>Many</td>
</tr>
<tr>
<td></td>
<td>LG</td>
<td>Many</td>
</tr>
<tr>
<td></td>
<td>Motorola</td>
<td>Many</td>
</tr>
<tr>
<td>More than 4</td>
<td>TCL</td>
<td>Many</td>
</tr>
<tr>
<td></td>
<td>ZTE</td>
<td>Few</td>
</tr>
</tbody>
</table>

### Missed patches

<table>
<thead>
<tr>
<th>Chipset</th>
<th>Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samsung</td>
<td>Lots</td>
</tr>
<tr>
<td>Qualcomm</td>
<td>Lots</td>
</tr>
<tr>
<td>HiSilicon</td>
<td>Many</td>
</tr>
<tr>
<td>Mediatek</td>
<td>Many</td>
</tr>
</tbody>
</table>

### Notes
- The tables shows the average number of missing Critical and High severity patches before the claimed patch date.
  * Samples – Few: 5-9; Many: 10-49; Lots: 50+
- Some phones are included multiple times with different firmwares releases.
- Not all patch tests are always conclusive, so the real number of missing patches could be higher.
- Not all patches are included in our tests, so the real number could be higher still.
- Only phones are considered that were patched October-2017 or later.
- A missing patch does not automatically indicate that a related vulnerability can be exploited.
- Again, we show the average of missing High and Critical patches for phones that use these chipsets.
- Samsung phones can run on a Samsung or Qualcomm chipset.
Agenda

- Research motivation
- Spot the Android patch gap

Try to exploit Android phones
Can we now hack Android phones due to missing patches?

<table>
<thead>
<tr>
<th>At first glance, Android phones look hackable</th>
<th>VS. Mobile operating systems are inherently difficult to exploit</th>
</tr>
</thead>
<tbody>
<tr>
<td>- We find that most phones miss patches within their patch level</td>
<td>- Modern exploit mitigation techniques increase hacking effort</td>
</tr>
<tr>
<td>- While the number of open CVEs can be smaller than the number of missing patches, we expect some vulnerabilities to be open</td>
<td>- Mobile OSs explicitly distrust applications through sandboxing, creating a second layer of defense</td>
</tr>
<tr>
<td>- Many CVEs talk of “code execution”, suggesting a hacking risk based on what we experience on Windows computers</td>
<td>- Bug bounties and Pwn2Own offer relatively high bounties for full Android exploitation</td>
</tr>
</tbody>
</table>
### Criminals generally use three different methods to compromise Android devices

<table>
<thead>
<tr>
<th>Approach</th>
<th>Social engineering</th>
<th>Local privilege escalation</th>
<th>Remote compromise</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Trick user into insecure actions:</strong></td>
<td>▪ Install malicious app</td>
<td>▪ Trick user into installing malicious app</td>
<td>▪ Exploit vulnerability in an outside-facing app (messenger, browser)</td>
</tr>
<tr>
<td>▪ Then grant permissions</td>
<td>▪ Then exploit kernel-level vulnerability to gain control over device, often using standard “rooting” tools</td>
<td></td>
<td>▪ Then use local privilege escalation</td>
</tr>
<tr>
<td>▪ Possibly request ‘device administrator’ role to hinder uninstallation</td>
<td>▪ Possibly request ‘device administrator’ role to hinder uninstallation</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Used for</strong></td>
<td><strong>Local privilege escalation</strong></td>
<td><strong>Remote compromise</strong></td>
<td></td>
</tr>
<tr>
<td>▪ Ransomware [File access permission]</td>
<td>▪ Targeted device compromise, e.g. FinFisher and Crysaor (Same company as infamous Pegasus malware)</td>
<td>▪ (Google bug bounty, Pwn2Own)</td>
<td></td>
</tr>
<tr>
<td>▪ 2FA hacks [SMS read]</td>
<td>▪ Advanced malware</td>
<td></td>
<td></td>
</tr>
<tr>
<td>▪ Premium SMS fraud [SMS send]</td>
<td>▪ Regular observed in advanced malware and spying</td>
<td>▪ Very few examples of recent criminal use</td>
<td></td>
</tr>
<tr>
<td><strong>Frequency in criminal activity</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▪ Almost all Android “Infections”</td>
<td>▪ Regular observed in advanced malware and spying</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Made harder through patching</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
An exploitable vulnerability implies a missing patch, but not the other way around

<table>
<thead>
<tr>
<th>Missing patches in source code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code parts that are ignored during compilation</td>
</tr>
<tr>
<td>Missed patches in binary</td>
</tr>
<tr>
<td>Vendor created alternative patch</td>
</tr>
<tr>
<td>Vulnerability requires a specific configuration</td>
</tr>
<tr>
<td>Bug is simply not exploitable</td>
</tr>
<tr>
<td>Errors in our heuristic (it happens!)</td>
</tr>
<tr>
<td>Open vulnerabilities</td>
</tr>
</tbody>
</table>

Diagram:

- Missing patches (source code analysis)
- Missing patches (binary analysis)
- Open vulnerabilities
A single Android bug is almost certainly not enough for exploitation

Android remote code execution is a multi-step process

1. **Information leakage** is used to derive ASLR memory offset (alternatively for 32-bit binaries, this offset can possibly be brute-forces)

2. **Corrupt memory** in an application. Examples:
   - Malicious video file corrupts memory using Stagefright bug
   - Malicious web site leverages Webkit vulnerability

   ➢ This gives an attacker control of the application including the apps access permission

3-4. Do the same again with two more bugs to gain access to system context or kernel

   ➢ This gives an attacker all possible permissions (system context), or full control over the device (kernel)

Simplified exploit chain examples with 4 bugs

Aside from exploiting MC and IL programming bugs, Android has experienced logic bugs that can enable alternative, often shorter, exploit chains
Remotely hacking a modern Android device usually requires chains of bugs.

**Famed real-world exploit examples**
- **Stagefright [2015]** Android < 5.1.1
- **Return to libstagefright [2016]** Android < 7.0
- **BlueBorne [2017]** Android < 8.0
- **Pixel - Nexus 6P [2017]** Chrome Android prior 54.0.2840.90
- **Pixel [2018]** Chrome Android prior 61.0.3163.79

**Weakness severities**
- Critical
- High
- Moderate

**Weakness classes**
- **DH** Data handling errors (CWE-19) e.g. buffer errors, input validation mistakes
- **SF** Security features gaps (CWE-254) e.g. permission errors, privileges mishandling, access control errors
- **TS** Time and state errors (CWE-361) e.g. race conditions, incorrect type conversions or casting

**Remote attacker**

1. Stagefright [2015] Android < 5.1.1
2. DH
3. System context protection mechanisms (e.g. ASLR, sandbox)
4. High privileged domain (e.g. system-server, Bluetooth)

**Application context protection mechanism**
(e.g. ASLR, sandbox)

**Step 1: Remote Code Execution and Information disclosure**
In many cases, one critical or high-severity weakness is exploited to allow for Remote Code Execution (RCE). (In the special case of BlueBorne, no sandbox exists.)

**Step 2: Escalation of Privilege**
At least one other weakness (or the users themselves) helps the attacker overcome protection mechanisms and gain access to higher privileges.
In case you want to dive deeper: More details on well-documented Android exploit chains

1. Heap pointer leak to bypass ASLR protection
2. ROP execution in mediaserver process
3. Module pointer leak to get address of executable code
4. Call mprotect to get RCE into privileged system-server domain

BlueBorne is a vulnerability in the Android Bluedroid/Fluorid userland stack, which is already a high-privileged domain.

Content view client in Chrome allowed arbitrary intent scheme opening, which allows escaping the Chrome sandbox.

Attacker perform arbitrary read/write operations leading to code execution based on incorrect optimization assumption in Chrome v8.

Open intent controlled URL in Google Drive to get shell in untrusted app context.

Exploit chain does not include break-out of untrusted app context.

BlueBorne 2017

Pixel / Nexus 6P 2017

Pixel 2018

Security Research Labs
SnoopSnitch version 2.0 introduces patch analysis for all Android users

**Tool name**

SnoopSnitch

**Purpose**

- [new in 2.0] Detect potentially missing Android security patches
- Collect network traces on Android phone and analyze for abuse
- Optionally, upload network traces to GSMmap for further analysis

**Requirements**

- Android version 5.0
- Patch level analysis: All phones incl. **non-rooted**
- Network attack monitoring: Rooted Qualcomm-based phone

**Source**

Search: SnoopSnitch
Take aways

- Android patching is more complicated and less reliable than a single patch date may suggest
- Remote Android exploitation is also more much complicated than commonly thought
- You can finally check your own patch level thanks to binary-only analysis, and the app SnoopSnitch

Many thanks to Ben Schlabs, Stephan Zeisberg, Jonas Schmid, Mark Carney, Luas Euler, and Patrick Lucey!

Questions?

Jakob Lell  <jakob@srlabs.de>
Karsten Nohl  <nohl@srlabs.de>
References

