Exposing Vulnerabilities in Media Software

David Thiel, iSEC Partners

BlackHat USA, August 2, 2007

Abstract

Deep media codec fuzzing presents a rich opportunity for turning up hard to find bugs in media players, codecs and other unexpected software, and can be a useful tool for developers to ensure the robustness of code. It also requires techniques a bit different than those used in traditional, bit-flipping file fuzzers. This paper explores possibilities, techniques and results of media codec fuzzing and exploitation, using several modern (and some antiquated) audio codecs as examples.

1 Introduction

The attack surface of audio and multimedia software is quite broad. Generally, desktop users tend to have a fairly small number of programs that are used on an almost constant basis - web browsers, Instant Messengers, e-mail readers and media players. Of those, media players have been underexplored as an attack vector. There have been simple overflow exploits of long playlists, filenames or HTTP responses, but very little has been done that is specific to media files themselves.

Audio in particular is an attractive attack vector because:

- Media players are very frequently used software; users tend to use them for an extended period of time, leaving them open during other tasks, and frequently switch media streams.
- There are a wide variety of different audio players, and many of different codecs and audio file plugins - all written by generally non-security-focused people.
- The file formats involved are binary streams, and tend to be reasonably complex. Much parsing is required to manipulate them, and codec calculations can easily result in integer bugs.
- Players take untrusted input from many different unreliable sources (often over the network), and run with fairly high privilege and priority. For instance, in Windows Vista, a low-privileged IE instance can launch content in a higher-privileged WMP.
- They're perceived as relatively harmless - users are likely to play files given to them.
- They're frequently invoked without the user's explicit acknowledgement, (i.e. embedded in a webpage)
- Media content is rich in metadata and external references. Media players behave more and more like web browsers.

There are also some challenges in this area of research. An attacker has to actually understand the file formats relatively in depth; ordinary file fuzzing may turn up bugs, but will more often simply cause the player ignore the file. Several compressed formats require CRCs to be correct before attempting to play a frame, and much of the frame header must be correct for the player to accept the input. These things require "smart" fuzzing, making our fuzzer format-aware.

Another challenge is that existing libraries for manipulation of audio data generally choke on fuzzed input, limiting their usefulness in a fuzzing framework. Errors will be thrown on encode if header data is grossly wrong, or file writes will fail due to calculations based on number of frames and sample rate. This leaves us with five options - either hack the libraries to remove error checking and some internal logic, write our own libraries, or edit the binary data directly ourselves. I've taken a mixed approach - fuzzing metadata (like id3 tags) using python with modified libraries, and fuzzing frame header data by loosely implementing the codec specification.

The examples in this paper will illustrate some of the techniques used to build a basic media codec fuzzer in Python, using a new tool, Fuzzbox, as an example. I've focused on primarily open-source players, because this is the software I'm most interested in improving. Of course, the same techniques also work on proprietary software.

2 Why fuzzing matters, despite what static analysis vendors say

Media codecs are complicated, and the code underlying them is often difficult. A manual code review would miss many bugs, and static analysis, while beneficial, can miss some tricky issues like integer overflows and uninitialized variables. For example:

mdct_bitreverse at mdct.c:377

```
w0[0]  = r0     + r2;
w1[2]  = r0     - r2;
w0[1]  = r1     + r3;
w1[3]  = r3     - r1;

x0     = x+bit[2];
x1     = x+bit[3];

r0     = x0[1]  - x1[1];    <-- SIGSEGV here. Easy to miss.
r1     = x0[0]  + x1[0];
r2     = MULT_NORM(r1     * T[2]   + r0 * T[3]);
r3     = MULT_NORM(r1     * T[3]   - r0 * T[2]);
r0     = HALVE(x0[1] + x1[1]);
```

No one can blame a developer for not catching a problem in a function like this. This is why we fuzz.

3 Fuzzing Techniques

The actual types of fuzzed data that we use will be the same as many other fuzzers; random garbage data of unusual sizes and types. We'll start with a basic function to output useful malicious data. The randstring() will return one of the following types of bad data:

- A random string of a single character, of a random length
- A string full of %n formats to agitate format string bugs
- A string of random ASCII, of random length, including non-characters
- A random alphanumeric string, of random length
- A string of random Unicode, with a 50/50 chance of being properly encoded or raw
- A large signed random integer
- NULLs

- Fencepost numbers
- HTML, to expose any interpreted HTML in GUIs
- URIs to detect auto-fetched content

The weightings for each of these can be easily changed with the Python random.randint() function.

4 The fuzzer's toolbox

There are a few nice tools to make this process easier, discovered in the course of this project.

- Hachoir[1]: a very nice file format dissector in Python. Hachoir will visually help us disassemble and make sense of file heathers, without having to say, read documentation.
- Mutagen[2]: a multi-purpose audio metadata editing tool. Good for fuzzing metadata, although its strictness and insistence on UTF-8 encoding can require a lot of patching.
- vbindiff[3]: a visual binary diffing tool, to help us narrow down which areas have been changed in fuzzed files.
- bvi[4]: or any hex editor. This one has the advantage of having similar keybindings to a certain one true editor.
- bbe[5]: sed for binaries - understands hexadecimal. Good for inserting shellcode in place of / after fuzzed data, or changing offsets.
- gdb: love it or hate it, it's all you get.

5 Audio Formats Examined

A brief overview of the particulars of some of the audio formats that we'll be attacking and their differentiating features.

5.1 Ogg-Vorbis

5.1.1 Vorbis comment tags

Several Ogg-contained formats (FLAC, Vorbis, Speex) use a common, format for media stream metadata, unhelpfully dubbed "Vorbis Comment" headers.[6] There are several tools available for the manipulation of Vorbis comments; the one used here is py-vorbis.[7] The only modification we need to make to py-vorbis is to alter the size of tag_buff to be something unreasonably large.

[1] http://hachoir.org/
[2] http://www.sacredchao.net/quodlibet/wiki/Development/Mutagen
[3] http://www.cjmweb.net/vbindiff/
[4] http://bvi.sourceforge.net/
[5] http://sourceforge.net/projects/bbe-/
[6] http://www.xiph.org/vorbis/doc/v-comment.html
[7] http://ekyo.nerim.net/software/pyogg/

5.1.2 Ogg Frame Data

An Ogg stream is composed of a series of multiple pages (sometimes referred to as frames), each of which contain a header which is used by the player to determine the number of channels, the sample rate, current position in a stream, and a CRC of the entire page. This data helps the player to recover from stream interruption and data corruption, and additionally makes the Ogg format fairly resistant to simple file fuzzing. There are so many discrete components of an Ogg page, each with their own error checks, that using the py-ogg library as a tool to fuzz with is impractical. Instead, we’ll need to write our own Ogg page constructor.

To do this, we make a simple dictionary of elements as with Vorbis comments, only constructed from data from an actual Ogg-Vorbis file. We do this as follows:

```
f=open(sourcefile, ’rb’)
y = {}
y[’01magic’] = f.read(4)
y[’02version’] = f.read(1)
y[’03headertype’] = f.read(1)
y[’04granulepos’] = f.read(8)
```

And so on. We can then feed this to our generic fuzzing routine, using Python’s struct[8] to pack the resulting data back into binary format:

```
thestring = ""
letsfuzz = random.choice(y.keys())
print "fuzzing %s"%letsfuzz
thestring = randstring()
stringtype = type(thestring)
length = len(y[letsfuzz])
if str(stringtype) == "<type ’str’>":
    y[letsfuzz] = struct.pack(’s’, thestring[:length])
elif str(stringtype) == "<type ’int’>":
    y[letsfuzz] = struct.pack(’i’, thestring)
else:
    thestring = ""
    for i in range(len(y[letsfuzz])):
        thestring += "%X" % random.randint(0,15)
```

We now have an Ogg with a fuzzed first frame header, with the rest of the original file appended. This however will find us almost no bugs, as the failure in CRC check will quickly stop playback. To fix this, we’ll borrow the crc lookup table from libogg’s framing.c[9], and recalculate the CRC of the fuzzed page:

```
# This excludes the CRC from being part of the new CRC.
page = page[0:22] + "\x00\x00\x00\x00" + page[26:]
for i in range(len(page)):
crc_reg = ((crc_reg<<8) & 0xffffffff) ^ crc_lookup[((crc_reg >> 24) & 0xff) ^ ord(page[i])]
# Install the CRC.
page = page[0:22] + struct.pack(’I’, crc_reg) + page[26:]
return page
```

The new CRC has now been installed in the appropriate location, and the file should either play correctly or expose bugs.

[8] http://docs.python.org/lib/module-struct.html

[9] This is necessary because Ogg does not use CRC32 as its CRC mechanism - otherwise we could use standard Python libraries for this.

5.2 WAV/PCM and AIFF

WAV and AIFF are fairly simple file formats, with relatively little included metadata. Fuzzbox altered Python wave and AIFF modules to fuzz the sample width, framerate, frame number, number of channels, frame number, and compression type. These types of changes are good for turning up integer overflows/underflows.

5.3 MP3

5.3.1 ID3 Tags

While the ID3v1 standard somewhat resembles Vorbis comments, ID3v2 tags are significantly more structured and complex. Rather than being arbitrary name/value pairs, the ID3v2 standard consists of an excessively large number of declared “frames”, which are of variable size and can be integers, text, or even binaries like images. The vast bulk of the ID3v2 specification is not implemented by most modern players, however; most either only utilize the standard ID3v1 tag categories, converted to the ID3v2 format, or only read ID3v1 comments, which usually coexist with ID3v2 tags. Like Vorbis comments, ID3v2 tags are stored at the beginning of a file rather than the end, to support streaming listening.

Because ID3v1 is inherently length limited, we’re not likely to find buffer overflows here, but there are still some possibilities. We’ll be using mutagen to do the metadata manipulations.

5.3.2 Frame Data

Unlike in Ogg-Vorbis, CRCs are optional in MP3, and fairly infrequently used. When they are used, it’s not uncommon for players to ignore them. So, we can generally fuzz frame data with impunity.

5.4 FLAC

FLAC utilizes Vorbis comments for media metadata, and uses an internal checksum, often compared to an external md5 fingerprint file. It can also be stored inside an Ogg container. Our fuzzing approach will be similar to that used with Ogg Vorbis. Mutagen will also be used here.

5.5 MPEG-4 / AAC / MP4 / M4A

The MPEG-4 container format encapsulates many types of content, including the AAC audio codec used primarily by Apple. Metadata is stored in hierarchical FOURCC “atoms”. In typical Apple fashion, the format is not publicly documented, but has been more or less figured out by the community at large. Hachoir, Mutagen and http://wiki.multimedia.cx/index.php?title=QuickTime_container were all useful for determining its structure.

5.6 Speex

Speex is a free, speech-optimized codec which makes its way into a number of products. The Asterisk PBX supports the use of Speex, allowing use of a high quality codec without paying licensing fees. Xbox 360 Live voice communications use Speex as a codec as well. Speex is structurally similar to Vorbis, and can also be contained within an Ogg. Vorbis comments can also be used for metadata, but are not commonly parsed.

6 Rounding Up Bugs

Fuzzbox will produce a specified number of fuzzed audio files, optionally spawning a player under GDB (Unix and OS X only, possibly Cygwin) to capture backtraces. This is done by generating a GDB batch file at runtime - any additional info you'd like to harvest can be specified by adding to the playit() function.

Here are some examples of bugs exposed by testing with Fuzzbox.

6.1 libvorbis

Fuzzbox turns up multiple issues in libvorbis, in PCM blocking (block.c), header packet processing (info.c), and mdct.c, which apparently does "normalized modified discrete cosine transform power of two length transform only [64 <= n]". These don't appear to be particularly exploitable for code execution - all we can get out of it is a denial of service. While this isn't terribly high impact, Ogg Vorbis is fairly widely used in third party products - for example in video games. Imagine a scenario where a game allows users to provide their own custom sound effects or taunts - a good way to take out your enemies.

Bugs in libvorbis affect the majority of Ogg-capable media players - some are even portable across independent implementations.

6.1.1 One: Blowing up the heap

This is a bug in residue/floor calculations, caused by the code's failure to handle a case not covered by the specification. The fix for this not only required changes to the code, but to the specification - Tremor, another Vorbis decoder, turned out to be affected as well.

```
Program received signal SIGSEGV, Segmentation fault.
0x280ad393 in _01inverse (vb=0x806f244, vl=0x80664c0, in=0xbfbfe650, ch=1, decodepart=0x280b0afd
<vorbis_book_decodev_add>) at res0.c:648
648 if(info->secondstages[partword[j][l][k]]&(1<<s)){
(gdb) bt
#0 0x280ad393 in _01inverse (vb=0x806f244, vl=0x80664c0, in=0xbfbfe650, ch=1, decodepart=0x280b0afd
<vorbis_book_decodev_add>) at res0.c:648
#1 0x280ad8f2 in res1_inverse (vb=0x806f244, vl=0x80664c0, in=0xbfbfe650, nonzero=0xbfbfe630, ch=1)
at res0.c:772
#2 0x280af8b5 in mapping0_inverse (vb=0x806f244, l=0x80b9000) at mapping0.c:783
#3 0x280a1aa5 in vorbis_synthesis (vb=0x806f244, op=0xbfbfe770) at synthesis.c:76
```

6.1.2 Two: When specs change

This is a read violation which has its root cause in a change in the Vorbis specification. Early versions of the specification allowed block sizes to be 8 samples or larger, but the final version requires they be at least 64 samples. Different parts of the code make different assumptions.

```
(gdb) bt
#0  0x2809e32c in vorbis_synthesis_blockin (v=0x806f1dc, vb=0x806f244) at block.c:774
#1  0x2808da64 in _fetch_and_process_packet (vf=0x806f000, op_in=0x0, readp=1, spanp=1) at
vorbisfile.c:499
#2  0x28090518 in ov_read (vf=0x806f000, buffer=0x8058840 "", length=4096, bigendianp=0, word=2,
sgned=1, bitstream=0x806f2c8) at vorbisfile.c:1545
#3  0x0805031e in ovf_read (decoder=0x8066180, ptr=0x8058840, nbytes=4096, eos=0xbfbfe8a4,
audio_fmt=0xbfbfe8b0) at oggvorbis_format.c:139
#4  0x0804fd8a in play (source_string=0x805a220 "segfaults_realplayer.ogg") at ogg123.c:529
#5  0x0804fa07 in main (argc=2, argv=0xbfbfea10) at ogg123.c:393
```

6.1.3 Three: Cleanup failure

This is an error in cleaning up after detecting an invalid mapping type[10]. The cleanup code itself unfortunately uses the invalid type.

```
Program received signal SIGSEGV, Segmentation fault.
0x280a6c14 in vorbis_info_clear (vi=0x805a260) at info.c:165
165 _mapping_P[ci->map_type[i]]->free_info(ci->map_param[i]);
(gdb) bt
#0  0x280a6c14 in vorbis_info_clear (vi=0x805a260) at info.c:165
#1  0x280a758c in _vorbis_unpack_books (vi=0x805a260, opb=0xbfbfe710) at info.c:327
#2  0x280a770f in vorbis_synthesis_headerin (vi=0x805a260, vc=0x805c440, op=0xbfbfe770) at info.c:380
#3  0x2808d1ef in _fetch_headers (vf=0x806f000, vi=0x805a260, vc=0x805c440, serialno=0x806f05c,
og_ptr=0xbfbfe790) at vorbisfile.c:262
```

It's worth noting that at least one prominent commercial static analysis tool failed to find all three of these problems. This is why fuzzing tools are necessary regardless of other techniques used to harden (or break) software.

6.2 VLC

6.2.1 Multiple segmentation faults in both ogg and wav modules.

An example backtrace:

```
Breakpoint 1, input_vaControl (p_input=0x87d4000, i_query=13, args=0x0) at input/control.c:69 69 in
input/control.c
(gdb) break vasprintf Breakpoint 2 at 0x28469625
(gdb) cont
Continuing.
Breakpoint 2, 0x28469625 in vasprintf () from /lib/libc.so.6
(gdb) where
#0  0x28469625 in vasprintf () from /lib/libc.so.6
#1  0x080d1d93 in input_vaControl (p_input=0x87d4000, i_query=142491908, args=0x87cbbcc
"%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n") at input/control.c:192
```

Looking at the file with ogginfo (which thankfully doesn't have the same problem, we can see this comes from a vorbis comment:

```
REPLAYGAIN_ALBUM_GAIN=albumgain
DESCRIPTION=%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n
LICENSE=Free.
```

We have a classic format string bug. As it turns out, not only is this a problem with vorbis comments, but with CDDA entries and data parsed from SAP/SDP broadcasts[11]. There are also a couple of other bonus issues; an integer overflow caused by an invalid sample rate, and a crash due to an uninitialized variable in input.c[12].

[10] See the Vorbis specification, section 4.2.4.5.

[11] http://www.videolan.org/sa0702.html

[12] http://www.isecpartners.com/advisories/2007-06-08-vlc.txt

6.3 iTunes

Due to an apparrent feeble attempt by Apple to defend their DRM schemes, debugging iTunes under GDB does not work without some changes. Not only is this protection worthless, but it is counterproductive, presenting a barrier to the developers of media codecs and plugins who wish to make their software more robust. Using the technique described athttp://steike.com/HowToDebugITunesWithGdbOnTiger, Fuzzbox can automatically bypass the correct ptrace() system calls, and uses AppleScript to direct the player to load up files and start playing.

The problems mentioned in libvorbis also appear here (through the Ogg Vorbis quicktime filters). Worth noting is that not only can iTunes be made to crash, but the Finder's preview functionality will allow the same bugs to take the Finder with it.

6.4 Windows Media Player

As mentioned previously, Windows Media Player plays Oggs via DirectShow filters, which are an independent implementation of the Vorbis specification. Also of note is the way Windows Media Player is treated under Windows Vista. While in older versions of Windows, both Internet Explorer and Windows Media Player run with the same level of priority. Windows Vista has been changed to run Internet Explorer under a fairly restricted level of privilege, to help mitigate browser attacks. However, when media files are viewed within IE, WMP is spawned at the user's default "medium" privilege level. For this reason, it seems likely that attackers might do well to shift their focus away from exploiting the browser itself to exploitation of external handlers like WMP.

6.5 RealPlayer

RealPlayer does not use libvorbis, but rather an independent implementation of the Vorbis specification. It's not perfect either:

```
Program received signal SIGSEGV, Segmentation fault.
0x29402d18 in ?? ()
*** glibc detected ***
/usr/X11R6/lib/RealPlayer/realplay.bin: malloc():
memory corruption: 0x0851d648 ***
```

There are also copious integer overflows in AIFF playback (and probably others); the best find from the AIFF fuzzing module. Oddly, these same bugs are handled correctly in HelixPlayer, but these fixes do not appear to have been ported to the commercial product. It's also worth noting that there's a rather obvious bug in command line parsing an excessively large number of audio files as argv[1] causes a SEGV. Getting someone to play a big directory of files all at once isn't the easiest attack vector, but it could certainly happen...

6.6 Flac-tools

Let's see what fuzzbox does to the basic player, flac123:

```
Program received signal SIGSEGV, Segmentation fault.
0x27272727 in ?? ()   <-- Kicking it old school.
#0  0x27272727 in ?? ()
#1  0x0804a811 in local__vcentry_matches (field_name=0x804afaf "artist", entry=0x8268038) at
vorbiscomment.c:32
#2  0x0804a9ac in get_vorbis_comments (filename=0xbfbfeb15 "output30.flac") at vorbiscomment.c:69
#3  0x08049564 in print_file_info (filename=0xbfbfeb15 "output30.flac") at flac123.c:121
#4  0x08049a97 in decoder_constructor (filename=0xbfbfeb15 "output30.flac") at flac123.c:245
#5  0x08049b2d in play_file (filename=0xbfbfeb15 "output30.flac")
```

If we open up output30.flac in bvi, we can easily see the source:

```
00000B90  68 71 6C 71 78 7A 72 67 7A 70 6D 75 31 37 66 34 hqlqxzrgzpmu17f4
00000BA0  77 6B 6E 6C 75 79 3D 00 64 03 00 00 74 69 74 6C wknluy=.d...titl
00000BB0  65 3D 27 27 27 27 27 27 27 27 27 27 27 27 27 27 e=”””””””
00000BC0  27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 ””””””””
00000BD0  27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 ””””””””
00000BE0  27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 ””””””””
```

We have a trivial overflow in the title field, for almost no effort. While flac123 might not be the most frequently used way to play FLAC files, there are several players that use the "123" family of tools to play media files - wrapper-style programs like PyTone. An entertaining characteristic of this particular bug is that it can be triggered after the playing of the audio file - allowing the attacker to taunt the victim before launching shellcode.

7 Collateral Damage

Because input parsed from media streams is fed to functions in many different toolkits, simply fuzzing metadata can turn up unexpected surprises. Not all of these avenues are heavily researched and some are entirely speculative, but they show the potential for attack surface beyond C bugs and media players themselves.

7.1 Media metadata as a vector for webapp attacks

There are several web applications that populate page content based on media metadata without adequate sanitization. This is a great vector for CSRF attacks via script injection, or scanning of internal networks via XSS. The first web interface tested against fuzzbox fuzzed files was phpMp, a frontend for MPD - it allowed for injection of script via media files. Besides internally-run apps, several ”social networking” sites allow for user-uploaded media. Do they parse this correctly?

7.2 VOIP

As previously mentioned, the Asterisk VOIP PBX can be configured to use Speex or Ogg Vorbis as codecs. Any DoS or code execution in these codecs potentially means a larger one in the PBX itself.

7.3 Indexing services

Search software like Beagle or other programs that index metadata via third-party libraries make themselves vulnerable to exploits in those libraries as well. This would also mean that simple possession of a malicious file would be enough to trigger these problems, rather than actual playback. Furthermore, media metadata could be a venue to exploit the product itself rather than the parsing libraries - it's worth noting that Beagle has a web interface.

8 Conclusions

Media is not often considered a security-sensitive area; however, the ubiquity and complexity of media codecs makes them especially sensitive to security bugs, some of them obscure and difficult to detect via source review, static analysis or simple fuzzing. Existing tools don't expose these bugs well, because they're not targeted to the stream formats involved.

Because of the vast amounts of untrusted data media software now consumes, it needs to become standard practice for vendors and programmers of media codecs, players and related software to write their own fuzzers for their products, to turn up these issues as part of the development process. Hopefully the information and tools presented here will help to initiate this process.

9 Acknowledgements

Thanks to:

- The VideoLAN and Xiph teams for their responsiveness to bug reports (and helping me understand the code), and to Dan Johnson for stepping up to fix flac-tools. Writing this software requires far more skills and knowledge than breaking it.
- Chris Palmer for his help and time with fuzzbox and debugging.
- Tim Newsham and Jesse Burns for their help in diagnosing some very obscure crashes.