

Using Structured Random Data to Precisely Fuzz Media Players

Colleen Lewis, Barret Rhoden, Cynthia Sturton

December 17, 2007

Abstract

Users rarely consider their media player as a security critical application. However, with an increasing amount of media content available on the web, users are exposing themselves to attack by downloading possibly malicious content. We focus on identifying vulnerabilities in three media formats (AVI, MPEG and Ogg) and two media players (MPlayer and VLC). We use a modification of traditional format-free fuzzing techniques to identify vulnerabilities in the format-strict environment of media players. We build upon typical fuzzing techniques by (1) adding structure to random files and (2) randomizing real files. We find that with these added techniques, fuzzing can be used to find bugs in applications with strict format requirements. Randomizing real files can, with no knowledge of file structure, identify a wide variety of bugs. While strategically adding structure to random files can produce a greater number of crashes, this was not correlated with finding a greater number of unique bugs.

1 Introduction

Everyday people download multimedia content from untrusted sources on the Internet, such as YouTube or BitTorrent. When users play this content on their computer, any bugs in their media players are exposed to attack. Fuzzing, the method of using random data as the input to an application, is a form of black box testing that has recently been used to find bugs and security vulnerabilities in many programs, including web browsers and operating system utilities [3,4,5,7,8,9].

We have developed a framework for fuzzing media players to explore the potential of applying fuzzing techniques to decrease the risk from downloaded multimedia. For the purpose of our study, we are focusing on 3 media file formats: (1) AVI, (2) MPEG and (3) Ogg with two media players: (A) MPlayer and (B) VLC. In our fuzzing framework, we use a variation of fuzzing by including some structure with the otherwise random program input. Without some structure, we were unable to find any bugs in our target file formats.

Through this format-aware fuzzing, we were able to crash MPlayer and VLC with the Ogg and AVI file formats and identify specific bugs in the applications.

In this paper, we provide an assessment of our two format-aware fuzzing methods in terms of total number of crashes, diversity of bugs found, and efficiency in finding vulnerabilities. In addition, we provide an analysis of our results with respect to each file format and each media player. To conclude, we provide a classification of all vulnerabilities identified.

1.1 Threat Model

We address the threat model in which an adversary finds an exploit in a media player and then puts explicitly malicious content on the Internet, disguised as a media file. Anyone who downloads and attempts to play this malicious content could have their computer compromised. An adversary could easily post their malicious content on a file sharing site to attack a wide range of individuals. We focused specifically on open-source media players, since the source is available for inspection. However, these methods would apply to both open and closed-source software.

1.2 The Difficulties

The file format requirements of media players already provide some protection from fuzzing. When we fuzzed media players with completely random data, in most cases the media player refused to play the file as it did not meet the minimum formatting requirements. To reach media player vulnerabilities, a file containing malicious content would need to have at least the appearance of a valid media file. Therefore, in order to search for those vulnerabilities, our fuzzing needs to make it past the format validation of media players by providing files with the necessarily level of structure. We satisfy this requirement with two different methods. Our first method is to add structure to random files so they appear to the media player as valid. Our second method is to corrupt real media files by changing random bits or random bytes.

2 Architecture

Our tool set for fuzzing media players comprises four distinct components:

1. Creating random files
2. Adding structure to random files (one for each format)
3. Randomizing valid media files

4. Testing media players with our assortment of random files

In addition to these main pieces, we also have tools to gather our results and search through them to enumerate crashes. Each of these distinct components are separate programs that are run by a small program that can be configured to start multiple runs of any combination of the components. The advantage of this modular approach is that each component can be written independently of the others and if needs change, any component can easily be swapped out and replaced with another. Figure 1 illustrates the components of our framework.

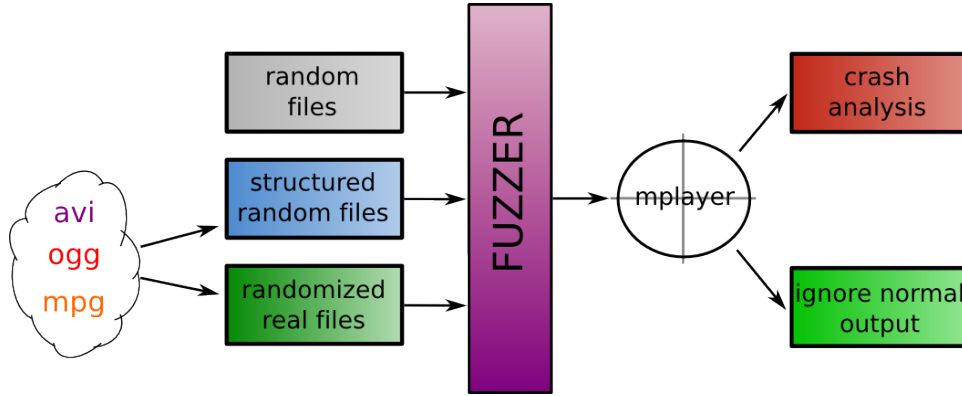


Figure 1: *Simplified architecture diagram. There are 3 components to prepare files (shown in the gray, blue and green boxes on the left): (1) creating random files, (2) adding file-format specific structure to random data and (3) corrupting valid media files. Each of these can be used to fuzz a media player (MPlayer or VLC) and the results are collected and organized. The framework has a single configurable module that can run multiple instances of the components.*

2.1 Naming Convention

Given the complexity of the architecture described above, we found it convenient to talk about the architecture with the metaphor of a ninja’s workshop. We outline our terminology here briefly because it was so useful to us when discussing our project that we will continue to use the terminology in this paper. The ninja attacks the target (the media players) with varying levels of files, all created by the craftsmen (miner, blacksmith, and recycler). The metaphor may seem silly, yet it helped us maintain clarity in our intra-team communication and organization of our code. To analyze the success of the ninja, the undertaker and coroner organize and categorize the results. A configurable program that can run instances of the components in multiple threads exists and is called the Machine.

2.2 Creating Random Files

2.2.1 Miner

The miner creates files of random data of varying sizes for the blacksmith. To compare to typical fuzzing methods of using purely random data, the ninja fuzzed the media players with the raw output of the miner.

2.3 Adding Structure to Random Files

The programs that add structure to a random file are the blacksmiths. We have three blacksmiths, one for each media file type that we fuzzed (AVI, MPEG, Ogg) and each structure-adding program is capable of adding two or more levels of structure.

2.3.1 AVI Blacksmith

The AVI specification is a container format that contains multiple streams of audio and video data. The structure has a main header that identifies the type and size of the file. Below that main header are two chunks, one that provides information about the stream formats and one that contains the stream data.

The AVI blacksmith has two possible levels of structure. The first level provides the minimum level of structure: the main header. The second level, in addition to including the main header, provides the header information for the two large chunks and nested header information for each data stream in the file. In the case of the nested headers, header fields (such as size of stream chunk) are filled in with valid data, while actual stream information (such as sampling rate) is left as random data.

2.3.2 MPEG Blacksmith

We focused on simulating the structure of MPEG-1, which is the first in the series of MPEG video and audio compression standards. This format has a hierarchical structure of headers. Each MPEG file has one MPEG header followed by a series of picture-group headers. Between each of the picture-group headers there are some number of picture headers, which represent the key frames of the video file.

The MPEG blacksmith consists of two possible levels of structure. The first level of structure provides the correct content for the header of the entire file. The second level of structure provides picture-group headers as well as picture headers.

2.3.3 Ogg Blacksmith

Ogg specifies a format for a container that can hold several streams of data. The data can come from any codec. The more common Ogg codecs, which the blacksmith can mimic, are Vorbis for audio and Theora for video. The codecs partition their data streams into packets, which are encapsulated in the Ogg container. A typical Ogg container consists of several pages, each beginning with “OggS”. A codec stream maps to several pages, such that each page also has information identifying its location in its stream. Each page has up to 255 segments, each up to 255 bytes long. Each codec packet is stored in one or more of these segments. The Vorbis and Theora codecs have similar structure. Both must start with three header packets that identify and initialize the stream. One of these header packets can contain the comment meta-data [14].

For our experiments, we used five levels of structure. The first level applied the page heading and no checksum. Level 2 ensured all pages belonged to a stream that was properly synchronized, but the streams had no particular codec. Level 3 used Vorbis and Theora stream header packets with no comment fields. Level 4 used real header packets, from a properly formatted file, with fuzzed comment fields. Level 5 used real header packets with no comment fields.

2.4 Randomizing Valid Media Files

2.4.1 Recycler

The recycler corrupts valid media files by replacing some percentage of the data in the file. It can be set to overwrite at the bit level or at the byte level. In bit mode each bit has a certain chance of being flipped. In byte mode each byte has a certain chance of being replaced with a random byte. In either case, the rate of corruption is a parameter set by the user.

2.5 Attacking Video Players

2.5.1 Ninja

The ninja sends the files created by the miner, blacksmith, and recycler to the media player as input and saves the output for later analysis. We categorize a failure as any time the media player crashes or exits abnormally. Anything else is a success. In many cases, the media player will exit without attempting to play the file, which is the proper behaviour. Another possible bug is for the program to hang. For the purposes of our experiments, if

the media player has not exited after a set amount of time, then the program is categorized as a timeout, which is not a failure.

One of the greatest advantages of our architecture is providing reproducible results, due to the fact that the input files are saved. All files that were sent to MPlayer were also sent to VLC to provide the opportunity to compare how the different applications responded to the same input. Furthermore, all files are available for future replays and analysis.

2.6 Assessing the Damage

2.6.1 Undertaker

We have scripts for culling the logs to find media files that caused a target application to crash and for categorizing those files according to which application, file type, and code library caused the crash. The undertaker searches through the ninja's output and reports on how many crashes occurred per target, format, and structure level. The undertaker also outputs a list of all of the files that caused a crash, which can then be examined by the coroner.

2.6.2 Coroner

The coroner is responsible for generating a back trace of every crash. For every file that caused a crash, the coroner runs the target in gdb with that file as input. When the target crashes, the coroner logs the back trace from gdb. Later, the coroner parses the gdb logs and reports the number of unique crashes and at which function the crash occurred. At this point, we manually inspect each unique crash to determine the specific cause.

3 Methodology

3.1 Players and Platforms

Our testing platform was a Gentoo Linux system, running kernel 2.6.23-gentoo-r1, on a T61 Thinkpad laptop. Our two target applications were MPlayer and VLC. All of the software was the latest stable packages for Gentoo as of November 2007. These were not the latest, bleeding-edge packages available, but were the versions a typical user would run. The only exception was MPlayer, which we were forced to upgrade to be able to get gdb backtraces due to an odd bug with rc1.

Software Package	Version
mplayer	1.0_rc1_p20070824 and later 1.0_rc2
vlc	0.8.6c
ffmpeg	0.4.9_p20070616
libogg	1.1.3
libtheora	1.0_alpha6-r1

3.2 Files Created

Our first component, the miner, created 250 random files for each of the file sizes listed below, for a total of 1500 files. Each of these files were sent to the media players to determine their behavior.

File sizes: 100KB, 512KB, 1MB, 2MB, 5MB, 10MB

Our second set of components, the three blacksmiths, used the above random files (250 of each size) and added structure for the AVI, MPEG, and Ogg formats. Each of the blacksmiths, as described above, has multiple levels of structure that it can provide. Each of the total of 1500 random files had each level of structure added and tested. Ogg, with five levels was tested with 7500 files, while AVI and MPEG with two levels were tested with 3000 files.

The third component, the recycler, used three real files as its starting point. Each file had identical content but was encoded with the target formats of AVI, MPEG, and Ogg. While keeping the real file fixed, we modified the percentage of corruption for both bit-wise and byte-wise modification. The percentages of modification for bit-wise and byte-wise modification are provided below. We created 100 files for each percentage of modification for a total of 1200 bit-wise and 1200 byte-wise modified files sent to the media players in each format.

Bit-wise corruption levels: 0.002, 0.02, 0.2, 0.3, 0.4, 0.5, 1, 5, 10, 20, 40, 60

Byte-wise corruption levels: 0.002, 0.02, 0.2, 1, 3, 4, 5, 6, 10, 20, 40, 60

4 Results

4.1 Effectiveness of Purely Random Data Fuzzing Media Players

To maintain comparability to typical methods of fuzzing (sending purely random data) we also sent the direct output of the miner to the media players. Not surprisingly, none of the completely random files were able to crash the application except when the file

was detected as RawDV format. In this project we are only focused on crashes produced with the formats of AVI, MPEG, and Ogg so the RawDV crashes are not considered. For our purposes of identifying bugs using the AVI, MPEG and Ogg formats, the completely random files are not useful. Additionally, we were unable to analyze the RawDV crashes with gdb. The bug that caused these crashes was fixed in mplayer-1.0_rc2, which was the version we needed to upgrade to so that we could run gdb.

4.2 Summary of Media Formats and Media Player Results

In Figure 2, we present the percentage of files which caused the target applications to crash, broken up by media format and file construction method. The results of sending both randomized real files (recycled) and structured random files (throwing stars or shuriken) are shown below. As a generalization, we were able to crash the applications more frequently using the strategically structured random files. Unfortunately, a greater number of crashes is not indicative of finding a greater number of unique bugs. Note that some bugs were discovered by multiple levels of recycling and blacksmithing. Overall, we found 21 unique bugs.

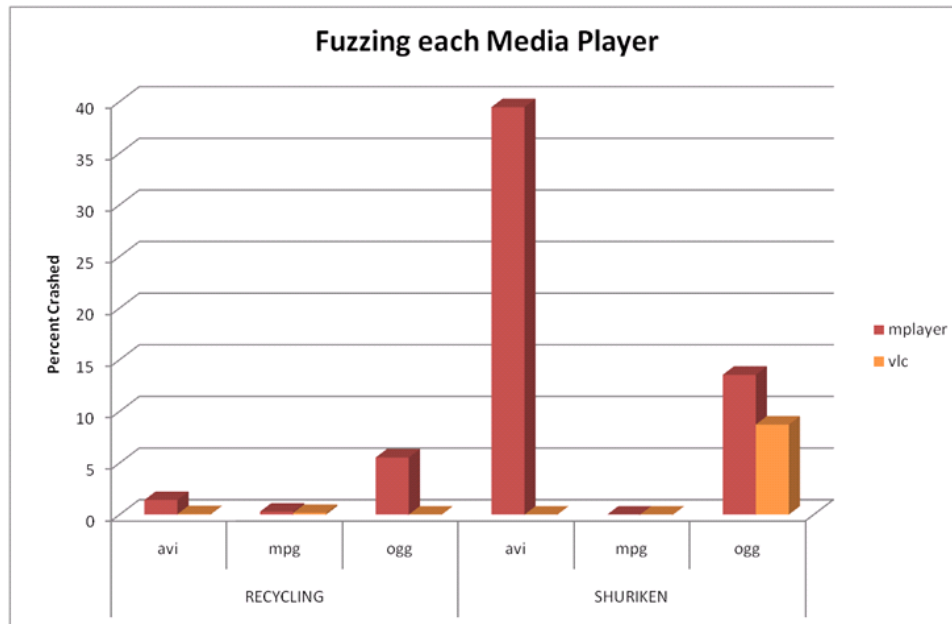


Figure 2: Comparison of media player crashes separated by media format (AVI, MPEG, Ogg) and file construction method (recycled, blacksmith).

4.2.1 AVI Performance

There was a significant disparity in the incidence of crashing for AVI. For the structured random AVI files sent to MPlayer the application crashed 39.5% of the time. In comparison, MPlayer only crashed 1.4% of the time for the recycled AVI content. We believe that the crash rate for the recycled content is a reasonable metric for the stability of the application; the same can not be said for the structured files. The structured random files in the case of AVI were able to narrow in on a particularly weak point, specifically one bug. See section 4.6 for a specific assessment of that and all other bugs.

4.2.2 MPEG Performance

In general, the MPEG files had very little success crashing the applications. With the corrupted real MPEG files, we were only able to crash MPlayer and VLC less than 10 times, combined. None of the structured random MPEG files ever crashed MPlayer or VLC. Due to the lack of density of MPEG crashes, the MPEG format is given less focus throughout the paper.

4.2.3 Ogg Performance

Although AVI had the single highest incidence of crashing, we found the greatest number of unique bugs in the Ogg implementations. Even though Ogg had more structured trials, it had the same number of recycled trails as AVI and MPEG, and had more unique bugs due to recycling. These results suggest either a general instability or greater complexity of the Ogg format implementations.

4.2.4 Comparison of MPlayer and VLC

It is noteworthy that there was drastically different incidence of crashes for MPlayer and VLC. While media players often share libraries for some of the media format-specific functionality, the discrepancy between percent crashed on MPlayer versus VLC suggests the bugs we found were not in the shared components.

4.3 Evaluation of Adding Structure to Random Files

When adding structure to random files, we delineated varying amounts of structure with levels. Although each successive level does not provide a linear increase in structure, files produced within a level have the same amount of format correctness and vary only in the

accompanying random data. Each level of structure is highly tuned and is attacking a specific slice of the program code. Our results show that if a specific level of structure discovered a bug with many crash scenarios, then it found that bug numerous times. The limitation of this method is that the fuzzing takes place at a depth of code determined by the blacksmith creator, which might not be granular enough to cover the code properly.

4.3.1 Adding AVI Structure to Random Files

The AVI results from adding structure are a clear indication that adding specific format to random files can produce a narrow band of testing. Although the incidence of crashing was very high, 39.5% in MPlayer for Level 2, this only found one unique bug. The results for the AVI crashes from structured data are in Figure 3.

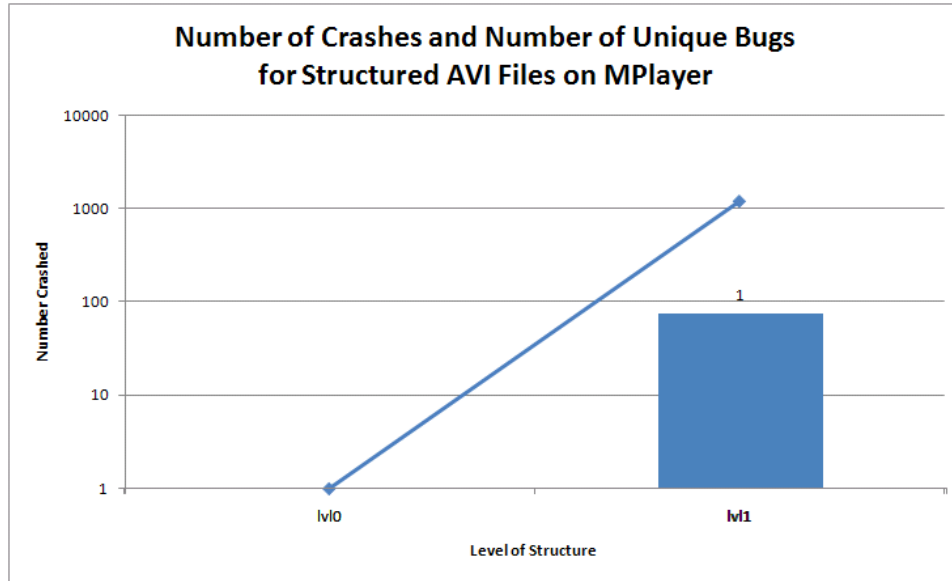


Figure 3: *Number of crashes and unique number of bugs from 1500 files run per level of AVI structure added. The line shows the percentage crashed for each level of file corruption. The bars show the number of unique bugs causing the crashes at each level of file corruption.*

4.3.2 Adding Ogg Structure to Random Files

The Ogg results from adding structure were able to produce more of a range of bugs than did adding structure to AVI, but not by much. We cannot do a comparison of structured Ogg to structured AVI since there is no objective quantification of what a level of structure

means across all file types. The results for the Ogg crashes from structured data are in Figure 4.

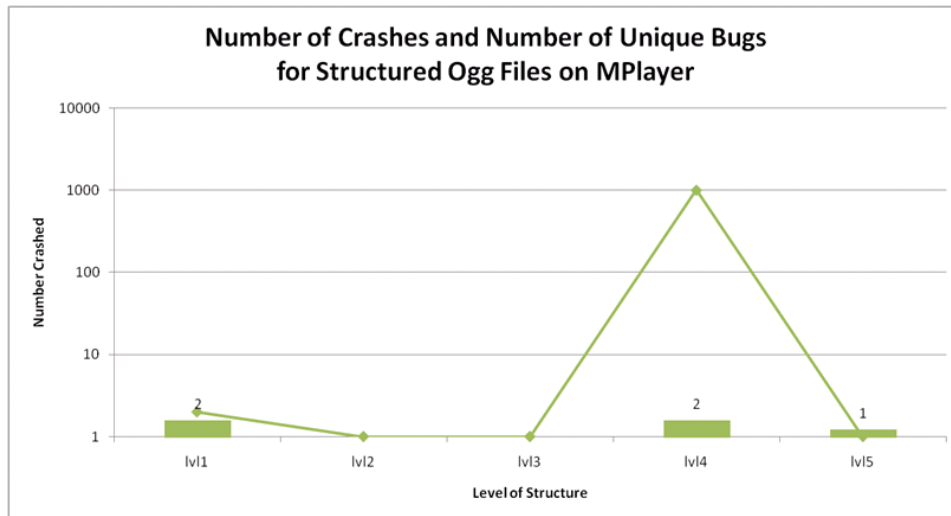


Figure 4: Number of crashes and unique number of bugs from 1500 files run per level of Ogg structure added. The line shows the percentage crashed for each level of file corruption. The bars show the number of unique bugs causing the crashes at each level of file corruption.

4.4 Evaluation of Recycling Files

Through our method of corrupting real files we tried to identify an optimal range of corruption. As the percentage of randomness added approaches 100%, the files will be comparable to the completely random files and will not be successful at identifying bugs in media players. In the figures below showing the percentage of crashes from AVI and Ogg versus percentage change, we can see that the bit-wise modifications reach a level of zero crashes at a lower percentage of change than the byte-wise modification. This is not surprising because the bit-wise changes can be more distributed through the file. Unfortunately, our data does not indicate whether bit-wise or byte-wise modifications are more effective. It is our hypothesis that they serve slightly different purposes. For example, if a byte is expected to be within a certain range, changing one bit within that byte will more likely still fall within the accepted range than if the entire byte was overwritten. Conversely, changing a single bit in a byte might trigger a bug due to the overall byte being close to what the program expects.

4.4.1 Comparison of Percentage Crash and Unique Bugs

The percentage of files crashing can be slightly misleading. For example, the structured AVI files caused 39.5% of files to crash on MPlayer, but all of these crashes were caused by only one unique bug in the AVI format. In the graphs below, while presenting the incidence of crashes for each corruption level, we also provide the number of unique bugs found at each level. In many cases, the frequency of crashes is correlated with a greater number of unique bugs at that level, but this is not guaranteed. In particular, we found that the method of adding structure to random files was likely to crash a high percentage of time on the same bug. While in some scenarios this narrow band of testing may be helpful, in general the method of adding structure seems to only narrow in on a specific section of code without providing broad test coverage. It is likely that this method of strategic testing is comparable to current (non-fuzzing) black box testing methodologies in place, which are time intensive and limited by the creativity of test creators.

4.4.2 Bit-wise Modified Files - Total Number of Bugs for Each Percentage Change

In Figures 5 and 6, we present the results for bit-wise modification of AVI and Ogg files. The MPEG data was too sparse to provide a meaningful comparison and is therefore omitted. For the bit-wise modification, AVI was able to produce crashes only up to the 0.5% rate of corruption while the bit-wise modifications of Ogg produced crashes up to the 1% rate of corruption. This may suggest that AVI has stricter formatting checks and so will stop recognizing a file as valid earlier in the corruption process than will Ogg. Alternatively, Ogg may have more features that are susceptible to bugs. Since the number of unique bugs is very small, the differences between these two demand further testing.

4.4.3 Byte-wise Modified Files - Total Number of Bugs for Each Percentage Change

In Figures 7 and 8, we present the results for byte-wise modification of AVI and Ogg files. Both AVI and Ogg reached points of no crashes at 10% byte-wise modification. This might be used as a rough approximation of a range of percentage byte-wise corruption to focus on if fuzzing an alternate media file format. For both bit-wise and byte-wise modifications, it is noteworthy that we were able to produce crashes with as little as 0.02% change and sometimes even as little as 0.002% change. On the low range of percentage change, it is likely that successes will involve the media player playing something similar to the correct file. On the high range of percentage change, it is likely that successful media players will simply refuse to play the file. One reason that the MPEG file format was so difficult to

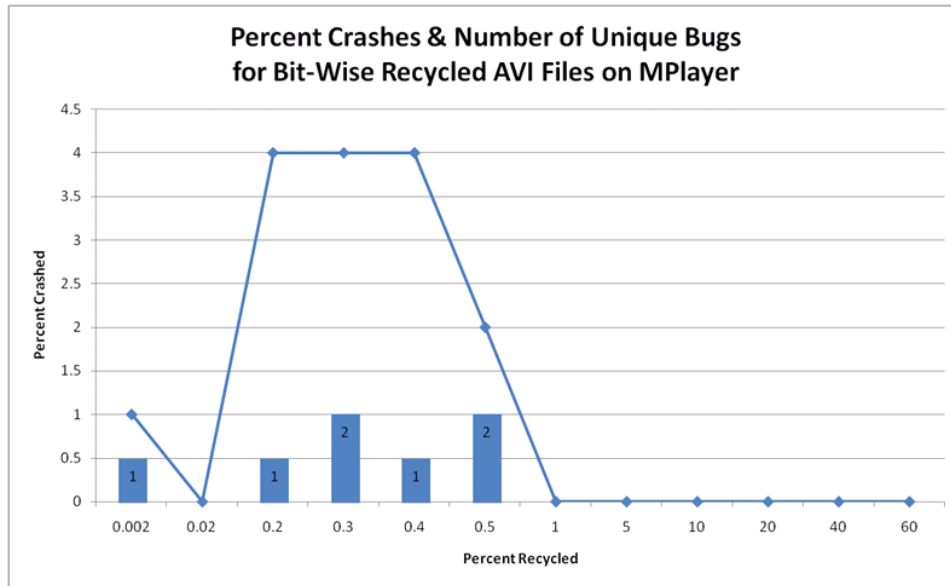


Figure 5: Comparison of the percentage crashes and the number of unique bugs in bit-wise recycled AVI files. The line shows the percentage crashed for each level of file corruption. The bars show the number of unique bugs causing the crashes at each level of file corruption.

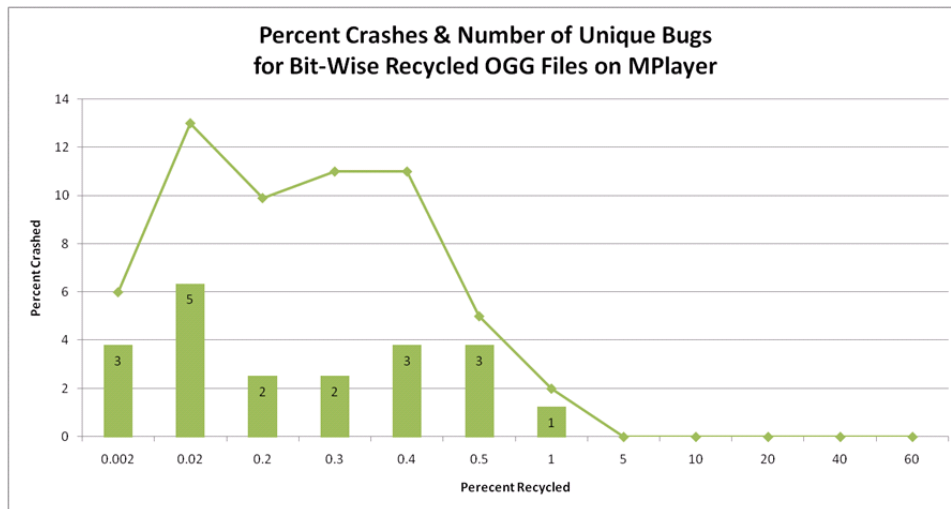


Figure 6: Comparison of the percentage crashes and the number of unique bugs in bit-wise recycled Ogg files. The line shows the percentage crashed for each level of file corruption. The bars show the number of unique bugs causing the crashes at each level of file corruption.

crash was that at even rather high rates of change the media player played some version of the corrupted file without crashing.

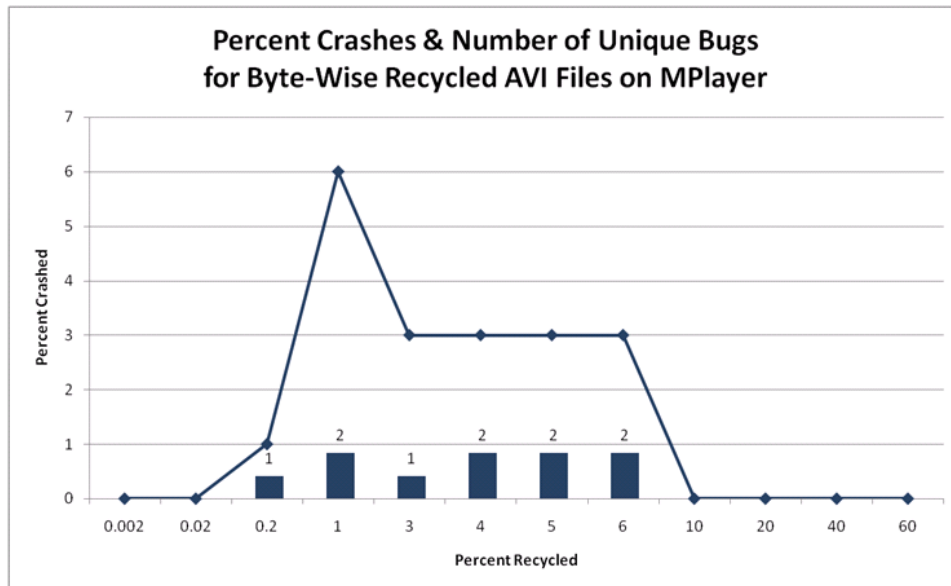


Figure 7: Comparison of the percentage crashes and the number of unique bugs in byte-wise recycled AVI files. The line shows the percentage crashed for each level of file corruption. The bars show the number of unique bugs causing the crashes at each level of file corruption.

4.5 Comparison of Recycler and Blacksmith

We used two methods to send structured random data to the media player. A comparison of these two methods found that sending structured random data was more effective in producing the greatest number of crashes. This method involved researching each format in great detail to properly format the random files and to be able to speculate about the possible weaknesses of the format. The method of adding structure to random files had a significant learning curve, but offers promising results for testers with more knowledge of the media format. At a minimum, this process would be substantially easier for a media file expert, who might be developing a media player. It is our belief that to achieve more targeted results, adding structure to random files would provide a more effective method. In contrast, it is much more time efficient to modify correct media files as a method of identifying bugs. Corrupting files with random bits or bytes requires no knowledge of the file format, and therefore the same process for randomizing an AVI file could be used for any media file.

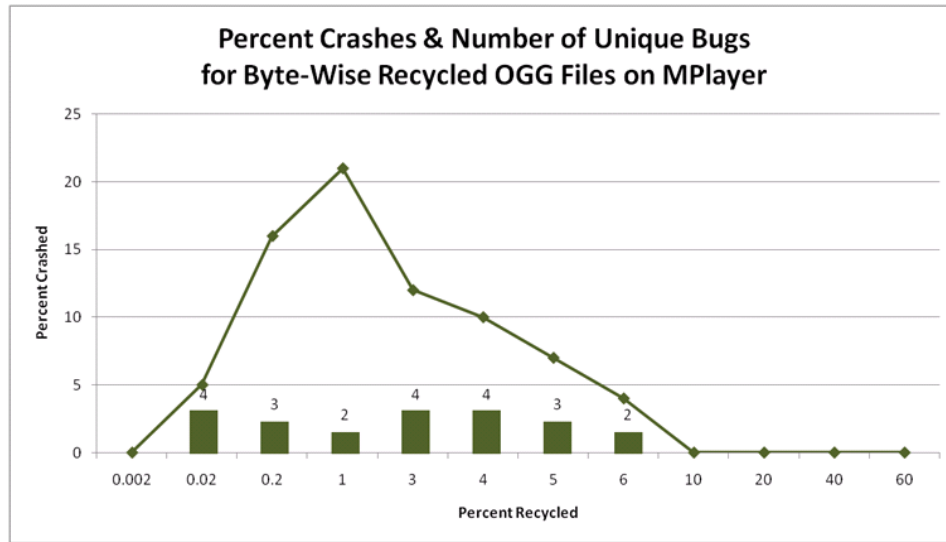


Figure 8: Comparison of the percentage crashes and the number of unique bugs in byte-wise recycled Ogg files. The line shows the percentage crashed for each level of file corruption. The bars show the number of unique bugs causing the crashes at each level of file corruption.

4.6 Analysis of Bug Cause

As a component of our analysis we have categorized all unique bugs by cause. In the table below, bugs are identified as five potential bug types:

1. Accessed invalid array index
2. Dereferenced null pointer
3. Divide by zero
4. Pointer out of bounds
5. Unknown

The greatest numbers of bugs were in dereferencing null pointers. Through fuzzing we were able to produce inappropriate memory access, which indicates that we might have the possibility of writing to memory that is not intended to be written.

Format	Bug Type	Attack Type	Source File	Source Function
AVI	Dereferenced null pointer	Blacksmith - level 1	aviheader.c:197	read_avi_header
AVI	Dereferenced null pointer	Recycler (.2-6%)	demux_avi.c:114	demux_avi_read_packet
AVI	Dereferenced null pointer	Recycler (.5%)	ad_pcm.c:23	init
AVI	Unknown / Varied	Recycler (.002%)	varied	varied
MPEG	Dereferenced null pointer	Recycler (10%)	motion_comp_mmx.c:546	MC_put_o_16_mmxext
MPEG	Dereferenced null pointer	Recycler (3%)	motion_comp_mmx.c:600	MC_put_y_16_mmxext
MPEG	Unknown	Recycler (5-40%)	libmpeg2_plugin.so	__udivdi3
OGG	Accessed invalid array index	Blacksmith - level 4	Bitwise.c:324	oggpack_read
OGG	Accessed invalid array index	Blacksmith - level 4	Bitwise.c:320	oggpackB_read
OGG	Dereferenced null pointer	Blacksmith - level 1	vp3.c:595	vp3_decode_frame
OGG	Dereferenced null pointer	Recycler (.02%)	codebook.c:158	vorbis_book_decode
OGG	Dereferenced null pointer	Recycler (.02%)	info.c:133	vorbis_info_clear
OGG	Dereferenced null pointer	Recycler (.2%)	demux_ogg.c:474	demux_ogg_add_packet
OGG	Dereferenced null pointer	Recycler (.2%)	codebook.c:158	vorbis_book_decodevv_add
OGG	Dereferenced null pointer	Recycler (.3%)	info.c:199	vorbis_synthesis_headerin
OGG	Dereferenced null pointer	Recycler (.5%)	oggparsetheora.c:123	theora_gptopts
OGG	Divide by zero	Recycler (.2%)	res012.c:271	res2_inverse
OGG	Divide by zero	Recycler (.5%)	mathematics.c:65	__divdi3
OGG	Pointer out of Bounds	Blacksmith - level 1	oggparsevorbis.c:51	vorbis_comment
OGG	Unknown	Recycler (.02%)	bitstream.h: 694	read_huffman_tree
OGG	Unknown	Recycler (.02%)	bitstream.h: 885	unpack_vlcs

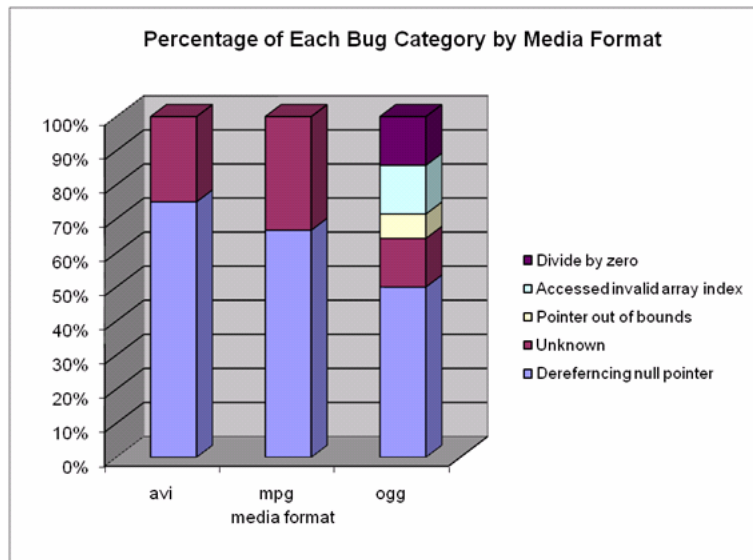


Figure 9: Percentage of each classification of bug for AVI, MPEG and Ogg

5 Related Work

Fuzzing as a quick and easy form of black box testing came into being on a dark and stormy night, literally [1]. Miller et al. first used fuzzing as a means to test UNIX utility programs after noticing noise on a telephone line caused their utilities to crash. Their work led to the discovery of numerous program bugs in a variety of UNIX platforms, demonstrating that fuzzing is a viable means of testing for bugs and a valuable tool to add to the assortment of testing procedures and formal verification techniques already in use. Since that time, work has been done in fuzzing various programs, including both command line and GUI-based programs on UNIX, Windows, and Mac-OS platforms with successful findings of bugs in all cases [3,4,5]. Bugs found by fuzzing are sometimes found to be security vulnerabilities [7,8,9,10].

More recently David Thiel applied fuzzing techniques to audio players [5,6]. In his research, he found that media players often look for certain formatting signposts (headers, checksums) before attempting to play the content. If those signposts are missing, the players will merely exit gracefully. These barriers made it difficult to attain any level of code coverage with purely random data. Thiel took the approach of modifying the media player's code to remove the checks that barred unformatted files from being played. In our work we attack video players from a different angle. We contend that, given the complex nature of media player code, any modifications to the code may have unforeseen consequences that affect the validity of any testing results. Therefore, we add a minimal layer of structure to a random file and use that to fuzz the media player. This is a slight divergence from the philosophy of fuzzing as a quick and dirty way to conduct useful black box testing, but we believe it is a useful one.

There are a number of fuzzing tools that have been written for different applications ([11,12] provide lists of existing fuzzers). Many of them are for fuzzing network protocols or web-based programs. To our knowledge, Thiel's work and zzuf [13] are the only fuzzing tools that target media applications. As mentioned above, our framework takes a different approach to fuzzing than Thiel's. Zzuf provides a framework for adding randomization to valid media files and then using those to fuzz media players, and it can work on other types of applications. This is just one aspect of our framework. In terms of our project, it acts mostly as a combination of a one-time recycler and ninja or a stand-alone recycler.

6 Future Work

6.1 Memory Guards

Although we have identified a number of bugs in MPlayer and VLC and identified places where array bounds were not checked or pointers were dereferenced without being checked, we only identified those locations because the random data happened to cause the program to crash. This indicates that user-provided data is not being thoroughly checked, but there may be undetected instances where memory is written with user input that should not be. Running the media players with our fuzzed data while using a memory guard tool such as Valgrind would let us determine whether or not fuzzed data was causing the program to access memory insecurely in the cases where the program does not crash.

6.2 Taint Tracking

We feel that the most promising addition to this project would be to run a taint analysis tool while playing the files that crashed. The majority of data processed by media players is “tainted” data provided directly by the user, providing enormous opportunities for an adversary to be able to control the program. A taint guard analysis would help us differentiate security vulnerabilities from simple bugs, by isolating files that can cause the media player to write user data.

6.3 Bug Reporting

We discovered several bugs, though we do not know if they can be exploited. Regardless, our next step will be to install the latest development versions of the target applications and see if the bugs still exist. If so, we will report them to the developers.

6.4 Non-Deterministic Bugs

One of the bugs with an unknown cause did not occur every time the file was played in MPlayer, and when it did crash, it crashed at different locations. We would like to inspect this more closely. Also, this is probably not an isolated occurrence. There may have been several other files that could potentially cause a crash. We would like to pursue this further by running the same trial repeatedly to see how often this occurs, and why.

6.5 Code Profiling

We would like to use profiling tools to better measure the amount of code covered by the recycled and structured files. From the crash logs, we know where we found bugs, but we do not know where we did not look. Intuitively, we could aim for functions by defining a certain amount of structure, but that does not tell us where we actually fuzzed. Knowing where we fuzzed would lead to a better evaluation of the usefulness of recycling and structure at testing different parts of an application.

7 Summary and Conclusions

We used fuzzing techniques to find potential vulnerabilities in media players. We sent thousands of random files, structured random files, and randomly corrupted real files to two media players: MPlayer and VLC. We found the method of sending strictly random files was not effective at finding bugs in media players because the applications needed to detect a format before they will attempt to play a file.

We were able to reach code beyond the format-requirement barriers by providing some level of structure to random files and by randomizing fully structured files. We explored the viability of two methods for providing structure to media files. The first, adding structure to completely random data, resulted in the greatest incidence of crashing but did not find the greatest number of bugs. This is only an advisable method of fuzzing if the format requirements are well known or if the need is to deeply exercise specific components. The second method, corrupting real files, reaches a great breadth of bugs and is advisable if you have insufficient knowledge of the format, need to test many different formats, or have very little time.

The current framework we have developed also could be used to fuzz any command line application with randomized real files. To add structure to real files, a new blacksmith would need to be created for each new format you wish to fuzz.

Regardless of the method of applying structure, our fuzzing techniques discovered real bugs in format-aware applications. Similar techniques will serve as a useful, preventative security tool in the future, especially when attackers shift their focus to the video vector.

8 References

- [1] B.P. Miller, L. Fredriksen, and B. So. “An Empirical Study of the Reliability of UNIX Utilities”. *Communications of the ACM* 33, 12, December 1990.

- [2] B.P. Miller, D. Koski, C.P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. “Fuzz Revisited: A Re- examination of the Reliability of UNIX Utilities and Services”. Computer Sciences Technical Report #1268, University of Wisconsin-Madison, April 1995.
- [3] J.E. Forrester and B.P. Miller. “An Empirical Study of the Robustness of Windows NT Applications Using Random Testing”. 4th USENIX Windows Systems Symposium, Seattle, August 2000.
- [4] B.P. Miller, G. Cooksey and F. Moore. “An Empirical Study of the Robustness of MacOS Applications Using Random Testing”. First International Workshop on Random Testing, Portland, Maine, July 2006.
- [5] D. Thiel. “Exposing Vulnerabilities in Media Software”. iSEC Partners, BlackHat USA, August 2007. <https://www.blackhat.com/presentations/bh-usa-07/Thiel/Whitepaper/bh-usa-07-thiel-WP.pdf>.
- [6] D. Thiel. “Exposing Vulnerabilities in Media Software”. iSEC Partners, BlackHat USA, August 2007. (presentation) http://www.isecpartners.com/files/iSEC_Thiel_Exposing_Vulnerabilities_Media_Software_bh07.pdf.
- [7] M. Zalewski. “Web browsers - a mini-farce”. October 2004. <http://archive.cert.uni-stuttgart.de/bugtraq/2004/10/msg00180.html>.
- [8] United States Computer Emergency Readiness Team. “Vulnerability Note VU#842160”. November 2004. <http://www.kb.cert.org/vuls/id/842160>.
- [9] Common Vulnerabilities and Exposures. “CVE 2006-1185”. March 2006. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-1185>.
- [10] Ipllosion Security. “Internet Explorer Fuzzing and Microsoft Incident Handling”. October 2006. <http://www.iplosion.com/archives/21>.
- [11] DarkNet.org.uk. “Fuzzing”. November 2007. <http://www.darknet.org.uk/tag/fuzzing/>.
- [12] Ethical Hacking and Penetration Testing. “Fuzzers- The ultimate list”. December 2005. <http://www.infosecinstitute.com/blog/2005/12/fuzzers-ultimate-list.html>.
- [13] S. Hocevar. “zuff - multi-purpose fuzzer”. January 2007. <http://sam.zoy.org/zzuf/>.
- [14] Xiph.org. “Ogg Documentation”. November 2007. <http://www.xiph.org/ogg/doc/framing.html>.