Fuzzing Android: a recipe for uncovering vulnerabilities inside system components in Android

Alexandru Blanda
Intel OTC Romania, Security SQE
ioan-alexandru.blanda@intel.com

Abstract

The paper focuses on a fuzzing approach that can be used to uncover different types of vulnerabilities inside multiple core system components of the Android OS. The paper will introduce the general idea behind this approach and how it applies to several real-life targets from the Android OS backed up by discovered vulnerabilities. The list of components that were targeted and found vulnerable contains: the Stagefright framework, the mediaserver process, the Android APK install process, the installd daemon, dex2oat, ART.

A number of topics will be covered, starting with the actual fuzzing process with the data/seed generation processes and test case execution, the logging and triage mechanisms, addressing challenges such as bug reproducibility, sorting out unique issues and prioritizing issues based on their severity. The second part of the paper will take the discussion towards explaining the creation of several tools that have been developed using this methodology. The actual implementation of the tools will be discussed with focus on the technical details, as well as the issues discovered, CVE entries released and possible exploitable patterns.

Brief introduction to fuzzing

This section of the paper describes what fuzz testing is and offers a very brief insight on how fuzzing works. Fuzzing can be considered, and it is often described as being a black-box software testing technique. At a very general level, a definition of fuzzing can be summed up as being the process of sending random or invalid data as input to a system, with the purpose of crashing the system and revealing possible security vulnerabilities or reliability problems. So, in other words, the purpose of fuzz testing is to find security-related problems or any other critical defects that could lead to an undesired behavior of the system, such as denial of service or degradation of service. Special types of programs or frameworks are used to achieve this objective. These tools are commonly referred to as fuzzers and in the last 10 to 15 years, as fuzzing has gradually developed, they have gained popularity among software security experts and quality assurance communities. Manually creating tests for exploring every combination of data, in order to create suitable test cases would prove to be an impossible task even for medium-complexity applications. Therefore, fuzz testing seems to be the logical solution, particularly because building a simple fuzzing tool is in many cases a trivial task. Intelligent fuzzing, that uses knowledge of the structure or logic of the system that is being tested has proven to be even more effective, and when dealing with complex

protocols or applications this is the only approach considered acceptable. Fuzzing can be used to test any type of piece of software that accepts some sort of input, no matter the programming language that was used for developing it. However, it is most suitable for testing code that was written in C/C++, mainly because it handles its own memory, meaning that fuzzing can bring up exploitable security threats. As we will see in the following sections of the paper, for the task of fuzzing different system components of Android, intelligent fuzzing approaches were combined with straight-forward dumb-fuzzing methods to obtain the desired results.

A fuzzing approach in Android

This section of the paper will go through a set of basic concepts and methods that can be used in fuzzing campaigns that target OS components in an Android environment. The topics that will be covered in this part relate to general ways in which a security researcher can go through the various phases of running a fuzzing campaign on Android.

Data generation

Fuzzing can often be classified as mutational or generational. Mutational fuzzing mainly refers to the fact that we take an initial valid input and apply different types of mutations before testing it against the target system. Generational fuzzing, on the other hand, refers to the process of creating the input from scratch taking into account the specific format of the type of input. Both approaches have advantages in certain situations.

In this section of the paper I will provide a list of several open-source fuzzing tools have been used for the projects that will be detailed in the following sections, with a brief description of each:

- Basic fuzzing framework (BFF) – mutational fuzzer targeting software that consumes file input
- Zzuf – application input fuzzer
- Radamsa – general purpose test case generator for fuzzing
- Fuzzbox – fuzzing tool specialized in targeting media codecs
- American Fuzzy Lop (AFL) – instrumentation driven file format fuzzer

The way the malformed data is executed on the device, varies across projects and depends greatly on system component we are targeting. This topic will be covered in the section detailing how the fuzzing approach was applied for specific projects.

Logging process

The Android system provides a method for collecting the system debugging information. The logcat command enables collecting various information from applications and other components of the system into a circular buffer that can be viewed and more importantly, filtered. There are 7 types of priorities for messages that end up being displayed using the logcat command: verbose, debug, info, warning, error, fatal and silent. This information is necessary to understand the way each malformed input can be logged on a targeted Android device.

Using the “log” Android shell command, a user can construct messages with various priorities that can be artificially inserted into the logcat buffer. In the case of this fuzzing approach, each corrupt input that is tested against a system component on an Android device, is artificially logged with a fatal priority, so if an

actual fatal message would appear, the input causing that fatal message would be displayed immediately above it. This enables us to pinpoint which input was responsible for triggering a certain issue. Below is an example on how to get this type of log:

```
$ adb shell logcat –v time *:F

01-16 17:46:12.240 F/<Component> (PID):  <test_case_index> *** <reproducibility_info>
01-16 17:46:19.676 F/<Component> (PID):  <test_case_index> *** <reproducibility_info>
01-16 17:46:24.328 F/<Component> (PID):  <test_case_index> *** <reproducibility_info>
17:46:24.405 F/libc (8321): Fatal signal 11 (SIGSEGV) at 0x18 (code=1), thread 831
(process_name)
01-16 17:46:25.128 F/<Component> (PID):  <test_case_index> *** <reproducibility_info>
01-16 17:46:55.933 F/<Component> (PID):  <test_case_index> *** <reproducibility_info>
```

As mentioned earlier, to insert a message with a predefined priority in the logcat buffer we use the log command:

```
$ adb shell log  -p F –t <Component> <test_case_index> *** <reproducibility_info>
```

It is important to note that the messages that will be inserted into the log buffer need to contain information regarding ways to reproduce the issues that may occur during testing. The type of information that should be contained will be better explained in the following sections that contain actual examples of logs.

Triage mechanism

The role of the triage mechanism is to quickly sort out the unique issues that may occur after a fuzzing campaign that generates a large number of crashes. A tombstone is a file generated in /data/tombstones, on the device, after each system crash and contains the backtrace for the crash, the signal that caused the issue and other useful information extracted from the logcat message buffer. It also contains the program counter (PC) at which the crash occurred inside the affected component. Each crash can be uniquely identified using the PC value of the crash. For a better understanding, please see below an extract of a tombstone that was generated by a malformed media file:

```
pid: 3438, tid: 3438, name: stagefright  >>> stagefright <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr deadbaad
Abort message: 'invalid address or address of corrupt block 0x8004d748 passed to
dlfree'
    eax b3ee0ff8  ebx b7b18f38  ecx b7b1d900  edx b3ee0ff8
    esi 8004d748  edi af6d4dee
    xcs 00000073  xds 0000007b  xes 0000007b  xfs 00000000  xss 0000007b
    eip b7a7202c  ebp bffff418  esp bffff3d0  flags 00010286

backtrace:
    #00  pc 0001402c  /system/lib/libc.so (dlfree+1948)
    #01  pc 0000d630  /system/lib/libc.so (free+32)
    #02  pc 000dcf1c  /system/lib/libstagefright.so
(android::MediaBuffer::~MediaBuffer()+108)
    #03  pc 000dd6eb  /system/lib/libstagefright.so
(android::MediaBuffer::release()+267)
    #04  pc 000ddf7b  /system/lib/libstagefright.so
(android::MediaBufferGroup::~MediaBufferGroup()+187)
```

The general idea behind the implementation of the triage mechanism can be observed below:

```
1. Parse the logs and identify the input that caused a crash
2. Re-test using the faulty input
3. For each input tested:
       a. Grab the generated tombstone
       b. Parse the tombstone and get the PC value
       c. Check to see if the PC value has been previously encountered
       d. Save the tombstone and the test case (reproducibility info) if the issue
          is new
```

Analyzing and debugging crashes in Android

There are a number of methods and tools that can be used for figuring out what is happening with a system crash in Android.

The most relevant information contained in a tombstone is the stack trace for the crash, the signal that caused the crash, the code for the signal, the fault address and the program counter value (PC). For example for a segmentation fault signal we can have a code 1 (SEGV_MAPERR), which means that the address is not mapped to the object or code 2 (SEGV_ACCERR), which means we have invalid permissions for the mapped object.

Dmesg is another option for debugging crashes on Android. It is a common command in Unix-like operating systems that prints the message buffer of the kernel. Below is an example of 2 messages generated by crashes that affected the Stagefright media framework in Android. The messages are related to a user-mode read resulting in no page being found and a user-mode write resulting in no page being found.

```
<6>[73801.130320] stagefright[12469]: segfault at 14 ip 00000000f72a5fff sp
00000000fff98710 error 4 in libstagefright.so[f71c6000+1b5000]

<6>[73794.579462] stagefright[12455]: segfault at c ip 00000000f728bcfe sp
00000000ff9d6f90 error 6 in libstagefright.so[f71e8000+1b5000]
```

Below you can find an explanation on how to translate the error codes for the messages in dmesg:

```
/*
 * Page fault error code bits:
 *
 *   bit 0 ==    0: no page found              1: protection fault
 *   bit 1 ==    0: read access                1: write access
 *   bit 2 ==    0: kernel-mode access         1: user-mode access
 *   bit 3 ==                                  1: use of reserved bit detected
 *   bit 4 ==                                  1: fault was an instruction fetch
 */
```

GDB can also be used in an Android environment as a more reliable debugging solution. To accomplish this you need to set up GDBserver on the device. This can be done by attaching GDBserver to the process that needs to be debugged by providing the PID of that process or directly calling the binary that needs to be debugged using GDBserver.

```
$ gdbserver :5039 --attach <process_pid>
    OR
$ gdbserver :5039 /path/to/executable <options> (ex: gdbserver :5039
/system/bin/stagefright -a file.mp3)
```

The Android device should be connected to a local machine where you run GDB as follows:

```
$ adb forward tcp:5039 tcp:5039
$ gdb
    (gdb) target remote :5039 (from the gdb shell)
    (gdb) continue (to resume process execution)
```

Additionally, to load the symbols for the shared libraries you can use:

```
(gdb) set solib-absolute-prefixdb
/path/to/tree/out/target/product/<product_id>/symbols/

(gdb) set solib-search-path
/path/to/tree/out/target/product/<product_id>/symbols/system/lib/
```

When you have access to the source code of the component you are fuzzing, it may be useful if you could link a crash to an actual section of the code, in order to better understand what is happening. The easiest way to achieve this is by using the *addr2line* Linux command. You need to get the program counter value from the tombstone file that was generated when the crash occurred and pass it to the *addr2line* command. The following example uses the PC value from the tombstone example shown in a previous section and should return the source code file and line number where the crash occurs.

```
$ addr2line -f -e
/path/to/tree/out/target/product/<product_id>/symbols/system/lib/libstagefright.so
000dcf1c
```

Fuzzing the media Framework in Android

The main idea behind this project is to create corrupt but structurally valid media files, direct them to the appropriate software components in Android to be decoded and/or played and monitor the system for potential issues (i.e. system crashes) that may lead to exploitable vulnerabilities. Custom developed Python scripts are used to send the malformed data across a distributed infrastructure of Android devices, log the findings and monitor for possible issues, in an automated manner. The actual decoding of the media files on the Android devices is done using the *Stagefright* command line interface. The results are sorted out, in an attempt to find only the unique issues, using a custom built triage mechanism.

Audio and video as attack vectors

There are a number of reasons why audio and video can be considered as very attractive attack vectors, both on systems such as desktop and laptop computers and on mobile devices. Since one of the main purpose of this paper is to present the steps taken to perform a fuzzing campaign on the media framework of the Android OS, the focus will be on the reasons that make these attack vectors attractive on mobile devices:

- The file formats involved in playing media content are binary streams that contain complex data. The parsing of this data by specialized components in the media framework can often result in memory corruption issues or other related problems.
- The large variety of different audio and video players, each one possibly using different codecs and plugins create an attractive habitat for potential attackers.
- The user's perception that audio and video files are harmless can have a decisive role, as they will usually not have second thoughts when downloading and playing media content from untrusted sources.
- Playing audio or video streams can be invoked without the user's explicit consent (i.e. when the media file is played inside a web page, media file is sent through a MMS)

The Stagefright framework

The Stagefright framework is responsible for the algorithmic logic of the media parsing system in Android. The general architecture of this framework can be observed in the following figure:

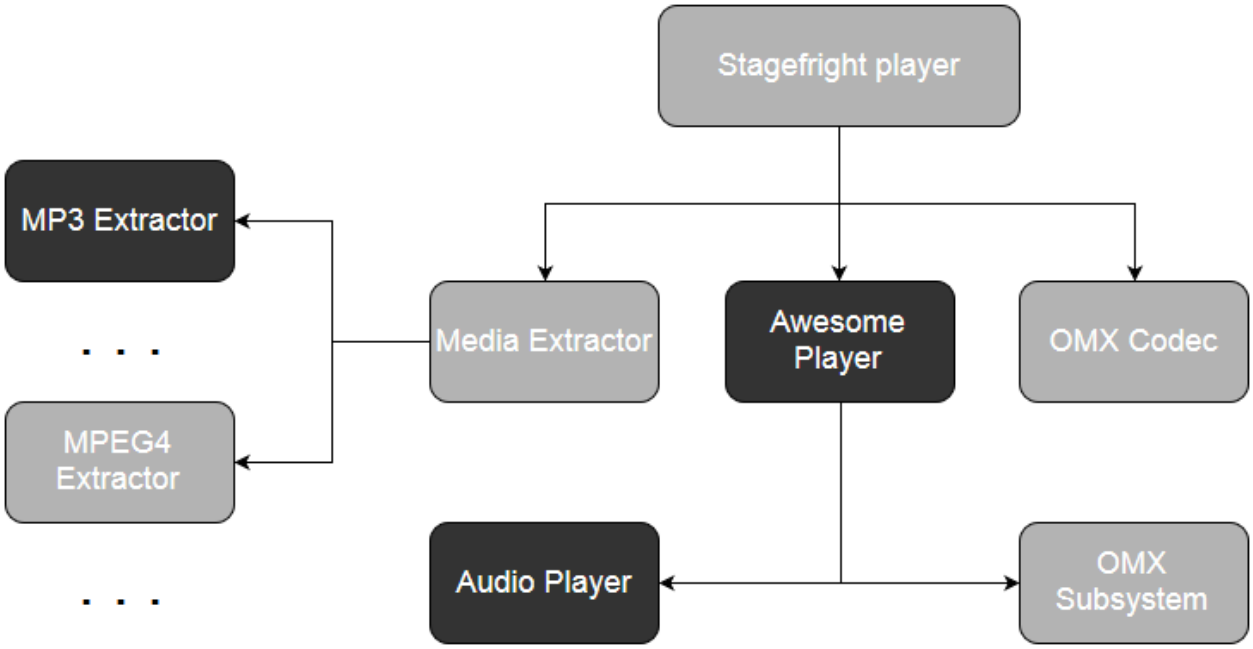

Figure 1. Stagefright media framework

The Stagefright player is only a client to the real media player: AwesomePlayer. It is this component that implements the functionalities of connecting video and audio caption sources with the corresponding decoders, playing the media files required by the user and synchronizing video with audio captions. The MediaExtractor component calls for the appropriate data parsers given the media file types that need to be read (i.e MP3, MPEG4). Finally, to prepare for playback, AwesomePlayer uses the OMXCodec component, in order to set up the decoders to use for each data source.

Executing the fuzzing campaign

The *stagefright* command line interface is used to actually decode each malformed media file on the tested Android devices. It can be custom built from the Android tree to be included in any Android image that is flashed on a device, and has the main capabilities to decode/encode a media file, force the usage of a software or hardware codec and a playback functionality for audio files. The full list of capabilities for this tool are listed below:

```
root@android:/ # stagefright -h
usage: stagefright
-h(elp)
-a(udio)
-n repetitions
-l(ist) components
-m max-number-of-frames-to-decode in each pass
-p(rofiles) dump decoder profiles supported
-t(humbnail) extract video thumbnail or album art
-s(oftware) prefer software codec
-r(hardware) force to use hardware codec
-o playback audio
-w(rite) filename (write to .mp4 file)
-x display a histogram of decoding times/fps (video only)
-S allocate buffers from a surface
-T allocate buffers from a surface texture
-d(ump) filename (raw stream data to a file)
-D(ump) filename (decoded PCM data to a file)
```

The actual fuzzing of the initial media files is done locally on the machine which connects the Android devices that are being tested. This process is regularly done using the Basic Fuzzing Framework, zzuf or Fuzzbox, which are open-source straight-forward to use tools. The malformed files are then sent to the device, where they are decoded using the *stagefright* command line interface. During the fuzzing campaign logs are generated using the format specified in the previous section. Below is an example of this type of log collected during a fuzzing campaign targeting the Stagefright framework with the media files mutated using BFF:

```
04-14 05:02:07.698 F/Stagefright(20222): - sp_stagefright *** 958 -
Filename:zzuf.32732.c8jZzT.mp4
04-14 05:02:13.382 F/Stagefright(20255): - sp_stagefright *** 959 -
Filename:zzuf.26772.zh7c8g.mkv
04-14 05:02:13.527 F/libc    (20256): Fatal signal 11 (SIGSEGV), code 1, fault addr 0x0
in tid 20256 (stagefright)
04-14 05:02:20.820 F/Stagefright(20270): - sp_stagefright *** 960 -
Filename:zzuf.12260.ayDuIA.mpg
04-14 05:02:21.259 F/Stagefright(20281): - sp_stagefright *** 961 -
Filename:zzuf.6488.F8drye.mp4
```

Fuzzing the application install process in Android

For this particular project, fuzzing has been used to modify different components of an APK (Android Application Package) using multiple approaches and check how this affects the install process in Android.

The main fuzzing targets inside an APK are the compiled app code which is represented by the *classes.dex* file packaged inside the APK and the *AndroidManifest.xml* file.

The fact that the install process in Android, more specifically the *installd* process, runs with high system privileges, made this component a very attractive target for fuzzing since any issues found during testing can have a greater impact from a system security perspective.

This section will offer an overview of the Android processes that are being used inside the project. Furthermore, the section offers an insight of the two different approaches that were used for the two Android versions that were taken into consideration: KitKat and Lollipop. The actual fuzzing process will be treated as a separate topic.

Overview of the application install process in Android

PackageInstaller is the default application responsible for installing other applications in Android. Package installer calls for the *InstallAppProgress* activity to receive the instructions from the user. This activity calls the Package Manager Service to install the package using the *installd* daemon which runs with system privileges and has the main functionality of receiving requests from the Package Manager Service. Going through the commands called during the installation of an application the most interesting from a fuzzing perspective are the run_dexopt(KitKat) and run_dex2oat(ART) methods which call for the *dexopt* and *dex2oat* command line binaries on the device. These will be the main targets for the fuzzing campaigns related to this project.

The main entry point in the system is considered to be the APK that allows an application to be installed on an Android based device. The 4 components that are of interest to us are: the *classes.dex*, the manifest file, the META-INF folder and the optional lib folder. The *classes.dex* and manifest file are important since these can be considered as the main fuzzing targets. The META-INF folder contains the signing information for a specific APK, and because after modifying and repackaging an APK it is mandatory to resign the application so it can be installed on a device this is a critical component. The optional lib folder that contains the compiled native Android libraries specific for a certain application can be considered as an alternative attack vector. The components of a generic APK are listed in the figure below.

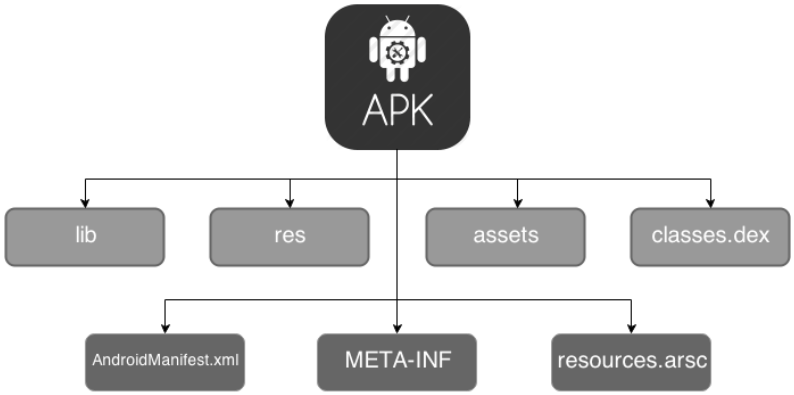

Figure 2. APK components

Code that is written for the Android platform is compiled into .dex (Dalvik Executable) files that are in turn zipped into an APK file. Dex files act in a similar manner as .class java files, but are used inside a virtual machine.

The following image depicts the general format of a .dex file.

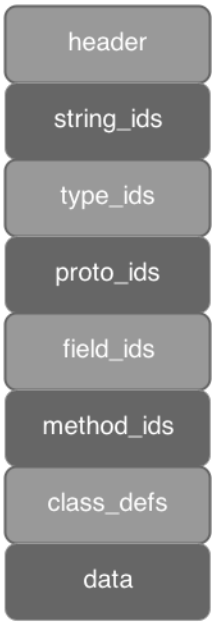

Figure 3. DEX file sections

The main fields composing a dex file, not taking into consideration the header which will be explained more into detail later on, are as follows :

- string_ids – the string identifiers list contains identifiers for all the strings used in the .dex file. This table stores the length and offsets for each of these strings.
- type_ids – contains identifiers for for all types (classes, arrays, or primitive types) included in this dex file. (i.e int, String[], com.google.Type)
- proto_ids – identifiers for all prototypes referred to by this dex file (i.e int fn(double), void fn()).
- field_ids – stores the data for pre-initialized fields in a class (i.e integer.MAX_VALUE)
- method_ids – contains identifiers for all the methods referred to by the .dex file, that are defined in the file or not
- class_defs – contains the class definitions list for all classes that are either defined in this dex file or that have a method or field accessed by code from the .dex file
- data – this section contains the data needed for all the tables listed above.

The header section, which is of particular importance for this project given the fact that a tampered *dex* file header can determine the system to stop parsing the file at a very early stage of the install process, contains the following fields:

- Magic (8 bytes) – “dex\n\035\0”
- Checksum (4 bytes) – Adler32 checksum for the file, from bytes offset 12
- Signature (20 bytes) – SHA-1 of file, from bytes offset 32

- File size (4 bytes)
- Header size (4 bytes) – constant value “112”
- Endian tag (4 bytes) – constant value “78563412” or reverse
- Link_size (4 bytes) – size of link section
- Link_off (4 bytes) – offset of link section
- Map_off (4 bytes) – offset of map list
- String_ids_size (4 bytes) – count of strings in the string ID list
- String_ids_off (4 bytes) – file offset of string ID list
- Type_ids_size (4 bytes) – count of types in the type ID list
- Type_ids_off (4 bytes) – file offset of type ID list
- Proto_ids_size (4 bytes) – count of items in the method prototype ID list
- Proto_ids_off (4 bytes) – file offset of method prototype ID list
- Fields_ids_size (4 bytes) – count of items in the field ID list
- Fields_ids_off (4 bytes) – file offset of field ID list
- Method_ids_size (4 bytes) – count of items in the method ID list
- Method_ids_off (4 bytes) – file offset of method ID list
- Class_defs_size (4 bytes) – count of items in the class definitions list
- Class_defs_off (4 bytes) – file offset of class definitions list
- Data_size (4 bytes) – size of data section in bytes (this value is actually the size of the map section plus the actual data section size)
- Data_off (4 bytes) – file offset of data section

Executing the fuzzing campaign

Two separate approaches have been identified for the 2 OS versions taken into consideration: KitKat and Lollipop.

Android KitKat

Although dexopt can be called as a standalone binary from the shell of an Android device, it cannot be used for the fuzzing purposes of this project, given the fact that it requires a large number of arguments that can't be passed from a shell environment. The solution in this case was to use the regular process of an APK installation. So the idea was to take large sets of valid APKs, fuzz the classes.dex files inside each application, repackage the APK and then try to install it on a device, to check how this malformed input is handled by the system at install time.

These are the steps taken for each APK that is being installed on a target device:

1. Extract classes.dex file from seed APK
    - *unzip –d </local/path/> </apk/path/>*
2. Fuzz extracted dex file
    - *<fuzz> –s <seed> classes.dex > fuzzed.dex*
3. Remove original .dex file from initial APK
    - *aapt r <original_apk> classes.dex*
4. Repackage APK with fuzzed APK
    - *aapt a <original_apk> classes.dex*
5. Create local keystore

- *keytool -genkey -v -keystore keystore.keystore -alias keystore -keyalg RSA -keysize 2048 -validity 10000*

6. Remove META-INF directory from APK
    - *zip --delete </apk/path/> META-INF/**
7. Resign the APK using local keystore
    - *jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore </keystore/path> </apk/path> <keystore_alias*

The figure below represents an extract of a log produced during a fuzzing campaign ran against a number of Android devices running KitKat. The logging mechanism has been explained in detail in the first section of the paper:

```
06-26 17:43:05.568 F/dexopt (14769): - sp_lib.py - APK_id = com.imangi.templerun.apk
seed = radamsa -s 1927
06-26 17:43:29.732 F/dexopt (14881): - sp_lib.py - APK_id = com.imangi.templerun.apk
combination = radamsa -s 2086
06-26 17:43:54.620 F/dexopt (14988): - sp_lib.py - APK_id = com.imangi.templerun.apk
seed = radamsa -s 5011
06-26 17:44:19.763 F/dexopt (15105): - sp_lib.py - APK_id = com.imangi.templerun.apk
seed = radamsa -s 1543
06-26 17:44:43.524 F/dexopt (15215): - sp_lib.py - APK_id = com.imangi.templerun.apk
seed = radamsa -s 9090
06-26 17:44:44.079 F/libc    (15227): Fatal signal 11 (SIGSEGV) at 0xaa4c04f8 (code=1),
thread 15227 (mangi.templerun)
06-26 17:45:09.950 F/dexopt (15338): - sp_lib.py - APK_id = com.imangi.templerun.apk
seed = radamsa -s 8098
06-26 17:45:33.771 F/dexopt (15451): - sp_lib.py - APK_id = com.imangi.templerun.apk
seed = radamsa -s 1069
06-26 17:45:59.802 F/dexopt (15570): - sp_lib.py - APK_id = com.imangi.templerun.apk
seed = radamsa -s 8925
```

Android Lollipop

As opposed to dexopt, the dex2oat binary can be used as a standalone command line tool for the fuzzing purposes of this project. The command line binary requires to take a minimum of 2 parameters: an input dex file and the resulting oat file. An example of some of the more common usage options for the dex2oat binary can be observed below:

```
Usage: dex2oat [options]...
-j<number>: specifies the number of threads used for compilation.
--dex-file=<dex-file>: specifies a .dex file to compile.
--zip-fd=<file-descriptor>: specifies a file descriptor of a zip file
      containing a classes.dex file to compile.
--zip-location=<zip-location>: specifies a symbolic name for the file
--oat-file=<file.oat>: specifies the oat output destination via a filename.
--oat-fd=<number>: specifies the oat output destination via a file descriptor.
--oat-location=<oat-name>: specifies a symbolic name for the file corresponding
    to the file descriptor specified by --oat-fd.
...
...
...
```

The general idea in the case of fuzzing on Android Lollipop would be to have an initial set of .dex files that have been previously extracted from valid APK files. For each of the valid .dex files in the initial set, a number of fuzzed combinations will be generated using the Radamsa tool and then passed as input files

to the dex2oat binaries. In case of a crash, the .dex file responsible for the crash should be repackaged inside the parent APK with the purpose of checking if the found issues can be reproducible in a normal usage scenario, which is represented by the installation of the APK on a device.

An example of log generated after a fuzzing campaign on an Android device running the L version can be observed in the figure below:

```
09-29 11:32:20.460 F/dex2oat ( 8041): - sp_libd.py - dex_id = com.evernote.apk seed =
radamsa -s 1012528
09-29 11:32:33.405 F/dex2oat ( 8054): - sp_libd.py - dex_id = com.evernote.apk seed =
radamsa -s 6186726
09-29 11:32:46.277 F/dex2oat ( 8066): - sp_libd.py - dex_id = com.evernote.apk seed =
radamsa -s 7338683
09-29 11:32:49.121 F/libc    (15227): Fatal signal 11 (SIGSEGV) at 0xaa4c0302 (code=1),
thread 15227 (evernote)
09-29 11:32:57.249 F/dex2oat ( 8079): - sp_libd.py - dex_id = com.evernote.apk seed =
radamsa -s 231131
09-29 11:33:08.528 F/dex2oat ( 8093): - sp_libd.py - dex_id = com.evernote.apk seed =
radamsa -s 4456070
```

Actual fuzzing methods

The most challenging part of this project is related to the fact that the Android system employs a large number of verifications and checks against the .dex files that are sent to be installed on the device. Because of this reason several fuzzing methods have been tried to overcome this obstacle. Below are some examples of error messages that stand in the way of a successful fuzzing campaign:

```
01-03 13:24:13.511 I/dex2oat ( 5671): dex2oat --dex-file=test7.dex --oat-
file=output.oat
01-03 13:24:13.125 W/dex2oat ( 5671): Failed to open .dex from file 'test7.dex': verify
                   dex file 'test7.dex': Bad checksum (790931db, expected 745631bc)
01-03 13:24:13.115 E/dex2oat ( 5671): Failed to open some dex files: 1
01-03 13:24:13.447 I/dex2oat ( 5671): dex2oat took 255.693ms (threads: 4)

01-03 03:22:23.581 I/dex2oat ( 5671): dex2oat --dex-file=test7.dex --oat-
file=output.oat
01-03 03:22:23.635 W/dex2oat ( 5671): Failed to open .dex from file 'test7.dex': verify
                   dex file 'test7.dex': Bad file size (143221ab, expected 435611cd)
01-03 03:22:23.635 E/dex2oat ( 5671): Failed to open some dex files: 1
01-03 03:22:23.837 I/dex2oat ( 5671): dex2oat took 255.693ms (threads: 4)

01-03 04:21:13.181 I/dex2oat ( 5671): dex2oat --dex-file=test7.dex --oat-
file=output.oat
01-03 04:21:13.235 W/dex2oat ( 5671): Failed to open .dex from file 'test7.dex': verify
                   dex file 'test7.dex': Invalid header size (7f, expected 70)
01-03 04:21:13.641 E/dex2oat ( 5671): Failed to open some dex files: 1
01-03 04:21:13.857 I/dex2oat ( 5671): dex2oat took 255.693ms (threads: 4)
```

Regarding the actual fuzzing process, 3 alternatives have been identified and used for this project. These are as follows:

- Completely random fuzzing (applies to dex2oat)
- Random fuzzing and partial header reconstruction (applies to dex2oat & dexopt)
- Targeted fuzzing and complete header reconstruction (applies to dex2oat & dexopt)

Random fuzzing and partial header reconstruction

The general idea behind this particular approach can be summed up in the following 2 steps:

- Alter random parts of the .dex file, using the Radamsa tool in a deterministic manner, so in case a crash occurs the exact corruption sequence can be retraceable
- Do a best-effort approach to try to repair the fields from the file header, given the fact that the file was modified in a random fashion and little is known of the exact sections that were malformed, so that the dex file appears structurally valid to the system

The actual header fields that are being repaired in this alternative are on the one hand the fields that have a constant value and nothing needs to be computed: the magic, endian tag and header size fields and on the other hand the fields that can be recomputed, although we do not know any information about the fields that have been fuzzed: the file size, checksum and SHA-1 fields. A summary of these fields can be observed in the image below:

struct header_item dex_header		0h	70h	Dex file header
struct dex_magic magic	dex 035	0h	8h	Magic value
uint checksum	B3D20217h	8h	4h	Alder32 checksum of rest of file
SHA1 signature[20]	6DB8EDA774	Ch	14h	SHA-1 signature of rest of file
uint file_size	1430508	20h	4h	File size in bytes
uint header_size	112	24h	4h	Header size in bytes
uint endian_tag	12345678h	28h	4h	Endianness tag
uint link_size	0	2Ch	4h	Size of link section
uint link_off	0	30h	4h	File offset of link section
uint map_off	1430336	34h	4h	File offset of map list
uint string_ids_size	11029	38h	4h	Count of strings in the string ID list
uint string_ids_off	112	3Ch	4h	File offset of string ID list
uint type_ids_size	2068	40h	4h	Count of types in the type ID list
uint type_ids_off	44228	44h	4h	File offset of type ID list
uint proto_ids_size	2592	48h	4h	Count of items in the method prototype ID list
uint proto_ids_off	52500	4Ch	4h	File offset of method prototype ID list
uint field_ids_size	5335	50h	4h	Count of items in the field ID list
uint field_ids_off	83604	54h	4h	File offset of field ID list
uint method_ids_size	12925	58h	4h	Count of items in the method ID list
uint method_ids_off	126284	5Ch	4h	File offset of method ID list
uint class_defs_size	1427	60h	4h	Count of items in the class definitions list
uint class_defs_off	229684	64h	4h	File offset of class definitions list
uint data_size	1155160	68h	4h	Size of data section in bytes

Targeted fuzzing and complete header reconstruction

This is a similar approach to the previous one. The main difference is the fact that the only section of the .dex file that is being modified is the data section. The steps used for this approach are as follows:

- Split the original file in 3 parts: the map section, the data section and the initial remaining section that also contains the header of the file (using the information provided by the header: section sizes and offsets)
- Fuzz only the data section as a separate file using the Radamsa tool, in a deterministic manner
- Glue all the chunks back together

- Recompute all the fields that have incorrect values after this process. This means that all the fields that were modified for the previous alternative will be rewritten as well, but in addition the data_size and map_offset fields are recomputed and rewritten since in this case we know exactly what has been modified.

This approach has the advantage that it greatly increases the chances of the system perceiving the .dex file as structurally valid and therefore achieving a better code coverage for the testing campaign.

Completely random fuzzing

This alternative uses a slighter different approach that the previous two. The main idea would be to have an initial set of valid .dex files extracted from a number of APK files, which are used to generate malformed .dex files using the Basic Fuzzing Framework tool. The malformed input is then sent to the devices under test, without having the file header modified in any way.

Although it would be expected that the system to reject from the start the corrupt dex files, as not having the proper structure, this alternative made possible the discovery of a number of issues, as it can be seen in the results section of the paper.

Using American Fuzzy Lop in Android

The American Fuzzy Lop (AFL) tool is one of the most popular open-source fuzzing solutions for the Linux environment. It is an instrumentation based fuzzing tool developed by Michal Zalewski that can be used against binaries that consume different file formats as input. The target binaries need to be compiled with afl-gcc, in order to enable the instrumentation of the binaries. There are two fuzzing modes: dumb-mode (performs completely random fuzzing on the target) and instrumented mode (detects changes to program control flow to find new code paths; works only with binaries that have been instrumented during compile-time with afl-gcc). In both of these operating modes the tool detects both hangs and crashes that affect the targets and sorts out the unique issues.

The tool has been originally developed to run on desktop Linux environments such as Ubuntu and Debian, but we have been using an Android port of the tool. Thanks to Adrian Denkiewicz of Intel, who ported the tool we were able to run AFL directly on Android devices.

Using AFL for Stagefright fuzzing

One of the challenges encountered was to completely automate the usage of AFL using an infrastructure of Android devices. An overview of the steps taken on each device can be seen below:

```
1. Check device prerequisites
       1) Root
       2) Remount
       3) Push afl target binary
       4) Load initial seeds
       5) Set scalling governor
2. Eliminate crashing test cases from initial seeds on each device
       1) Run AFL in a loop with timeout
       2) Identify crashing test case and delete it from input folder
       3) Restart AFL with timeout -> if crash occurs goto 2) else goto 4)
       4) No crash occurred after the timeout -> AFL successfully started -> kill the
       process
3. Restart the AFL process with clean input directory and redirect output to /dev/null
```

However, given the large number of devices, there was the need to come up with a way to automate the usage of AFL on the entire infrastructure of devices. Some of the processes that needed to be covered were: generate and load the seeds that were consumed by AFL, run AFL processes on each device, retrieve the results and try to sort the unique issues. For confirming the results given by AFL and triaging the unique issues, all the crashes that were generated by AFL were passed through the MFFA tool and if they were confirmed as crashes they were sorted using the custom triage mechanism.

The initial seeds are generated using a functionality of ffmpeg. The format list was constructed in a way that covers all possible combinations between the codecs and containers specified in the Android Compatibility Definition Document (CDD). After the generation phase, the files are loaded onto the devices and the AFL processes are started in the manner described earlier. The results are collected by extracting the generated crashes and hangs on each device. To validate the issues, these preliminary results are passed as seeds for the MFFA tool that generates the logs that contain the crashes that actually reappeared. To uniquely identify the issues, the logs are sent to the triage mechanism.

This data flow can be observed in the following figure:

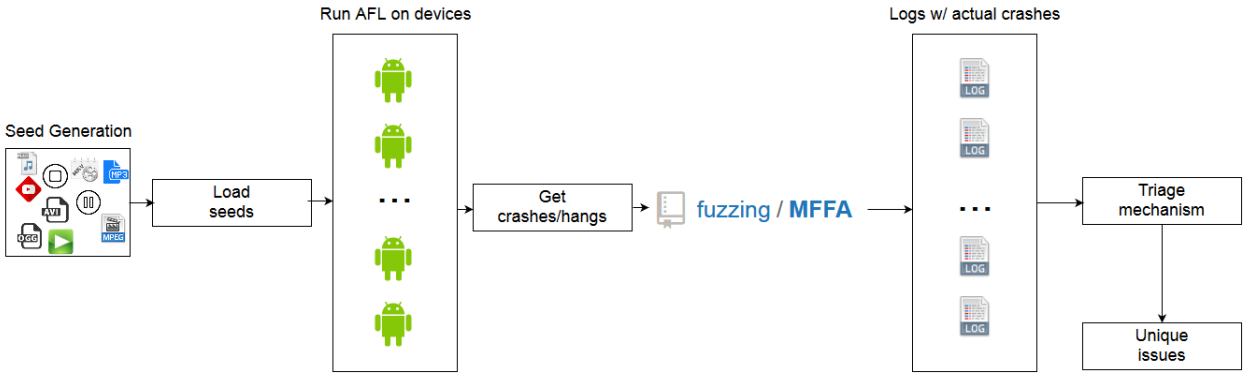

Figure 4. Automating AFL for Stagefright fuzzing

Results and conclusions

The first issues affecting the Stagefright media framework in Android were reported to Google in March 2014. These initial fuzzing campaigns generated an unexpected number of crashes. Their number was in the range of tens of thousands of crashes per week, on a cluster of less than 20 devices. This was the main reason that led to the development of the triage mechanism since manual sorting was not an option in this case. Out of the initial issues reported to Google, three were considered to be high severity. They were included in the Android Partner Security Bulletin from September 2014 and issued CVE numbers in November 2014 (CVE-2014-7915,CVE-2014-7916,CVE-2014-7917). These issues are all related to integer overflows that affect libstagefright. The fuzzing tool was open-sourced in February 2015 and is available on Github under a GNUv2 license. The tool is now being used as a complementary solution along with AFL.

The approach based on the usage of AFL led to the discovery of one critical issue: a heap corruption in libstagefright which can lead to arbitrary code execution in the mediaserver process. This was assigned CVE-2015-3832 and was listed in the first public Nexus security bulletin from August 2015. Several other low severity issues were discovered using this approach. These were mainly related to null-pointer-dereferences and integer division by zero situations and were reported to and fixed by Google.

The fuzzing campaigns targeting the application install process led to the discovery of one critical issue affecting the Lollipop version. In November 2015 CVE-2014-7918 was assigned for this issue. Several low priority issues affecting both KitKat and Lollipop were reported and fixed.

References

Media Fuzzing Framework for Android (MFFA) – https://github.com/fuzzing/MFFA

CVE-2014-7915 – http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-7915

CVE-2014-7916 – http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-7916

CVE-2014-7917 – http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-7917

CVE-2014-7918 – http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-7918

CVE-2015-3832 – http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3832

Nexus Security Bulletin (August 2015) – https://groups.google.com/forum/#!topic/android-security-updates/Ugvu3fi6RQM