The Fuzzing Awakens: File Format-Aware Mutational Fuzzing on Smartphone Media Server Daemons

Minsik Shin, Jungbeen Yu, Youngjin Yoon, Taekyoung Kwon

The *Fuzzing* Awakens: File Format-Aware Mutational Fuzzing on Smartphone Media Server Daemons

MinSik Shin, JungBeen Yu, YoungJin Yoon, and Taekyoung Kwon*

Information Security Lab, Yonsei University, Seoul, 03722, Korea
{msshinaktl,symnoisy,youngjin339,taekyoung}@yonsei.ac.kr

Abstract. Media server daemons, running with a high privilege in the background, are attractive attack vectors that exist across various systems including smartphones. Fuzzing is a popularly used methodology to find software vulnerabilities although symbolic execution and advanced techniques are obviously promising. Unfortunately, fuzzing itself is not effective in such format-strict environments as media services. Thus, we study file format-aware fuzzing as a technical blend for finding new vulnerabilities. We present our black-box mutational fuzzing on the latest smartphone systems, Android and iOS, respectively, with manipulation of the MPEG-4 Part 14 file format and show results that affect a wide range of related systems. In our approach, we automate a seed file selection process to crawl a crowd-sourcing public website and validate arbitrary m4a/mp4 audio files according to the FOURCC atom list we gained through white-box analysis in Android. We acquired eight seed files covering all effective atoms in 2,600 seconds. We then performed size field mutation in a little amount and generated 1,102 test cases common to both systems. During six CPU hours of fuzzing, we identified three crash atoms in iOS 9.3.5 and 15 in Android 6.0.1, respectively. Due to format-awareness, we were able to easily locate crash points through a mutation table. It was discovered that the new crash atoms found in iOS allowed remote attackers to execute arbitrary code or cause a denial of service by memory corruption in iOS and also OS X, tvOS and watchOS.

Keywords: Mutational fuzzing · Format awareness · Media server

1 Introduction

Multimedia services are attractive attack vectors that exist across various systems and platforms. For instance, audio services, such as phone calls, ringtones, alarming sounds, audio file players, and audio streaming over mobile web browsers, are frequently requested in smartphones and related smart devices. To deal with these requests, a special daemon process having a high privilege runs in the background, e.g., `mediaserverd` in iOS and `mediaserver` in Android, and automatically restarts when it occasionally crashes. Unfortunately, its framework is quite complex, e.g., as a mixture of Java and native code in Android,

and a wide variety of audio codecs and plugins are written by non-security-experts. Furthermore, users are likely to install third-party sounds (e.g., the Star Wars imperial march ring-tone) and perceive an audio file or streaming relatively harmless, even played without user's consent (e.g., an auto-streaming on Facebook). Indeed, many vulnerabilities [11] have been discovered in Android 6.0.x regarding the Stagefright engine that manipulates audio-video playbacks, and significantly fixed in Android 7.1.x. On the contrary, a relatively small number of media file related vulnerabilities were found in iOS, and in particular audio-related vulnerabilities of iOS were rarely discovered (e.g., CVE-2010-0036 and CVE-2015-5862). An academic study was also less presented in the literature [19, 21]. Thus, it might be interesting to investigate audio-related vulnerabilities on both iOS and Android platforms because the same audio file formats are readily accepted in both smartphone platforms.

A fuzzing method, first introduced by Miller et al. in 1990 [25], is still the most widely-used tool[1] for finding software vulnerabilities in a variety of systems although symbolic execution and advanced techniques are obviously promising. Unfortunately, however, fuzzing itself is not effective in such format-strict environments as media services [33] because it needs a large amount of fuzzing work, mostly format-blocked, and it can hardly locate a crash point. To the best of our knowledge, there is no explicit literature studying the file format-aware fuzzing on the media server daemons of the smartphone systems and also for the latest versions, such as iOS 9.3.x and Android 6.0.x, when we conducted this study. Thus, it would also be interesting to investigate the file format-aware fuzzing on the latest smartphone systems.

Contributions. In this paper, concerning the problems and motivations above, we 'awaken' the file format-aware fuzzing for finding new security vulnerabilities related to media server daemons (with very little amount of fuzzing work and by easily locating the crash points under specific file formats) in the 'latest' versions of iOS and Android. For convenience, we call our methodology *TFA*, standing for 'Tiny Fuzzing on Audio' or 'The Fuzzing Awakens'. After reviewing backgrounds in Section 2, we describe TFA in Section 3 and evaluate it in Section 4. We discuss limitations and future work in Section 5 and related work in Section 6, and then conclude this paper in Section 7. We summarize our study as follows:

- TFA is *strategic*: We choose a seed file format as MPEG-4 Part 14 which is most widely played by default media players in smartphones, and confront several challenges to conduct file format-aware fuzzing. Our unique strategies to overcome such challenges are as follows: (1) To ease format-awareness, we utilize a GNU-GPL parser tool called AtomicParsley. This might be a benefit of targeting multimedia files. (2) To gain the valid MPEG-4 atom[2] list, we conduct a white-box analysis to the Android 6.0.1 source code. (3) To automate a seed file crawling process, we setup selection criteria of target websites and select a crowd-sourcing public website called `4shared.com`

[1] https://lcamtuf.blogspot.kr/2015/02/symbolic-execution-in-vuln-research.html
[2] An atom is a basic data unit in MPEG-4. Readers are referred to Figure 1 and [1].

for automation. (4) To automate a mutational fuzzing process in iOS, we backward-fuzz the old jailbroken version of iOS (7.1.2) and forward-verify new crashes in the latest version (9.3.5 on conducting our study) of which a jailbreaking is impossible for now.

- TFA is *efficient*: In a seed file selection phase, we obtain eight seed files only in 2,600 seconds to cover all of the effective and valid atoms. In a mutational fuzzing phase, we perform mutation of the atom size fields that are likely to cause a heap overflow. We generate 1,102 test cases[3] and a mutation table having 58 records only. They are commonly applied to Android and iOS.
- TFA is *effective*: Due to format-awareness, it is possible to simply locate crash atoms[4] through a mutation table.
  For six CPU hours, we found three crash atoms in iOS 9.3.5 and reported the results to Apple (CVE-2016-4702, CVSS Score 10.0). Interestingly, these new crash atoms commonly affected iOS and the related systems such as OS X, tvOS, and watchOS to allow remote attackers to execute arbitrary code or cause a denial of service by memory corruption. The new vulnerabilities were fixed in iOS 10.x.x.
  For six CPU hours, we found 15 crash atoms in Android 6.0.1 but they were thrown to an exception due to the existence of `CHECK()` functions that signal the kernel to kill and restart a media server daemon process.

2 Background

2.1 Attack Vectors: Media Server Daemons and Multimedia Files

As for file format-aware fuzzing, our target (media server daemons) and seed (multimedia file) both have a great implication as an attack vector because of the followings.

To deal with frequent audio service requests, media server daemons always run with a high privilege in the background, commonly in modern smartphones and the related systems. For instance, an media daemon called `mediaserverd` aggregates the sound output of all applications and governs events such as volume and ringer-switch changes in iOS and similarly in tvOS, watchOS, and OS X [19]. Android also runs a media server daemon called `mediaserver`, which is responsible for starting media related services, including Audio Flinger, Media Player Service, Camera Service, and Audio Policy Service [12], and the related systems, such as Android Auto, Android TV, Android Wear, and Android Things, also share this property in their source code. Thus, if a critical vulnerability is discovered in a smartphone regarding media server daemons, the related systems and devices might be affected by the same attack that exploits it.

Furthermore, it might be highly likely to feed a corrupted multimedia file to a vulnerable system as studied in the previous work [33], particularly without specialized access conditions or authentication to exploit the vulnerability.

[3] Much more test cases are required in dumb fuzzing.

[4] It is technically infeasible to identify unique crash atoms by performing a random bit-flipping only.

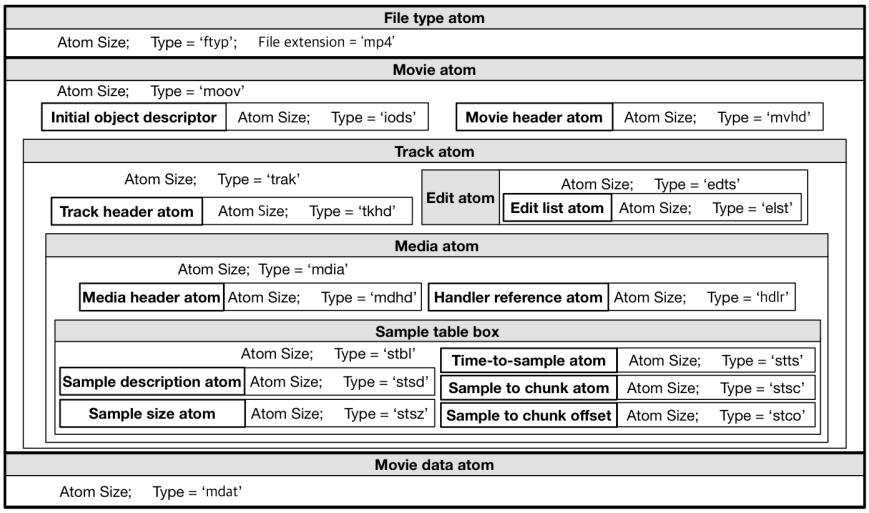

Fig. 1. MPEG-4 atom structure [1]

2.2 Seed File Format

According to `www.file-extensions.org`, 30 type extensions of audio and sound files, such as m4a, wma, and mp3, are most commonly used in various platforms among 1,285 file type extensions. Meanwhile, 35 type extensions of video and movie files, such as mp4, mov, and avi, are commonly used in various platforms among 659 file type extensions. All file extensions are categorized under digital container format such as Ogg, 3GP, QuickTime.

We chose a seed file format as MPEG-4 Part 14 which is the most widely used digital multimedia container format to store video and audio, such as mp4, m4a, m4r, m4b, m4p, and m4v. As depicted in Figure 1, m4a/mp4 files are structured hierarchically by the basic data unit called an *atom* or a *box*. An atom is called a container atom if it is not a leaf atom in the nested hierarchy. Each atom consists of a header, followed by atom data. The header contains the atom's size and type fields, giving the size of the atom in bytes and its type. Note that each atom has an 'offset' that describes the atom's position according to the size value. The size and the type fields are assigned 32-bit integers, respectively, and the size fields are the main target of our mutation in this paper because of high likeliness of raising memory collisions. Particularly, an mp4 file structure consists of three container atoms: File type atom (`ftyp`) has a certain file type of format, movie atom (`moov`) contains all meta data of corresponding media, and movie data atom (`mdat`) stores raw data. Moreover, movie atom (`moov`) involves various leaf atoms including track atom (`trak`) and media atom (`mdia`). There could be multiple track atoms (`trak`) in the same file. Note that it is very important to have a concrete strategy to select a qualified seed file because different atom structures can be made in files depending on a file generation environment, such as codec, encoder, program, and device.

Algorithm 1: TFA file format-aware fuzzing algorithm

Input : Website W

1: $T_x = \emptyset$, $T_y = \emptyset$ ▷ $T_y \leftarrow \{address, atom\_type, file\_names[n]\}$
2: **repeat**
3: S = SEED_CRAWLING(W) ▷ `Seed File Selection Phase`
4: **if** new atom found in FORMAT_AWARENESS(S) **then**
5: add S to T
6: **end if**
7: **until** abort-signal
8: **repeat**
9: t = CHOOSE_NEXT(T)
10: t' = ATOM_MUTATION(t, FORMAT_AWARENESS(t)) ▷ `Mut. Fuzzing Phase`
11: add t' to T_y
12: **if** t' crashes **then**
13: add t' to T_x
14: **end if**
15: **until** abort-signal

Output: Crashing Inputs T_x, Mutation Table T_y

3 File Format-Aware Mutational Fuzzing

3.1 Overview

We describe a general overview of TFA fuzzing procedure in Algorithm 1. The TFA fuzzing procedure consists of a seed file selection phase and a mutational fuzzing phase: We collect and choose seed files in the former, and perform mutation and fuzzing in the latter.

Let T_x and T_y denote a list of crashes and a mutation table, respectively. We assume they are initially null. In the seed file selection phase, we repeatedly perform SEED_CRAWLING through website W and verify atoms of the result S as in FORMAT_AWARENESS(S). If there exist new atoms in S, we add S to the queue T. We repeat this phase until we collect seed files that cover all atom lists. Otherwise, we abort this process. (line 2-7)

In the mutational fuzzing phase, we set t as the next seed file queued in T, and identify offset addresses of atom (size) fields through FORMAT_AWARENESS(). We then generate a test case t' through ATOM_MUTATION() and add this to a mutation table T_y which contains offset addresses, atom types and file names. We set $n = 19$ in our experiment. We finally feed this test case to the target system. If the generated test case t' crashes the media server daemon, it is added to the crashing input set T_x. We repeat this phase until we finish our fuzzing work with collected seed files, and otherwise abort this process. (line 8-15)

Note that it is simple to locate crash atoms through T_x and T_y due to file format-awareness. We automate the whole processes and implement the system for experiments with an environmental setting as described in Table 1.

Table 1. System environment

Device:	Web Server	MacBook (Xcode)	iPhone 4/6s	Nexus 5
OS ver:	Ubuntu 12.0.4.5	OS X 10.11.6	iOS 7.1.2/9.3.5	Android 6.0.1

3.2 Challenges

We dealt with the following challenges in a practical sense in our fuzzing work.

Format-awareness. It is well known that format-awareness is a crucial job in the related work [13, 29, 34]. To ease format-awareness, we utilize a GNU-GPL parser tool called AtomicParsley. We use parsed atom types for seed validation and offset address for mutation.

An m4a/mp4 file contains many atoms that include size, type, and even data fields while the number of structural atom types that could extend data fields is 302 in total according to the atom list of `www.mp4ra.org`. Thus, we conducted a white-box analysis to the Android 6.0.1 source code to gain the effective MPEG-4 atom list. As shown in Table 2, the FOURCC function, composed of switch statements as in (1), processes the input file by each atom as in (2). We then collect a FOURCC atom list of 94 atoms and validate arbitrary audio and video files according to this list.

Seed Crawling. To automate a seed file crawling process, we setup selection criteria of target websites as follows.

- Avoid a robot check function
- Utilize an internal search function for audio/video files
- Aim at a sufficiently large file pool

According the criteria, we selected a crowd-sourcing public website, `4shared.com` for automation.

Latest Version Problem. The final challenge was a *jailbreaking* or *rooting* process required for fuzzing on the smartphone platforms, that is, for automation inside the smartphones. It was infeasible for us to jailbreak the latest version of iOS (9.3.5) while it was possible to root the latest version of Android (6.0.1). Thus, as for iOS, we decided to awaken the old version of iOS (7.1.2) running on iPhone 4. As illustrated in Figure 2, we first performed fuzzing on iOS 7.1.2 by jailbreaking iPhone 4 and installing OpenSSH, BigBoss recommend tools, and SButils for running a fuzzer.

Table 2. Android FOURCC function examples

(1)

```
switch(chunk_type) {
 case FOURCC('m','o','o','v'):
 case FOURCC('t','r','a','k'):
 ...
 case FOURCC('m','d','i','a'):
 case FOURCC('m','d','a','t'):
 { ... } }
```

(2)

```
if (chunk_type==FOURCC('t','r','a','k'))
 { isTrack = true;
   Track *track = new Track;
   track->next = NULL;
   if (mLastTrack) {
     mLastTrack->next = track;
     ...} ... }
```

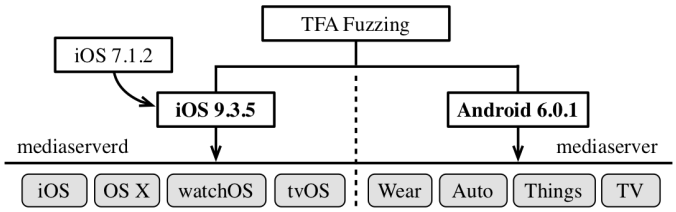

Fig. 2. Target system versions (Bold characters denote the latest versions.)

To verify the crashes (of iOS 7.1.2 iPhone 4) on iOS 9.3.5 in iPhone 6s, we connect iPhone 6s to the latest version of Xcode (8.2.1) running in a MacBook and check the device console and the device logs by manually loading the crash files to mobile Safari one by one. Note that the number of crash files, i.e., the test case files that aroused crashes in iOS 7.1.2 iPhone 4, is quite lightweight for us to manually load them to the latest version.

3.3 Main Phases

Seed File Selection Phase. As we mentioned, we chose `4shared.com` because it satisfies the requirements stated in our selection criteria. There was no robot check function in `4shared.com` but an internal search function was provided. The pool of `4shared.com` was sufficiently large with more than 1.5 million m4a/mp4 files. Note that we compare `4shared.com` with the case of `YouTube.com` in Section 4.1. We implemented automation software to crawl the website through mouse/keyboard macro features in C. We described this as SEED_CRAWLING in Algorithm 1. We implemented a validation tool in Python, which has a function that utilizes AtomicParsley to parse atom types and offset addresses.

Mutational Fuzzing Phase. Given the set of test case files from seed file selection phase, we implemented format-awareness for get the offset address of atom size field. Given the mutation targets from format-awareness, i.e., the positions of the size fields of the seed file, we need to generate small test cases for fuzzing. We implemented a simple generation tool in a Python code for automation of the test case generation. We see that this actually happened due to the mutation of atom size fields with the integers highly likely to arouse memory collisions [31].

We sequentially mutate the value of the size fields with four-byte mutation values as summarized in Table 3. We don't mutate size fields of duplicate atom types which mean the atoms already contained in another seed file. Those mutation values are highly probable integers for arousing memory collisions and

Table 3. Mutation values for atom size fields

1: 0x000000FF	2: 0x0000FF00	3: 0x00FF0000	4: 0xFF000000	5: 0xFFFFFFFF
6: 0x7FFFFFFF	7: 0x80000000	8: 0x20000000	9∼19: 0x00000000 ∼ 0x0000000A	

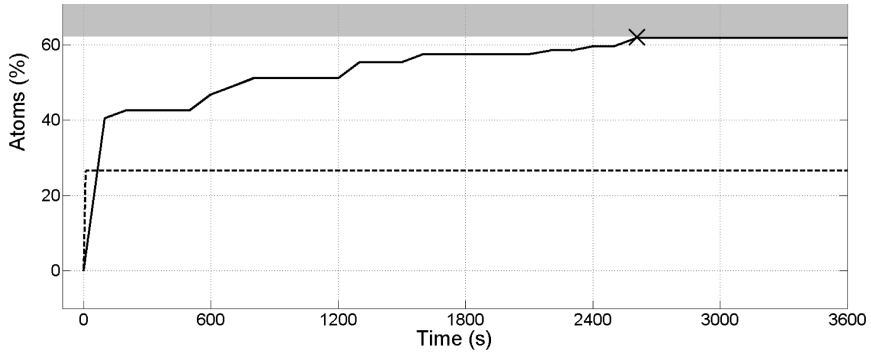

Fig. 3. #Atoms/FOURCC atom list over time for `4shared.com` (solid line) vs. `YouTube.com` (dashed line). Shade area indicates atom types unused in MPEG-4 Part 14 format. '✕' represents the point that achieves the maximum number of effective atoms, i.e., 58 atoms in 2,600 seconds.

actually borrowed from [31]. We construct a mutation table to contain a name of a test case file and atom types in each raw.

We need to instruct a media server daemon to consecutively load and handle each of the mutated test case files, and monitor and log possible crashes. We also automate this phase by writing a shell script to automatically open the test case files in the mobile web browsers, such as mobile Safari and mobile Chrome, one by one while monitoring the media server daemons for faults. Note that the simple shell script is very similar to that of [19]. We can easily locate the crash points effectively by file names of crash log due to the generated mutation table in test case generation phase.

4 Evaluation

4.1 General Results

Seed File Selection Phase. We performed a seed file crawling experiment in both `4shared.com` and `YouTube.com` using our automated crawling software. Figure 3 shows the results of crawling, i.e., the percentage of covered atoms in the FOURCC atom list (total 94 atoms). The gray area implies a portion (36 atoms including MPEG-B Part 7 atoms such as `sinf`, `tenc`, and `enca`) that is actually unused (i.e., ineffective) in MPEG-4 Part 14. As illustrated in a solid line in Figure 3, we were able to cover all effective atoms (58 atoms) in 2,600 seconds through `4shared.com`. We also made a comparison with `YouTube.com`. However, as shown in a dashed line in Figure 3, the number of atom types was constant (25 atoms) in `YouTube.com` because all files should have been uploaded in webm type and then converted to m4a/mp4 format in the server when downloaded. On the contrary, `4shared.com` downloaded crowd-sourcing files as in genuine

Table 4. TFA fuzzing results and comparisons. (Note: Numbers in parenthesis denote mp4 and the others m4a.)

Target	#Seed	TFA: #Atoms		Comparison: #Crash_Files	
		Mutated	Crash	ZZUF	AFL
iOS 9.3.5	4 (4)	44 (14)	3 (0)	0 (0)	-
Android 6.0.1	4 (4)	44 (14)	12 (3)	9 (1)	6 (4)

format. Consequently, we acquired eight seed files (four m4a and four mp4 files) covering all effective atoms in 2,600 seconds through `4shared.com`.

Mutational Fuzzing Phase. As summarized in Table 4, we found three crash atoms in iOS 9.3.5 and 15 crash atoms in Android 6.0.1 from 1,102 test cases for six CPU hours, respectively. Table 4 shows crash atom numbers according to OSs and seed types (mp4 in parenthesis). Due to the mutation table constructed on fuzzing, we were able to easily locate crash points and exploit input files.

As for crashes, we were able to discover vulnerabilities that cause memory corruption in iOS because we only mutated atom size fields related to memory assignment. The new crash atoms were `mvhd`, `trak`, and `udta` in iOS 9.3.5 and they were significant. However, in Android, various kinds of `CHECK()` and `CHECK_XX()` functions were used to verify size fields and then format-block (i.e., `SIGABRT`) a file that has unusual values in size fields. Thus, a mediaserver process which was regarded as a format error should have been killed and restarted by the Android kernel. Such crash atoms were `stbl`, `trak`, `mean`, `name`, `data`, `cprt`, `covr`, `albm`, `gnre`, `perf`, `titl`, `yrrc`, `auth`, `dinf`, and `mp4a` but they were only insignificantly denied in Android 6.0.1. Table 5 summarizes these results. Note that the crashes make the native daemon, i.e., the media server daemon and not only a forked server process, die and restart. As a result, the whole media server daemons and related services, such as audio, radio, and camera, were killed together. Even a simple crash could make a media server daemon and its related services go down temporarily because such a daemon process does not fork to handle individual requests.

Table 5. Experiment results for all FOURCC atoms. (Shade area indicates 58 fuzzed effective atoms. 17 crash atoms are in bold. Three iOS crash atoms are underlined.)

©alb	3g2b	**covr**	edts	h263	isom	meta	mp4v	s263	sinf	stz2	trex
©ART	3gp4	cpil	elst	hdlr	keys	mfra	MSNV	saio	**stbl**	tenc	trkn
©day	aART	**cprt**	enca	hev1	mdat	minf	**mvex**	saiz	stco	text	trun
©gen	**albm**	ctts	encv	hvc1	mdhd	moof	**mvhd**	samr	stsc	tfhd	tx3g
©nam	**auth**	d263	esds	hvcC	mdia	moov	**name**	sawb	stsd	**titl**	**udta**
©wrt	avc1	**data**	frma	ID32	mdta	mp41	**perf**	sbtl	stss	tkhd	**yrrc**
©xyz	avcC	**dinf**	ftyp	ilst	**mean**	mp42	pssh	schi	stsz	traf	
3g2a	co64	disk	**gnre**	iso2	mehd	**mp4a**	qt	sidx	stts	**trak**	

4.2 Comparisons

We adopt open source fuzzers such as ZZUF 0.15 [20] and AFL-fuzz 2.39b [37], for comparisons with TFA regarding their fuzzing performance (#Crashes/Time). The seed files selected in TFA were commonly used for comparison experiments. In ZZUF, we set the default mutation ratio as 0.4 for generating test cases. Note that even this small ratio makes a large amount of mutation. AFL-fuzz is a coverage-based grey-box fuzzer which uses coverage information to fuzz [37]. In this study, we used original AFL-fuzz with porting for comparison of fuzzing on Android. We plan to perform a comparison of fuzzing on iOS in the future study. In AFL-fuzz, we needed a specific procedure to make a precise comparison. First, we compiled the stagefright module that uses `libstagefright.so` library with `afl-clang-fast` and `afl-clang-fast++`, and then cross-compiled afl-fuzz for armv7-a to run on Nexus 5. Subsequently, we loaded afl-fuzz binary to Nexus 5 and conducted fuzzing on libraries. Finally, we conducted fuzzing again, but on a chrome browser with crash files obtained through libraries, to verify the crash results under our experiment design.

For comparisons with TFA, we treated the same eight seed files (four m4a and four mp4 files) selected from TFA, and set the same amount of time (six CPU hours) for fuzzing in ZZUF and AFL-fuzz, as TFA. The comparison results are summarized in Table 4. We were able to exactly identify crash atoms due to format-awareness in TFA but only crash files in ZZUF and AFL-fuzz. Furthermore, we were able to find more crashes, three of which found on iOS 9.3.5 were significant. We reported the results to Apple to be archived in CVE-2016-4702 (CVSS Score 10.0).

5 Limitations and Future Work

In this Section, we review some limitations of this work and discuss promising (and our own on-going) future work.

Seed Container Type Selection. Although we selected two audio/video file types such as m4a and mp4 from the MPEG-4 Part 14 file type lines, it would be promising to expand the sort of container types for more interesting results in the future study. Note that, for example, there exist 3 audio file container, 2 image file container, and 13 audio/video container used for multimedia file types. If there does not exist a file format specification or a parser tool, it would be promising to adopt the automatic input format recovery [6, 9, 23] to automatically analyze the native file formats for enabling the automated file format awareness phase in general.

Coverage-based Fuzzing. It would be promising to combine file format-aware fuzzing like TFA and coverage-based fuzzing in the future study. We did not consider deeper paths and code coverage in TFA but plan to work on coverage-based fuzzers, such as libFuzzer and AFL-fuzz, by replacing the mutation strategy part with TFA-like format-awareness strategy. It might be expected to improve the performance of fuzzers significantly.

Latest Version. When we work on TFA, the latest version of iOS was 9.3.2 - 9.3.5 and we reported our result to Apple. Our result was announced as CVE-2016-4702. We confirmed that all the crashes found in TFA work was successfully blocked in the current latest version of iOS 10.2.1. In the future study, we will work on iOS 10.2.1 with the plans described above.

6 Related Work

Fuzzing. Fuzzing first appeared as technical jargon in 1990 when Miller et al. showed that randomly generated character streams, fed into UNIX utility programs, could result in significant program crashes [25]. Fuzzing is such a security test method that causes crashes and discovers potential memory corruption and vulnerabilities by feeding randomized data into target application programs [24, 32]. Fuzzing is classified into Black-box fuzzing [10, 19, 21, 28, 36], white-box fuzzing [5, 7, 14, 16, 17, 35], and grey-box fuzzing [2–4] according to the existence of internal information of software. Our TFA fuzzing starts with a white-box analysis to gain the FOURCC atom list but mainly runs with black-box fuzzing on both iOS and Android. Fuzzing can also be classified into generation-based fuzzing [15, 18, 26] and mutational-based fuzzing [21, 28, 36] according to the way of input data generation. Our TFA fuzzing is mutational-based fuzzing Fuzzing was actively used for protocol security test around 1999, e.g., the PROTOS test suite of the University of Oulu [29]. File fuzzing received public attention in 2004 due to Microsoft security bulletin MS04-028.

Recently, fuzzing received academic attention regarding seed selections and mutation strategies. In 2014, A. Rebert et al. [28] pointed out that there had been little systematic effort in understanding the science of how to fuzz properly and studied how best to pick seed files to maximize the total number of bugs found during a fuzz campaign. In 2015, Cha et al. [8] studied how to compute a probabilistically optimal mutation ratio when a certain program-seed pair is given. They leveraged white-box symbolic analysis on an execution trace for a given program-seed pair to detect dependencies among the bit positions of an input. In 2016, Spephens et al. [30] proposed the so-called Driller which is a hybrid vulnerability excavation tool to exercise deeper paths in executables. They used selective concolic execution to generate inputs which satisfy the complex checks separating the compartments exercised by inexpensive fuzzing. In 2016, Bohme et al. [4] proposed the CGF (Coverage-based Greybox Fuzzing) method to explore significantly more paths with the same number of tests. They used the Markov chain model for the purpose.

File Format-awareness. Sutton et al. [31] introduced the benefits of investigating seed file formats. Due to file format-awareness, it would be possible to appropriately mutate more interesting fields in seed files. In 2007, Lewis et al. [22] used a modification of traditional format-free fuzzing techniques to identify vulnerabilities in the format-strict environment of media players. They stressed the significance of having the appearance of a valid media file on fuzzing. In 2008,

Thiel presented the results of file format-aware fuzzing on 'PC' media software by investigating various media file formats [33]. It was possible to find new bugs that could not be found in simple fuzzers.

Fuzzing on Smartphone. There have been studies of fuzzing on iOS and Android related to our research but there is no file format aware fuzzing in this area [19, 21, 27]. In 2010, Klein et al. introduced a black-box 'byte-wise' mutational fuzzing on the previous versions of iOS 3.1.3 by selecting an m4r/m4a ring-tone file (415,959 bytes) as a seed file and discovered a unique bug regarding `mediaserverd` (CVE-2010-0036). He generated test cases by sequentially mutating offset 0 to 999 with value of 255 [19]. In 2015, Lee et al. [21] studied a qualified seed file selection strategy and adopted a mutational fuzzing to find bugs from old iOS versions (6.x-6.1.x) and Android old versions (2.3.x-4.3.x), respectively [21]. They discovered several crashes by fuzzing real world format (*gif*, *jpeg*, *png*, *mp3*, and *mp4*) in both iOS and Android. They used CACE (Crash Automatic Classification Engine) method to design SFAT (Seed File Analysis Tool) for a good selection of seed files and categorized the discovered crashes automatically. As a result, they achieved better results by discovering about 1900 crashes and seven (unlike) unique bugs in iOS and Android. Compared with Lee et al.'s work, our research is different in analyzing the file format to optimize the number of test cases and fuzzing time.

7 Conclusion

In this paper, we studied file format-aware fuzzing on the media server daemons running in the latest versions of smartphones, and presented a new smartphone attack that exploits the results of such a fuzz input. Our methodology was strategic to overcome fuzzing challenges, such as format-awareness, seed file crawling, format-aware fuzzing, and latest versions. It was efficient for black-box mutational fuzzing with regard to a seed file selection and a test case generation, and also effective to find critical crash atoms in iOS and related systems. Although both iOS and Android have been updated on submission of this paper, we believe our strategic, efficient, and effective methodology and the proven results (CVE-2016-4702 CVSS Score 10.0) are promising for the future study. We plan to diversify input file formats in the near future.

Acknowledgement

This work was supported by the National Research Foundation of Korea (NRF-2015-R1A2A2A01004792).

References

1. Quicktime file format specification. `https://developer.apple.com/library/mac/documentation/QuickTime/QTFF/QTFFPreface/qtffPreface.html`

2. Bekrar, S., Bekrar, C., Groz, R., Mounier, L.: Finding Software Vulnerabilities by Smart Fuzzing. In: Proc. the IEEE International Conference on Software Testing, Verification and Validation (ICST). pp. 427–430 (2011)
3. Bekrar, S., Bekrar, C., Groz, R., Mounier, L.: A Taint Based Approach for Smart Fuzzing. In: Proc. the IEEE International Conference on Software Testing, Verification and Validation (ICST). pp. 818–825 (2012)
4. Böhme, M., Pham, V.T., Roychoudhury, A.: Coverage-based Greybox Fuzzing as Markov Chain. In: Proc. the ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 1032–1043 (2016)
5. Bounimova, E., Godefroid, P., Molnar, D.: Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. In: Proc. the International Conference on Software Engineering (ICSE). pp. 122–131 (2013)
6. Caballero, J., Yin, H., Liang, Z., Song, D.: Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis. In: Proc. the ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 317–329 (2007)
7. Cadar, C., Ganesh, V., Pawlowski, P., Dill, D., Engler, D.: Exe: A System for Automatically Generating inputs of Death Using Symbolic Execution. In: Proc. the ACM Conference on Computer and Communications Security (CCS) (2006)
8. Cha, S.K., Woo, M., Brumley, D.: Program-adaptive Mutational Fuzzing. In: Proc. the IEEE Symposium on Security and Privacy (S&P). pp. 725–741 (2015)
9. Cui, W., Peinado, M., Chen, K., Wang, H.J., Irun-Briz, L.: Tupni: Automatic Reverse Engineering of Input Formats. In: Proc. the ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 391–402 (2008)
10. De Ruiter, J., Poll, E.: Protocol State Fuzzing of TLS Implementations. In: Proc. the USENIX Security Symposium (2015)
11. Drake, J.: Stagefright: Scary Code in the Heart of Android. BlackHat USA (2015)
12. Drake, J.J., Lanier, Z., Mulliner, C., Fora, P.O., Ridley, S.A., Wicherski, G.: Android Hacker's Handbook. John Wiley & Sons (2014)
13. Eddington, M.: Peach Fuzzing Platform. `http://www.peachfuzzer.com/`
14. Ganesh, V., Leek, T., Rinard, M.: Taint-based Directed Whitebox Fuzzing. In: Proc. the International Conference on Software Engineering (ICSE). pp. 474–484 (2009)
15. Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based Whitebox Fuzzing. In: Proc. the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). vol. 43, pp. 206–215 (2008)
16. Godefroid, P., Levin, M.Y., Molnar, D.: SAGE: Whitebox Fuzzing for Security Testing. Queue 10(1), 20 (2012)
17. Godefroid, P., Levin, M.Y., Molnar, D.A., et al.: Automated Whitebox Fuzz Testing. In: Proc. the Network and Distributed System Security Symposium (NDSS). vol. 8, pp. 151–166 (2008)
18. Holler, C., Herzig, K., Zeller, A.: Fuzzing with Code Fragments. In: Proc. the USENIX Security Symposium. pp. 445–458 (2012)
19. Klein, T.: A Bug Hunter's Diary: A Guided Tour Through the Wilds of Software Security. No Starch Press (2011)
20. Labs, C.: ZZUF. `http://caca.zoy.org/wiki/zzuf`
21. Lee, W.H., Srirangam Ramanujam, M., Krishnan, S.: On Designing an Efficient Distributed Black-box Fuzzing System for Mobile Devices. In: Proc. the ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 31–42 (2015)

22. Lewis, C., Rhoden, B., Sturton, C.: Using Structured Random Data to Precisely Fuzz Media Players. Project Report. University of UC Berkeley (2007)
23. Lin, Z., Zhang, X.: Deriving Input Syntactic Structure from Execution. In: Proc. the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE). pp. 83–93 (2008)
24. Miller, B.P.: Fuzz Testing of Application Reliability. UW-Madison Computer Sciences (2007)
25. Miller, B.P., Fredriksen, L., So, B.: An Empirical Study of the Reliability of UNIX Utilities. Communications of the ACM 33(12), 32–44 (1990)
26. Miller, C., Peterson, Z.N.: Analysis of Mutation and Generation-based Fuzzing. Independent Security Evaluators (2007)
27. Mulliner, C., Miller, C.: Fuzzing the Phone in Your Phone. In: BlackHat USA. vol. 25 (2009)
28. Rebert, A., Cha, S.K., Avgerinos, T., Foote, J., Warren, D., Grieco, G., Brumley, D.: Optimizing Seed Selection for Fuzzing. In: Proc. the USENIX Security Symposium. pp. 861–875 (2014)
29. Röning, J., Lasko, M., Takanen, A., Kaksonen, R.: Protos-systematic Approach to Eliminate Software Vulnerabilities. Invited presentation at Microsoft Research (2002)
30. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In: Proc. the Network and Distributed System Security Symposium (NDSS) (2016)
31. Sutton, M., Greene, A.: The Art of File Format Fuzzing. In: BlackHat USA (2005)
32. Takanen, A., Demott, J.D., Miller, C.: Fuzzing for Software Security Testing and Quality Assurance. Artech House (2008)
33. Thiel, D.: Exposing Vulnerabilities in Media Software. In: BlackHat EU (2008)
34. Tool: Spike Fuzzer Platform. `http://www.immunitysec.com`
35. Wang, T., Wei, T., Gu, G., Zou, W.: Taintscope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In: Proc. the IEEE Symposium on Security and Privacy (S&P). pp. 497–512 (2010)
36. Woo, M., Cha, S.K., Gottlieb, S., Brumley, D.: Scheduling Black-box Mutational Fuzzing. In: Proc. the ACM SIGSAC Conference on Computer and Communications Security (CCS). vol. 9 (2013)
37. Zalewski, M.: American Fuzzy Lop (AFL) fuzzer. `http://lcamtuf.coredump.cx/afl/`