A Whole New Efficient Fuzzing Strategy for Stagefright

Porting and Optimizations

by Zinuo Han

at Ruxcon 2017

About Me

Security Researcher at Chengdu Security Response Center, Qihoo 360

Focused on Android vulnerability research and exploit development

ele7enxxh@gmail.com

https://github.com/ele7enxxh

Agenda

Introduction

Design

Implementation

Disclosure

Conclusion

Introduction

Basic information about this talk

What is Stagefright

A logical algorithm framework library for parsing multimedia on Android

- Written in C++

Support a wide range of audio and video formats

- Including MP3, MP4, MKV, MPEG2, MPEG4, and many more

Lots of vulnerabilities have already been found in Stagefright

Why Stagefright again

Stagefright vulnerabilities keep attractive

- Continuously patched in every Android security update since August 2015

Most Stagefright vulnerabilities assessed as Critical

- This means higher bounty

Found vulnerabilities in multiple Stagefright components

- Libstagefright library, especially MPEG4Extractor.cpp
- OMX
- SW codecs(Most in 2017)

Why Stagefright again

Stagefright vulnerabilities keep attractive

- Continuously patched in every Android security update since August 2015

Most Stagefright vulnerabilities assessed as Critical

- This means higher bounty

Found vulnerabilities in multiple Stagefright components

- Libstagefright, especially MPEGExtractor.cpp
- OMX
- SW codecs(Most in 2017)

Still 0days

Related Work

Research	Foucsed on	Coverage-Guide	ASAN	Fast	Crash Tolerant
Fuzzing the Media Framework in Android(MFFA) by Alexandru Blanda at ELC 2015	libstagefright	✗	✗	-	✓

Related Work

Research	Foucsed on	Coverage -Guide	ASAN	Fast	Crash Tolerant
Fuzzing the Media Framework in Android(MFFA) by Alexandru Blanda at ELC 2015	libstagefright	✗	✗	-	✓
Stagefright: Scary Code in the Heart of Android by Joshua Drake at Blackhat USA 2015	MPEG4Extractor	✓	✓	+	✓

Related Work

Research	Foucsed on	Coverage -Guide	ASAN	Fast	Crash Tolerant
Fuzzing the Media Framework in Android(MFFA) by Alexandru Blanda at ELC 2015	libstagefright	✗	✗	-	✓
Stagefright: Scary Code in the Heart of Android by Joshua Drake at Blackhat USA 2015	libstagefright	✓	✓	+	✓
Fuzzing Android OMX by MingjianZhou and ChiachihWu at HITCON 2016	OMX	?	?	+	✓

About this talk

What will be talked about next

- How to design and implement a efficient fuzzing strategy for Stagefright
- What vulnerabilities did I find by the above method
- Conclusion of this talk

What will not be talked about next

- The root cause of the vulnerabilities
- Vulnerabilites exploitation

Design

Goal & Architecture overview

Goal

More targeted

- Mainly focus on SW codecs

Goal

More targeted

- Mainly focus on SW codecs

More faster

- Run Stagefright on desktop Linux
- Optimize Stagefright workflow

Goal

More targeted

- Mainly focus on SW codecs

More faster

- Run Stagefright on desktop Linux
- Optimize Stagefright workflow

More technical

- Coverage-Guided fuzzer(American Fuzzy Lop)
- AddressSanitizer

Goal

More targeted

- Mainly focus on SW codecs

More faster

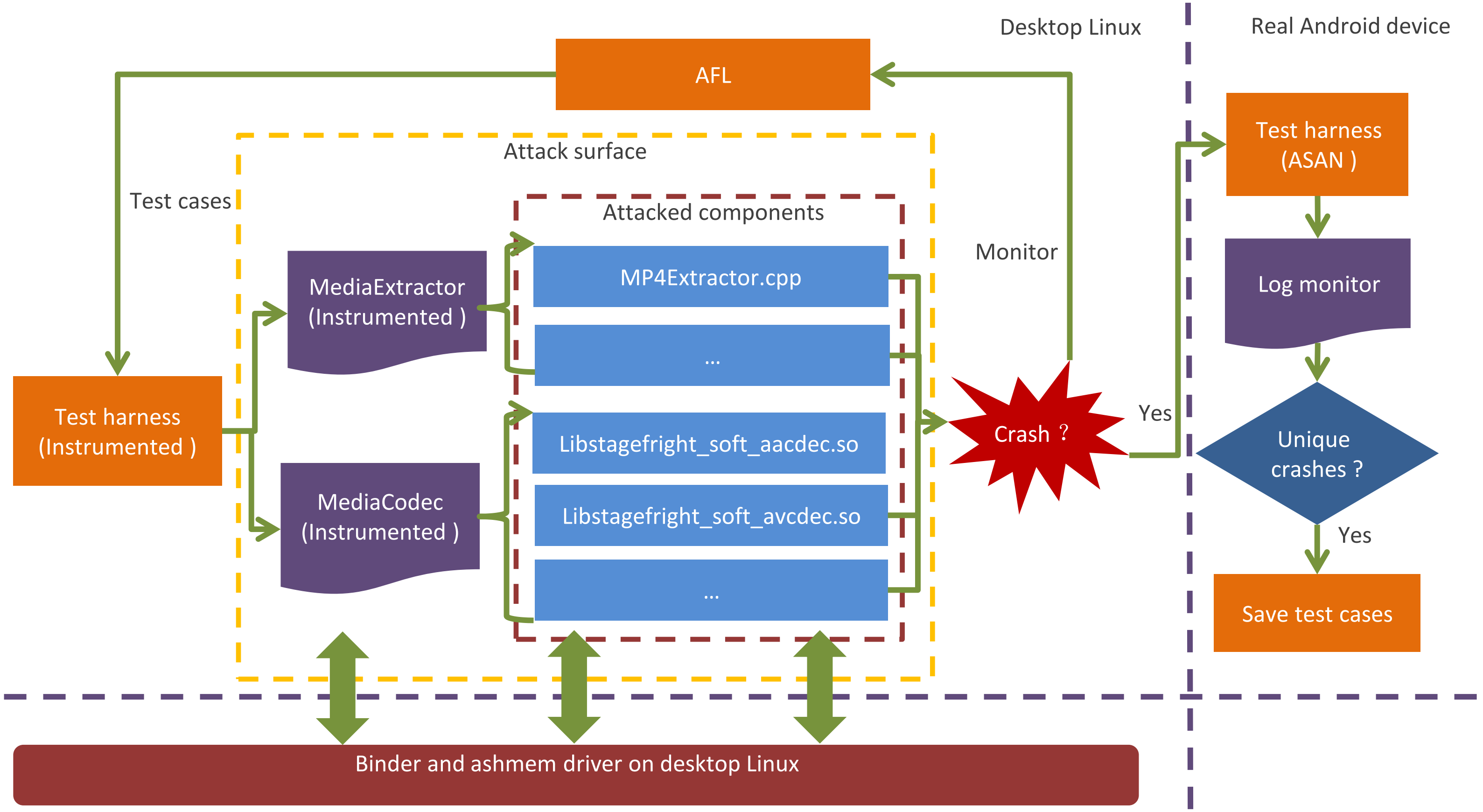
- Run Stagefright on desktop Linux
- Optimize Stagefright workflow

More technical

- Coverage-Guided fuzzer(American Fuzzy Lop)
- AddressSanitizer

Find 0days more easily

Architecture overview


Desktop Linux
Real Android device
AFL
Test cases
Attack surface
Attacked components
Monitor
MediaExtractor (Instrumented)
MP4Extractor.cpp
…
Test harness (Instrumented)
MediaCodec (Instrumented)
Libstagefright_soft_aacdec.so
Libstagefright_soft_avcdec.so
…
Crash ?
Yes
Test harness (ASAN)
Log monitor
Unique crashes ?
Yes
Save test cases
Binder and ashmem driver on desktop Linux

Implementation

Details of the fuzzing strategy

Steps in the fuzzing strategy

Find attack surface

Porting

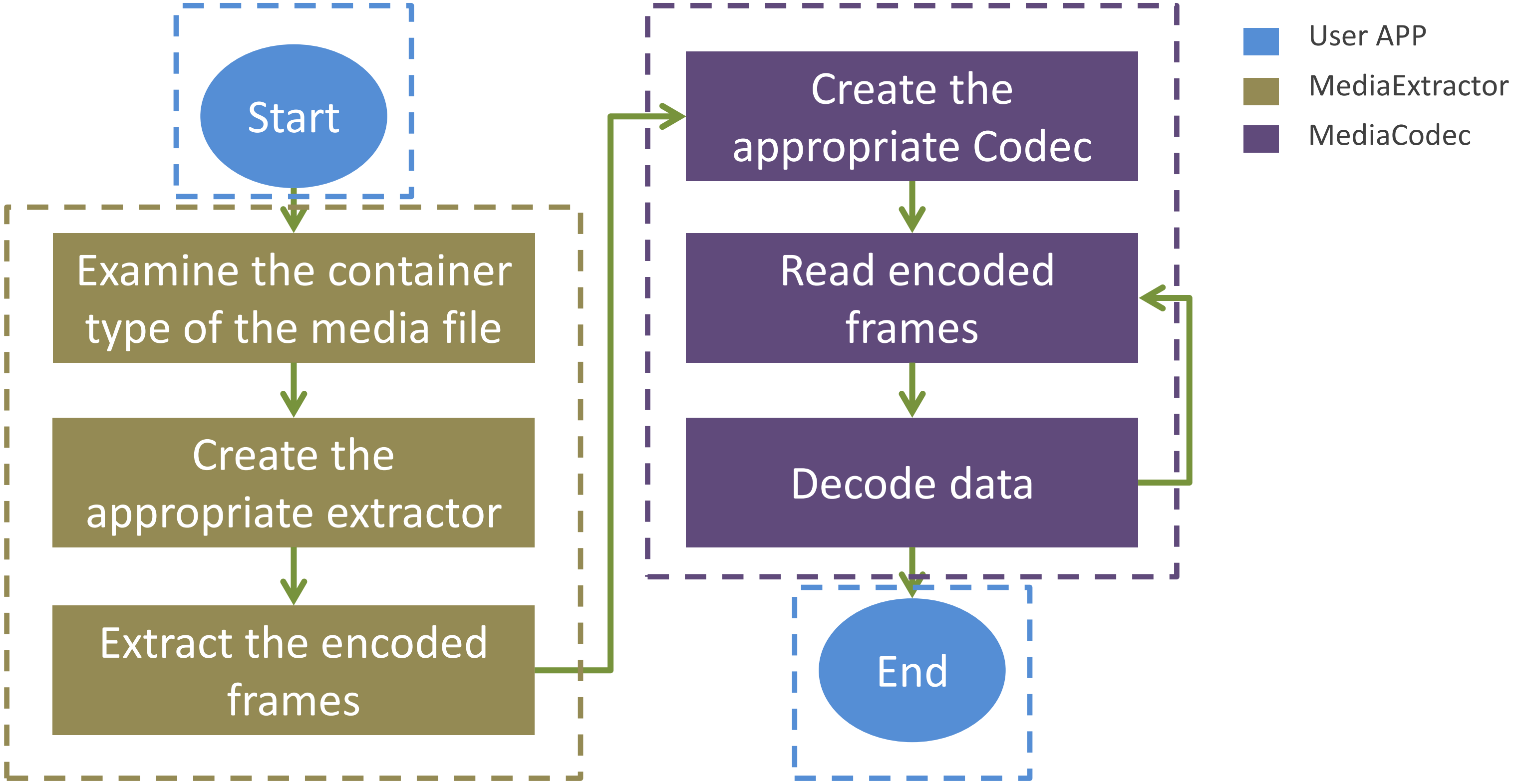
Optimizations

Get more powerful test cases

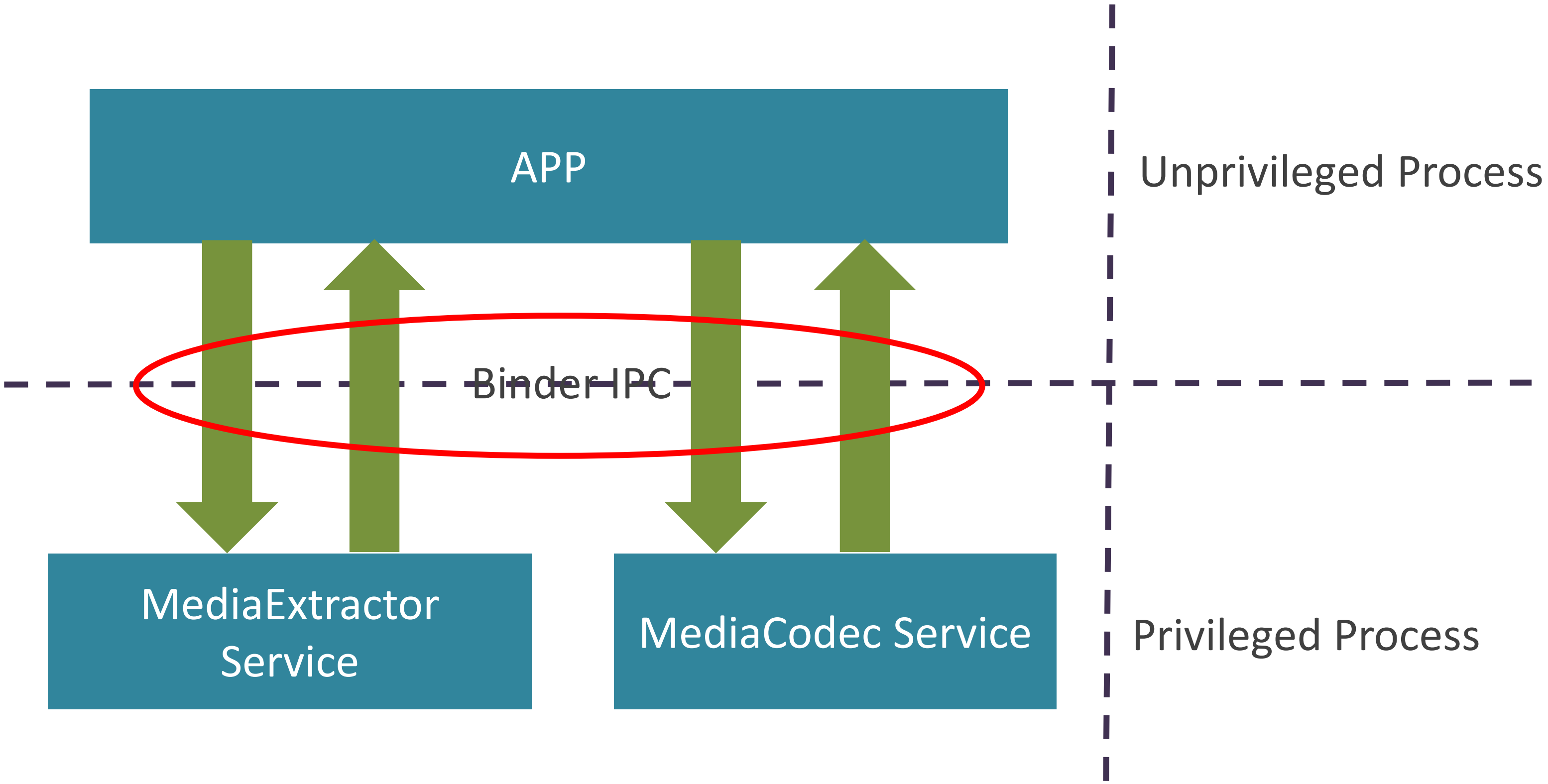
Fuzzing mp4 container

Recognize unique crashes

Find attack surface – How is audio and video played


Start
Examine the container type of the media file
Create the appropriate extractor
Extract the encoded frames
Create the appropriate Codec
Read encoded frames
Decode data
End
🟦 User APP
🟫 MediaExtractor
🟪 MediaCodec

Find attack surface – How is audio and video played



Find Attack Surface – Attacked components

Container Type	Module	Codec Type	ASAN
MP4	MPEG4Extractor.cpp	AAC	libstagefright_soft_aacdec.so
MP3	MP3EXtractor.cpp	AMRNB AMRWB	libstagefright_soft_amrdec.so
AMRNB AMRWB	AMRExtractor.cpp	H264	libstagefright_soft_avcdec.so
FLAC	FLACExtractor.cpp	HEVC	libstagefright_soft_hevcdec.so
WAV	WAVExtractor.cpp	G711	libstagefright_soft_g711dec.so
OGG	OGGExtractor.cpp	MPEG2	libstagefright_soft_mpeg2dec.so
MKV	MatroskaExtractor.cpp	H263 MPEG4	libstagefright_soft_mpeg4dec.so
MPEG2TS	MPEG2TSExtractor.cpp	MP3	libstagefright_soft_mp3dec.so
WVM	WVMExtractor.cpp	VORIBS	libstagefright_soft_vorbisdec.so
AAC	AACExtractor.cpp	OPUS	libstagefright_soft_opusdec.so
MPEG2PS	MPEG2PSExtractor.cpp	VP8 VP9	libstagefright_soft_vpxdec.so
MIDI	MIDIExtractor.cpp	GSM	libstagefright_soft_gsmdec.so

Find Attack Surface - Summary

Two potential attacked processes with privilege: mediaextracor and mediacodec

- Mediaextractor is used to extract audio and video frames
- Mediacodec is used to encode and decode audio and video frames

Full controllable input data

Complex input formats

- It means the possibility of more vulnerabilities

Easy to trigger

- No special permissions required

Porting – What and why

Port Stagefright to x86

- Android device most likely uses the ARM architecture, but the x86 family of processors is used in most desktop

Port binder and ashmem driver to linux

- Stagefright works with binder and ashmem driver, which are not enabled in desktop Linux as default

Port AFL to Android toolchains

- The shmat() function is used in the afl-llvm-rt.o.c, however Android toolchains can’t recognize it

Setup running environment

Porting – What and why

Port Stagefright to x86

- Android device most likely uses the ARM architecture, but the x86 family of processors is used in most desktop

Port binder and ashmem driver to linux

- Stagefright works with binder and ashmem driver, which are not enabled in desktop Linux as default

Port AFL to Android toolchains

- The shmat() function is used in the afl-llvm-rt.o.c, however Android toolchains can’t recognize it

Setup running environment

Make it efficient

Porting – Port Stagefright to x86

Download AOSP code from: https://android.googlesource.com/

Build Android for x86

```
$ cd aosp
$ source build/envsetup.sh
$ lunch aosp_x86-eng
$ make -j64
```

Build Stagefright to x86

```
$ cd framework/av/cmds/stagefright
$ make -j64
```

Porting – Port binder and ashmem driver to Linux

Download the latest version of the Linux kernel from: http://www.kernel.org

Enable the binder and ashmem driver

- e.g. (Linux kernel version 4.8.17)

  ```
  CONFIG_ANDROID=y && CONFIG_ANDROID_BINDER_IPC=y && CONFIG_ASHMEM=y
  ```

Add new udev rules to set correct permissions

```
$ echo -e "KERNEL==\"binder\",
MODE=\"0666\"\nKERNEL==\"ashmem\", MODE=\"0666\"" |
sudo tee /etc/udev/rules.d/android.rules
```

Porting – Port AFL to Android toolchains

Use syscall() function instead of shmat() function in afl-llvm-rt.o.c

```
- shmat(shm_id, NULL, 0);
+ syscall(SYS_ipc, IPCOP_shmat, id, flag, &addr, addr);
```

Force the compile wrapper(afl-clang-fast) to instrument the shared libraries

```
- if (!strcmp(cur, "-shared")) maybe_linking = 0;
```

Cross-compile afl-llvm-rt.o which should be linked into the target Android binary

Porting – Setup running environment

Create a soft link

```
$ ln -s out/system /system
```

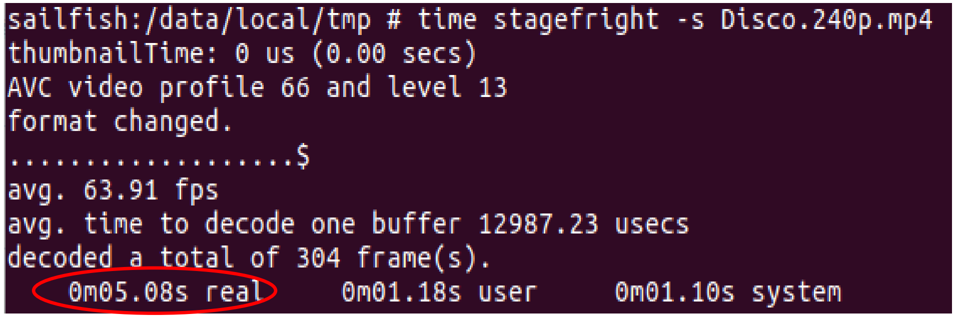
Copy configuration files

```
$ cp out/system/etc/media_codecs_google_audio.xml /etc
$ cp out/system/etc/media_codecs_google_telephony.xml /etc
$ cp out/system/etc/media_codecs_google_video.xml /etc
$ cp out/system/etc/media_codecs.xml /etc
```

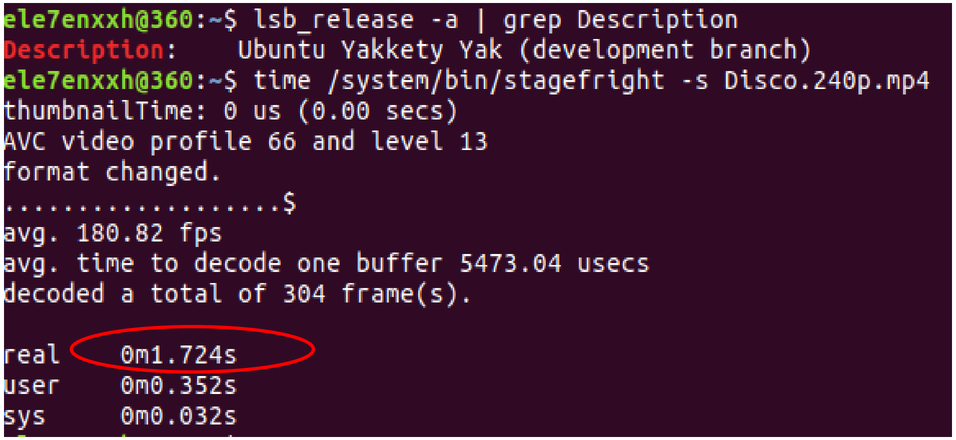
Startup dependency services: e.g. mediaextractor, mediacodec...

Porting – It works

Running 5x times faster



```
sailfish:/data/local/tmp # time stagefright -s Disco.240p.mp4
thumbnailTime: 0 us (0.00 secs)
AVC video profile 66 and level 13
format changed.
...................$
avg. 63.91 fps
avg. time to decode one buffer 12987.23 usecs
decoded a total of 304 frame(s).
    0m05.08s real       0m01.18s user       0m01.10s system
```



```
ele7enxxh@360:~$ lsb_release -a | grep Description
Description:    Ubuntu Yakkety Yak (development branch)
ele7enxxh@360:~$ time /system/bin/stagefright -s Disco.240p.mp4
thumbnailTime: 0 us (0.00 secs)
AVC video profile 66 and level 13
format changed.
...................$
avg. 180.82 fps
avg. time to decode one buffer 5473.04 usecs
decoded a total of 304 frame(s).

real    0m1.724s
user    0m0.352s
sys     0m0.032s
```

Porting – what else

Make it more efficient

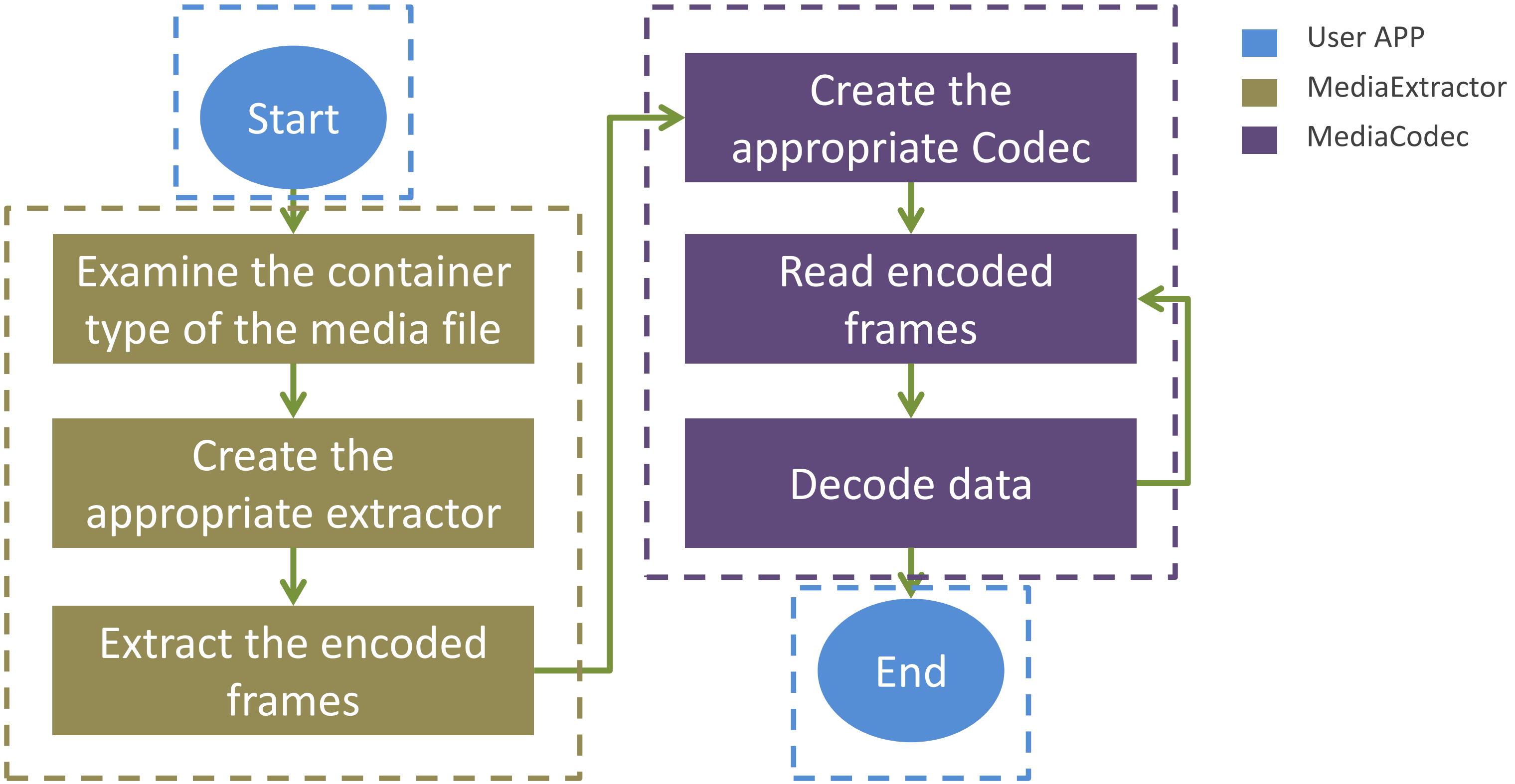
- Make extractor and codec work independently
- Bypass the media type sniffing mechanism
- Decode only one encoded frame

Make it more AFL-friendly

- Running as a single process

Let's start optimizing

Optimizations – Take a step back to Stagefright workflow


Start
Examine the container type of the media file
Create the appropriate extractor
Extract the encoded frames
Create the appropriate Codec
Read encoded frames
Decode data
End
User APP
MediaExtractor
MediaCodec

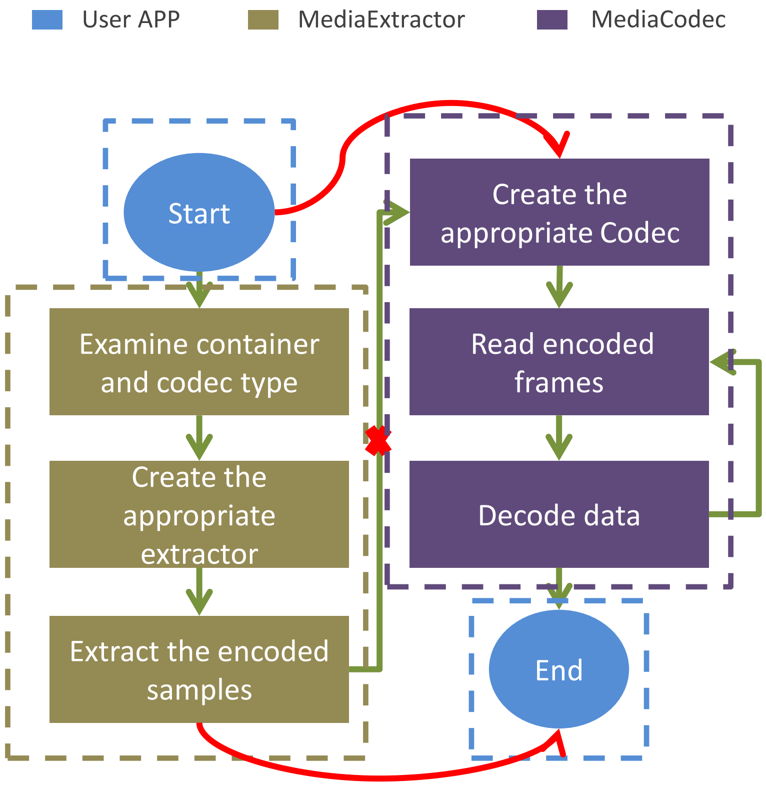
Optimizations – Make extractor and codec work independently

Why

- When fuzzing target is codec(extractor), extractor(codec) will waste CPU time
- When fuzzing target is codec, unnecessary check logic in extractor could stop the decoding process ahead of time

How

- Fuzz only one target at a time


User APP
MediaExtractor
MediaCodec
Start
Examine container and codec type
Create the appropriate extractor
Extract the encoded samples
Create the appropriate Codec
Read encoded frames
Decode data
End

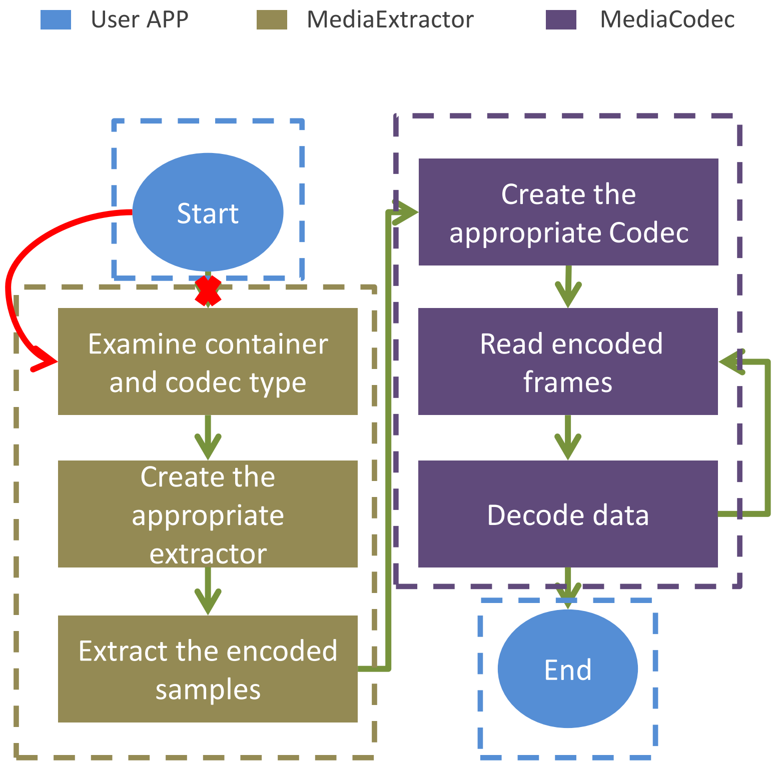
Optimizations – Bypass the sniffing mechanism

Why

- Fuzz one specify meida type at a time is more efficient, the sniffing is not necessary for this job
- Cause meaningless mutations: there is chance that one media type could be turned into other media type by AFL

How

- Specify the container and codec type


User APP
MediaExtractor
MediaCodec
Start
Examine container and codec type
Create the appropriate extractor
Extract the encoded samples
Create the appropriate Codec
Read encoded frames
Decode data
End

Optimizations – Decode only one frame

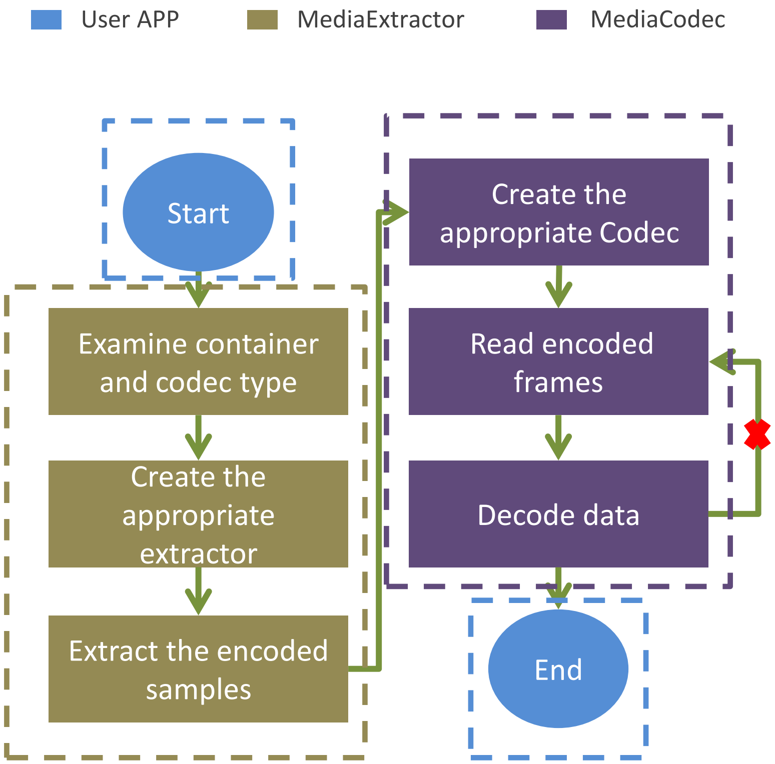
Why

- Typically, a standard media file contains multiple frames
- However, most vulnerabilities have been triggered when decoding the first frames – Just in my experience

How

- Break the decoding loop

```
if (numFrames == 1) break;
```


User APP
MediaExtractor
MediaCodec
Start
Examine container and codec type
Create the appropriate extractor
Extract the encoded samples
Create the appropriate Codec
Read encoded frames
Decode data
End

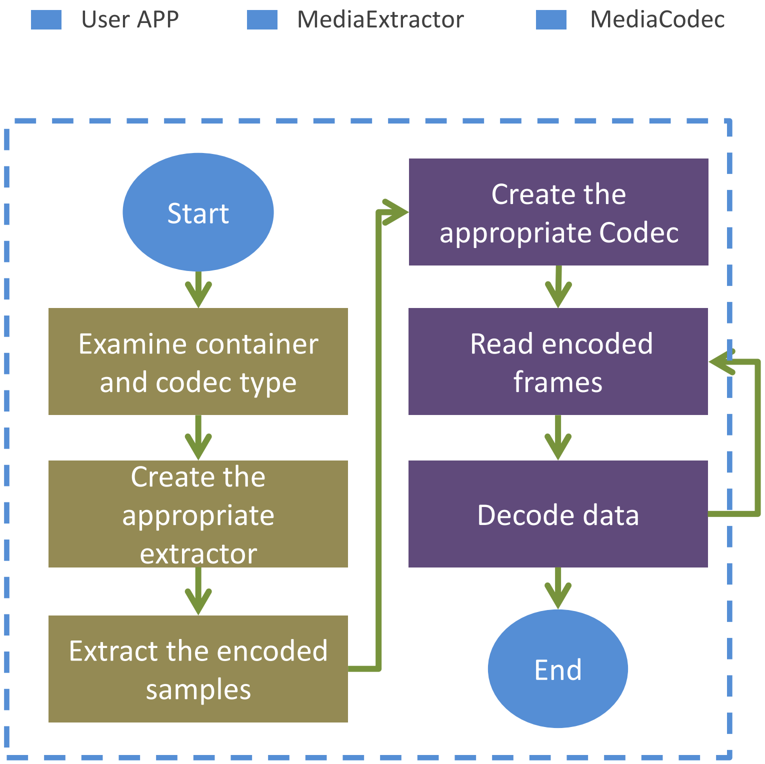
Optimizations – Running as a single process

Why

- Multi-process communication brings extra overhead
- AFL-unfriendliness

How

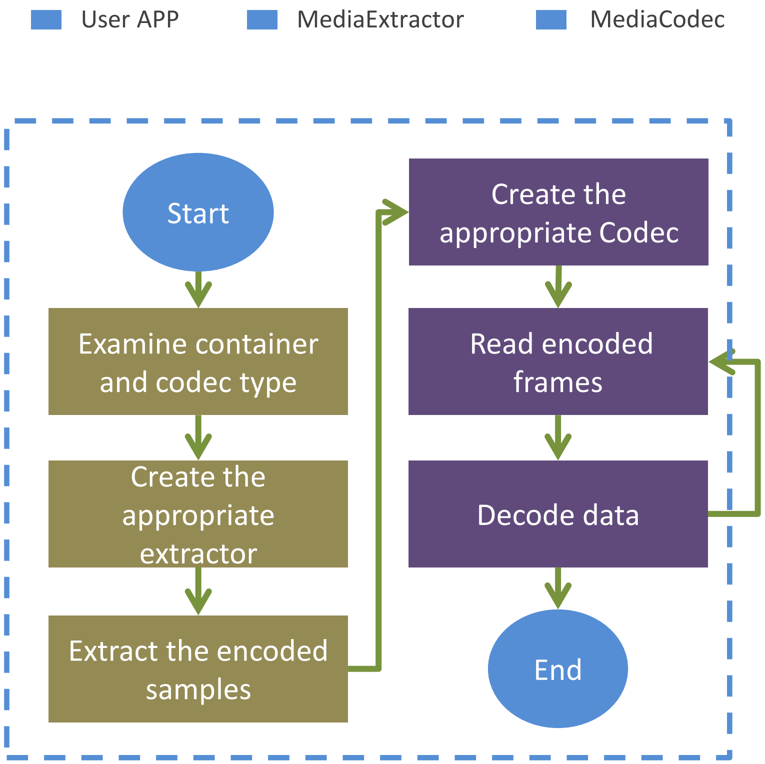
- Create services in local


User APP
MediaExtractor
MediaCodec
Start
Examine container and codec type
Create the appropriate extractor
Extract the encoded samples
Create the appropriate Codec
Read encoded frames
Decode data
End

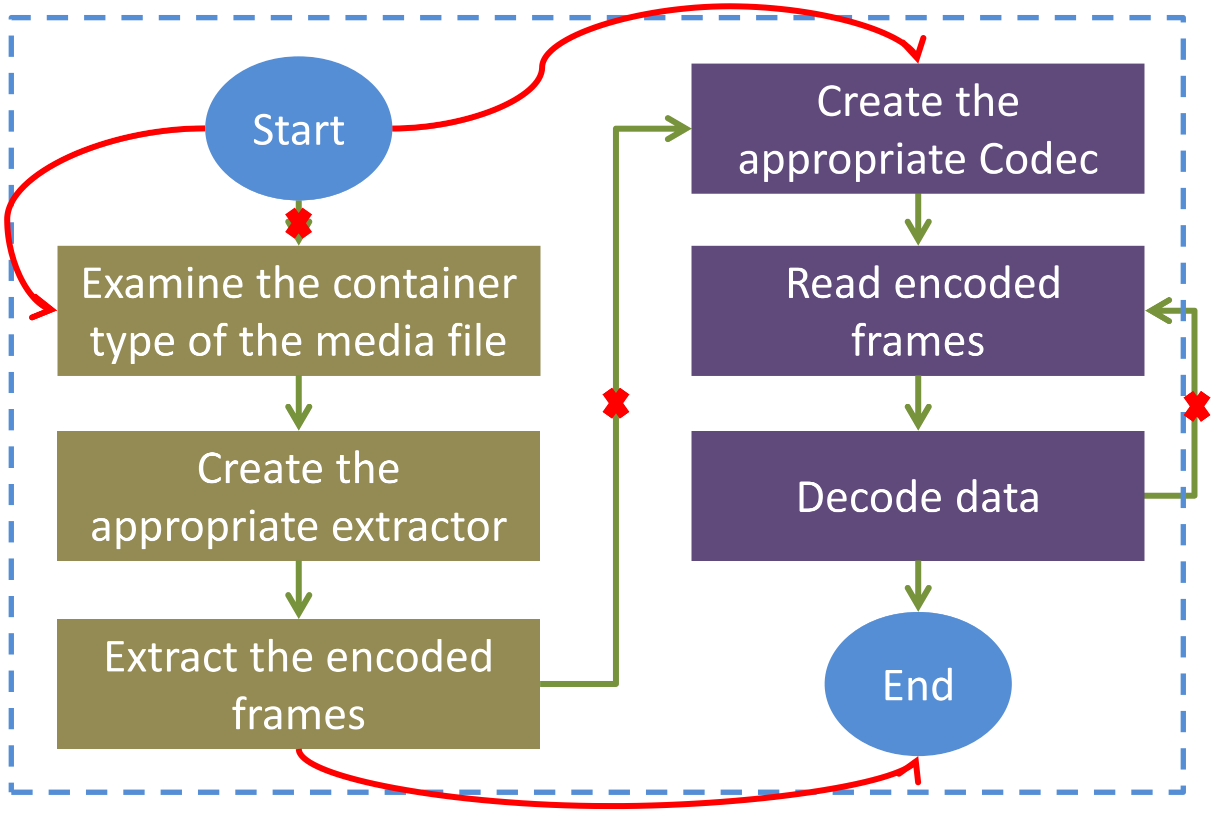
Optimizations – Running as a single process

```
// Create MediaCodecList object instance in local
sp<IMediaCodecList> MediaCodecList::getLocalInstance()
{
  sCodecList = new MediaCodecList;
  return sCodecList;
}

sp<IMediaCodecList> MediaCodecList::getInstance() {
  // Get the remote interface of the mediaserver via
binder IPC
  sp<IBinder> binder = defaultServiceManager()-
>getService(String16("media.player"));
  sp<IMediaPlayerService> service =
interface_cast<IMediaPlayerService>(binder);
  if (service.get() != NULL) {
    sRemoteList = service->getCodecList();
    if (sRemoteList == NULL) {
      // if failed to get remote list, create local
list
      sRemoteList = getLocalInstance();
    }
  }
  return sRemoteList;
}
```


User APP
MediaExtractor
MediaCodec
Start
Examine container and codec type
Create the appropriate extractor
Extract the encoded samples
Create the appropriate Codec
Read encoded frames
Decode data
End

Optimizations – The new Stagefright work flow

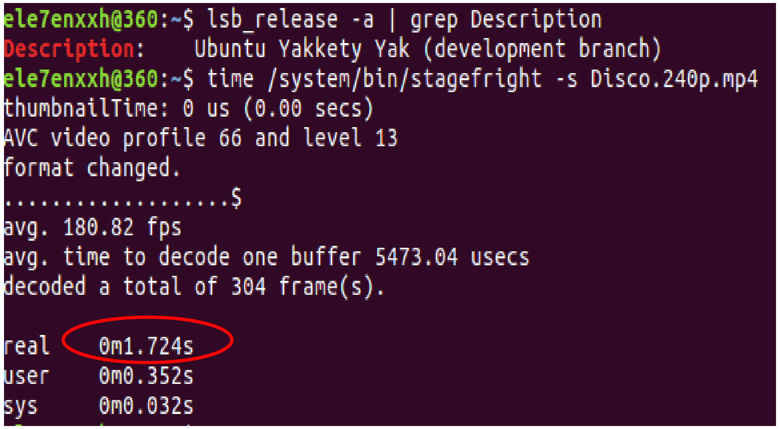

Start
Examine the container type of the media file
Create the appropriate extractor
Extract the encoded frames
Create the appropriate Codec
Read encoded frames
Decode data
End

- 🟦 User APP
- 🟦 MediaExtractor
- 🟦 MediaCodec

Optimizations – It works faster

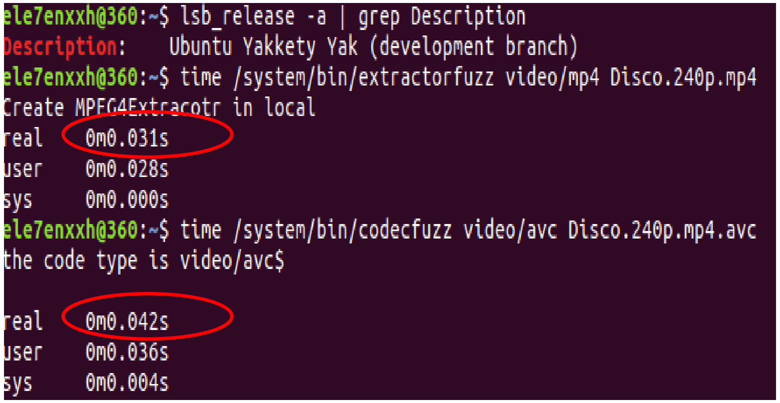
Two test harness for extractor and codec

20x+ performance gains



```
ele7enxxh@360:~$ lsb_release -a | grep Description
Description:    Ubuntu Yakkety Yak (development branch)
ele7enxxh@360:~$ time /system/bin/stagefright -s Disco.240p.mp4
thumbnailTime: 0 us (0.00 secs)
AVC video profile 66 and level 13
format changed.
...................$
avg. 180.82 fps
avg. time to decode one buffer 5473.04 usecs
decoded a total of 304 frame(s).

real    0m1.724s
user    0m0.352s
sys     0m0.032s
```



```
ele7enxxh@360:~$ lsb_release -a | grep Description
Description:    Ubuntu Yakkety Yak (development branch)
ele7enxxh@360:~$ time /system/bin/extractorfuzz video/mp4 Disco.240p.mp4
Create MPEG4Extracotr in local
real    0m0.031s
user    0m0.028s
sys     0m0.000s
ele7enxxh@360:~$ time /system/bin/codecfuzz video/avc Disco.240p.mp4.avc
the code type is video/avc$

real    0m0.042s
user    0m0.036s
sys     0m0.004s
```

Get more powerful test cases – Where to get test cases

The AOSP repository contains a large number of test cases

Search in Google

```
e.g. -inurl:htm -inurl:html intitle:"index of" .mp4
```

File format conversion tools

- ffmpeg

Open source project

- https://github.com/MozillaSecurity/fuzzdata
- https://github.com/stribika/afl-fuzz/tree/master/testcases

Get more powerful test cases – How to get encoded frames

Dump the data when decoding a media file

Refer to the following code

```
// framework/av/media/libstagefright/codecs/avcdec/SoftAVCDec.cpp
void SoftAVC::onQueueFilled(OMX_U32 portIndex) {
  …
  // If input dump is enabled, then write to file
  // pv_stream_buffer points to encoded frames
  // u4_num_Bytes is encoded frames length
  DUMP_TO_FILE(mInFile, s_dec_ip.pv_stream_buffer, s_dec_ip.u4_num_Bytes);
  …
}
```

Get more powerful test cases - Keep it small (< 1kb)

Keep only the video or audio data that actually want to fuzz

```
$ ffmpeg -i input_file -vcodec copy -an output_file_video
$ ffmpeg -i input_file -acodec copy -vn output_file_audio
```

Keep only one frame data

```
$ ffmpeg -i input_file -codec copy -frames 1 output_file
```

Use afl-tmin tool

Get more pwerful test cases – Keep it small (< 1kb)

Keep only the video or audio data that you actually want to fuzz

```
$ ffmpeg -i input_file -vcodec copy -an output_file_video
$ ffmpeg -i input_file -acodec copy -vn output_file_audio
```

Keep only one frame data

```
$ ffmpeg -i input_file -codec copy -frames 1 output_file
```

Use afl-tmin tool

Remove useless data, save CPU time

Fuzzing mp4 container – Test harness

```
// extractorfuzz.cpp
int main(int argc, char **argv) {
  if (argc != 3) return -1;
  ProcessState::self()->startThreadPool();
  // Running in persistent mode
  while (__AFL_LOOP(1000)) {
    // argv[2] is the input file path
    sp<FileSource> fileSource = new
FileSource(argv[2]);
    // argv[1]is the container type of the input file
    sp<IMediaExtractor> extractor =
MediaExtractor::Create(fileSource, argv[1]);
  }
  return 0;
}
```

Fuzzing mp4 container – Instrument

Add the following to Android.mk in test harness and libstagefright

```
LOCAL_CLANG := true
LOCAL_CFLAGS += -fno-omit-frame-pointer
-O2
export AFL_PATH :=
/usr/local/lib/afl/arm
export AFL_CC := /usr/local/bin/clang
LOCAL_CC := afl-clang-fast
export AFL_CXX :=
/usr/local/bin/clang++
LOCAL_CXX := afl-clang-fast++
```

Build

```
$ mm -j16
```

Fuzzing mp4 container – Get test cases

Get original test cases from here

- cts/tests/tests/media/res/raw/a_4_aac.mp4
- cts/tests/tests/media/res/raw/swirl_128x128_h264.mp4

Keep only one frame

```
ffmpeg -i a_4_aac.mp4 -codec copy -frames 1 a_4_aac_1frame.mp4
ffmpeg -i swirl_128x128_h264.mp4 -codec copy -frames 1 swirl_128x128_h264_1frame.mp4
```

Fuzzing mp4 container – Lunch afl-fuzz loop

Fuzzing in distributed mode

- Make full use of CPU performance

Increase the -m and -t limits

- The decoding process requires more memory and time

```
$ afl-fuzz -M fuzz0 -m 1024 -t 1000 -i in -o out --
extractorfuzz video/mp4 @@
$ afl-fuzz -S fuzz1 -m 1024 -t 1000 -i in -o out --
extractorfuzz video/mp4 @@
…
```

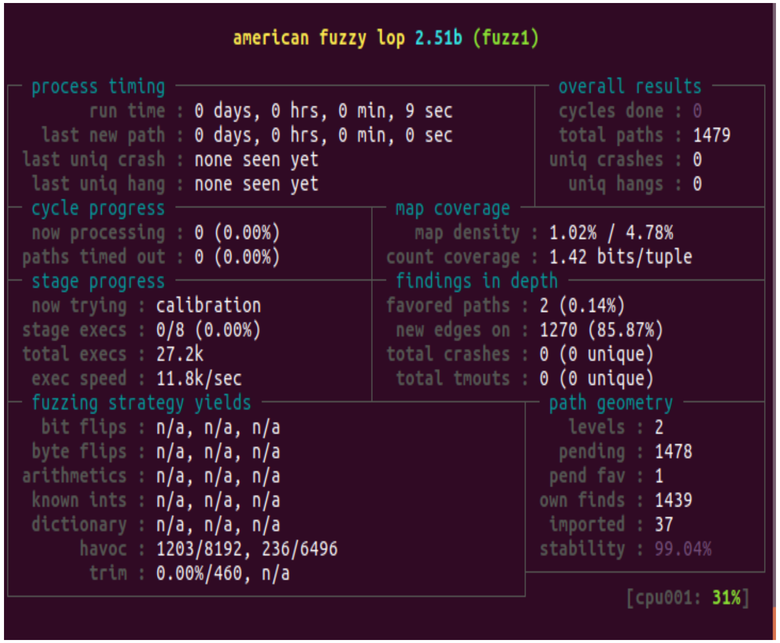
Fuzzing mp4 container – Running screen

Faster

- Exec speed > 10k/sec

Efficient

- Find 1400+ paths in 9 seconds



```
                    american fuzzy lop 2.51b (fuzz1)

┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 0 min, 9 sec        │  cycles done : 0     │
│   last new path : 0 days, 0 hrs, 0 min, 0 sec        │  total paths : 1479  │
│ last uniq crash : none seen yet                      │ uniq crashes : 0     │
│  last uniq hang : none seen yet                      │   uniq hangs : 0     │
├─ cycle progress ────────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 0 (0.00%)         │    map density : 1.02% / 4.78%       │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.42 bits/tuple     │
├─ stage progress ────────────────────┼─ findings in depth ───────────────────┤
│  now trying : calibration           │ favored paths : 2 (0.14%)             │
│ stage execs : 0/8 (0.00%)           │  new edges on : 1270 (85.87%)         │
│ total execs : 27.2k                 │ total crashes : 0 (0 unique)          │
│  exec speed : 11.8k/sec             │  total tmouts : 0 (0 unique)          │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : n/a, n/a, n/a                         │    levels : 2          │
│  byte flips : n/a, n/a, n/a                         │   pending : 1478       │
│ arithmetics : n/a, n/a, n/a                         │  pend fav : 1          │
│  known ints : n/a, n/a, n/a                         │ own finds : 1439       │
│  dictionary : n/a, n/a, n/a                         │  imported : 37         │
│       havoc : 1203/8192, 236/6496                   │ stability : 99.04%     │
│        trim : 0.00%/460, n/a                        ├────────────────────────┘
└─────────────────────────────────────────────────────┘          [cpu001: 31%]
```

Recognize unique crashes – Why

The AFL recorded crashes may be non-reproducible

- Different code for different processor architectures

Some crashes not interesting

- Assertion

The unique crashes recorded by AFL are not always unique

- AFL's uniqueness was determined based on tuple instrumentation that is too strict

Not crash ≠ Not vulnerability

- Some vulnerabilities(e.g. use-after-free) don't cause crashes

Recognize unique crashes – How

Push the unique crashes and the generated corpus to a real Android device with the latest security updates

Examine all corpus with ASAN-enabled again

- Build AOSP with ASAN:

```
$ make -j42
S SANITIZE_TARGET=address make -j42
$ fastboot flash userdata && fastboot flashall
```

Monitor crash logs

```
$ adb logcat
```

Record the unique crashes

- The uniqueness is determined based on ASAN backtrace information

Disclosure

Discovered vulnerabilities

Discovered vulnerabilities – Summary

As of October 1, 2017, total 30 vulnerabilities have been discovered

- 13 vulnerabilities are duplicate ☹
- 17 vulnerabilities(11 critical, 5 high and 1 moderate) have been disclosed on Android Security Bulletins ☺
- Some issues are still in process ☹

Covered multiple memory corruption vulnerability types

- Heap overflow
- Heap use after free
- Stack buffer overflow
- Global buffer overflow - Not fix yet
- Alloc dealloc mismatch
- FPE

Discovered vulnerabilities – Details

CVE	Type	Severity	components
CVE-2017-0678	heap-user-after-free	Critical	mp4/container
CVE-2017-0714	heap-buffer-overflow	Critical	h263/codec
CVE-2017-0719	heap-buffer-overflow	Critical	mpeg2/codec
CVE-2017-0718	heap-buffer-overflow	Critical	mpeg2/codec
CVE-2017-0722	heap-buffer-overflow	Critical	h263/codec
CVE-2017-0720	heap-buffer-overflow	Critical	hevc/codec
CVE-2017-0745	heap-buffer-overflow	Critical	mpeg4/codec
CVE-2017-0758	heap-buffer-overflow	Critical	hevc/codec
CVE-2017-0760	heap-buffer-overflow	Critical	mpeg2/codec

Discovered vulnerabilities – Details

CVE	Type	Severity	components
CVE-2017-0761	heap-buffer-overflow	Critical	avc/codec
CVE-2017-0764	stack-overflow	Critical	vorbis/codec
CVE-2017-0776	heap-buffer-overflow	High	avc/codec
CVE-2017-0777	heap-buffer-overflow	High	sonivox/container
CVE-2017-0778	heap-buffer-overflow	High	mp4/container
CVE-2017-0820	fpe	High	mp4/container
CVE-2017-0813	alloc-dealloc-mismatch	Moderate	mp4/container
CVE-2017-0814	heap-buffer-overflow	High	vorbis/codec

Discovered vulnerabilities – POCs

https://github.com/ele7enxxh/poc-exp

Conclusion

Presentation conclusion

Conclusion

A new efficient fuzzing strategy for Stagefright has been implemented, and 17 new vulnerabilities have been found

This fuzzing strategy is besed on Android Nougat and AFL-2.51b, it also compatible with other version in theory, but the details may need to be changed

Is there still have 0days in Stagefright ? - Yes, but need more powerful technologies or tricks

Not only fuzzing Stagefright on Linux, Fuzzing all Android native binaries on Linux too

Reference

https://source.android.com/security/bulletin/

http://events.linuxfoundation.org/sites/events/files/slides/ABS2015.pdf

https://www.blackhat.com/docs/us-15/materials/us-15-Drake-Stagefright-Scary-Code-In-The-Heart-Of-Android.pdf

https://hitcon.org/2016/CMT/slide/day2-r2-c-1.pdf

http://lcamtuf.coredump.cx/AFL/

https://github.com/huntcve/slides/blob/master/seven_shen_shakacon.pdf

https://www.blackhat.com/docs/eu-16/materials/eu-16-Jurczyk-Effective-File-Format-Fuzzing-Thoughts-Techniques-And-Results.pdf

Thank you very much

Zinuo Han

ele7enxxh@gmail.com