

NOVEMBER 23, 2015

# COMP9447

Security Project Report

Team The logo for the team 'JIM' is composed of stylized, multi-lined letters. The 'J' is formed by several parallel vertical lines that curve at the bottom. The 'I' is a single vertical line. The 'M' is formed by several parallel vertical lines that curve at the top and bottom. A small blue triangle is positioned above the 'I'.

**Authors:**

Jason Ng (z3420276)

Ian Wong (z3463696)

Mohammad Ghasembeigi (z3464208)

# Abstract

An attempt was made to find and exploit vulnerabilities in the multimedia software FFmpeg. The processes used involved fuzzing, and source code analysis. Reverse engineering was not necessary, as it was an open source software.

To find a vulnerability using fuzzing, this meant trying to crash the program by giving an abnormal input. Our fuzzing involved developing our own file mutators and test harness. For source code analysis, it was about finding the possibility of taking advantage over poor memory management through specific malicious input. For this process, we used static code analysis tools, as well as manual reviewing.

Unfortunately (for us), FFmpeg is maintained extremely well - it seemed to respond effectively to abnormal media files from our fuzzer (although some weird audio inputs were accepted and converted successfully), and all memory management functions (such as memcpy, strlen, sscanf) were used properly.

## Summary of Tools Used in project

### Collaboration

**Code repository:** Github

**Issue tracker:** Github Issues

**File storage:** Dropbox folder

**General Communication:** Facebook Group

**Meetings:** Weekly Uni meetups + Skype meetings

**Report writing:** Google Docs, Google Drawings

### Programming Languages

**Test Harness:** Python 2.7

**Fuzzer (Data mutator):** C++11

### Tools

**Hex viewer/editor:** HxD Hex Editor

**File Viewer:** RIFFPad

**Debugger:** GDB

**Memory checker:** Valgrind

**Fuzzing tools:** AFL, FuzzBox, notSPIKEfile, and Basic Fuzzing Framework

**Code Static Analysis tools:** FlawFinder, Visual Code Grepper

# Historical Context

## About FFmpeg



Figure 1: FFmpeg logo

FFmpeg is an open-source, command-line tool used to convert multimedia files to other formats. It is popular due to its open-source nature and is used by many programs like the Pidgin chat client. FFmpeg supports many file formats from obscure ancient formats to new cutting edge formats. Furthermore, FFmpeg can be run on a large number of operating systems including Mac OS X, Microsoft Windows, BSD, Solaris, Android and more.

## Bugs in FFmpeg

FFmpeg constantly changes every day and the development team patch bugs that are reported very quickly. Over the course, we constantly reviewed bug reports and fixes to try and understand which areas of code were often problematic. We discovered that the **libavcodec** and **libavformat** folders were changed many times daily (with many changes being bug fixes) while other core areas were very solid and secure. This information was useful when planning what to do for source code analysis.

The FFmpeg bug tracker website can be found here: <https://trac.ffmpeg.org/>

The FFmpeg github repo that contains all the latest changes can be found here: <https://github.com/FFmpeg/FFmpeg>

Reports typically contain the task that was trying to be accomplished (“I was converting a file from AVI to FLV”), the problem that was encountered (“shit crashed yo!”), the exact command line that was used (ie, `./ffmpeg -i juicy_cucumbers.avi -an -vcodec foo output.flv`) and the input files that triggered the crash. Thus it appears that fuzzing can be valuable in finding uncommon bugs in FFmpeg.

## **Other efforts**

Google itself has revealed about 1000 bugs since FFmpeg became a primary multimedia tool, including:

- NULL pointer dereferences
- Invalid pointer arithmetic leading to SIGSEGV due to unmapped memory access
- Out-of-bounds reads and writes to stack, heap and static-based arrays
- Invalid free() calls
- Double free() calls over the same pointer
- Division errors
- Assertion failures
- Use of uninitialized memory

Fixes and patches have been deployed since 2012.

# Cloud Computing



Figure 2: Net Virtue logo

Cloud computing was used to fast track the fuzzing process. Rather than running all tests on personal machines, our team decided to run our fuzzing script continuously in the hopes of being able to cover many more cases. At first our team considered using Microsoft Azure to set up a python cron job but that option seems too expensive even though it would offer great computational power. Amazon instances were also considered but in the end we decided to utilise a virtual private server owned by one of the group members. This VPS, hosted by Net Virtue in Sydney, was powerful enough to run our fuzzer 24/7 in the background while running web servers and other nifty tools.

```
root@vps:/home/fuzzer
[root@vps fuzzer]# ls
Config.conf  FileVault  Fuzzer.py  MutantHorse  ffmpeg
[root@vps fuzzer]#
```

Figure 3: Standard configuration setup for the fuzzer









Overview		
 103.18.108.██	 57GB SSD Disk	 2 CPU Cores
 Online	 Syd3	 500GB Bandwidth
 1GB RAM	 CentOS 6.x 64bit	

Figure 4: VPS Overview

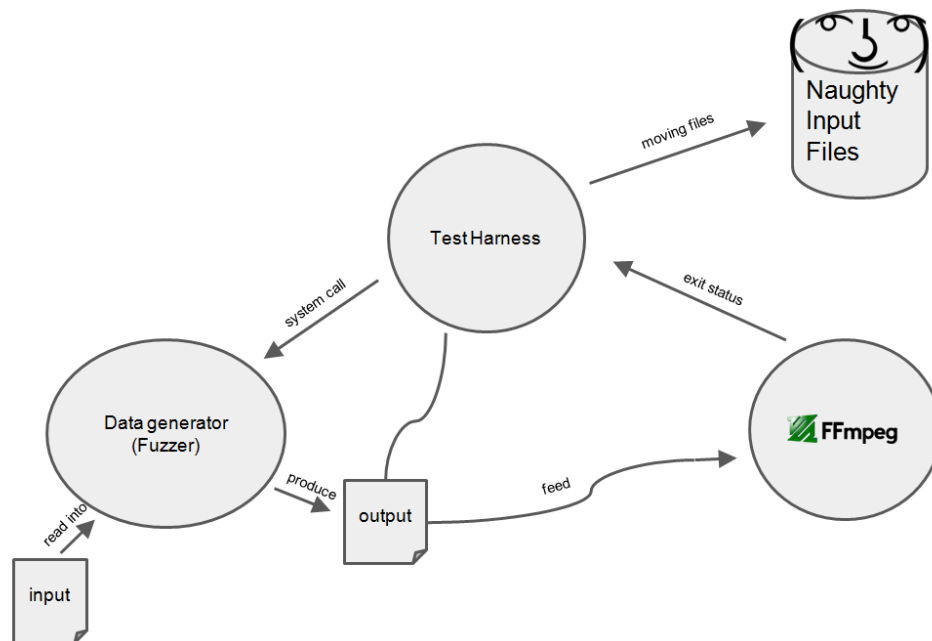
The **screen** program was used to run our process without maintaining an active shell session. The benefit of using screens was that team members could log in and check the status of the tool at any time and we could leave the test harness running all the time. We ran into some problems early on where the test harness script would randomly crash without warning, this was later tracked down to a bug that occurred when files were created (and mutated) too quickly. A change in how we called the fuzzer (MutantHorse) program fixed this issue. Updates to the test harness and MutantHorse were easy to perform. We simply stopped the active process by entering the screen in which it was running. We then looked for any saved files that would indicate that some abnormal behavior occurred. After this we would clear the file storage directories, upload the new programs and restart the test harness script.

# Fuzzing

Our team used two main fuzzers for this project - our own, and American Fuzzy Lop.

## Our Fuzzer

Our first fuzzing tool was one made ourselves, using a python script as the test harness, and a C++ program as the actual data mutator. The C++ mutator is able to support a variety of formats - the code for mutating each format is contained in different files. The C++ mutator alters the data part of the file only, ie the parts that contain the footage and sound (so headers, packet sizes, and other important bytes are not changed). An overview of our fuzzer is shown below:



**Figure 5: Fuzzing Process Overview**

The process (ie python script) relies on iteratively mutating a single file. The python script will pass this mutated file through to FFmpeg, and use gdb to examine the return system error codes. Any special return error codes will indicate to the test harness that something abnormal happened, so the file that caused that return code will be preserved in another folder, along with its corresponding return code. Note that codes that are NOT considered to be abnormal are those that indicate FFmpeg has terminated correctly. These include:

- 0: Terminated successfully
- 1: Output file already exists, or outputted file is empty
- 69: Conversion failure (ie FFmpeg detects an issue with input, and exits cleanly)

The mutator will be chosen based on the file extension of the input file, or using a generic file mutator if depending on the value of the 'generic mutator flag'. There is more detail regarding this under 'documentation of our fuzzer'.

The process will test mainly for problems with the INPUT file. That is, our fuzzer would make FFmpeg convert to an output file of a fixed format (we've investigated several output formats - AVI, MP4, MP3, H263, MKV, and OGG).

### Documentation for our fuzzer

Our fuzzer can be run using the command:

```
./mutator <input file> <output file> <generic mutator flag> <extra args>
```

where:

- generic mutator flag indicates which one of the generic file mutators are to be used
- extra args provide additional information that certain generic file mutators may use

Generic mutator flags are detailed below:

Flag	Name	Description
gf	Generic Full	Every byte in the file will be randomised. This is essentially a dumb fuzzer.
gfp	Generic Full Probability	Every byte in the file has a change to be randomised based on a provided probability argument.
g20	Generic 20	Every byte in a random section in the first 20% of the file is randomised.
g20p	Generic 20 Probability	Every byte in a random section in the first 20% of the file has a change to be randomised based on a provided probability argument.
pad	Padding Mutator	Adds 200 zero bytes (0x00) to the end of a file.
z	Zero-Middle	Every byte in the middle zone (40-60%) of the file will be set to zeros (0x00).
zp	Zero-Middle Probability	Every byte in the middle zone (40-60%) of the file has a chance to be set to zeros (0x00) based on a provided probability argument.

Possible values for extra arguments are detailed below:

Value	Name	Description
Floating point value within <b>[0.0, 1.0]</b> (inclusive)	Probability	This probability argument is required by all probability variants of the generic mutators (ie gfp). The probability reveals the likelihood that a byte is mutated. So, a probability of 1.0 would mean that every byte is mutated (high chance of corruption) while a probability of 0.01 would mean very few bytes are mutated (high chance of maintaining a valid file).

### Mutators

If no generic mutator flag is provided, the fuzzer is capable of mutating some file formats. When mutating these file formats, our program reads headers to determine the location of data segments (video/audio/metadata). These data segments are altered in order to produce valid files with different data segment content.

Two methods for mutating bytes were tested. The first method is simply generating a random value for the byte (between 0x00-0xFF). The second method involved offset bytes by a particular (perhaps randomised) value. In this case a value of 0x37 could be offset by 1 and become 0x38. Interestingly, small offset values (-2,-1,1,2..etc) typically kept file formats in a valid state. We experimented with other mutation methods including burst mutation (mutating a random amount a bytes in a row by setting them to all 0xFF). This was designed to simulate burst noise corruption that is somewhat common when transmitting files over poor communication channels.

The file formats that we implemented were: FLV, WAV, MIDI

Unfortunately implementing many of the common, compression supported file formats proved to be difficult (including AVI, MP4) and simple data segment mutation would always result in corrupt file due to the complexity of these file formats. As a result the number of implemented file formats was limited.

Our generic mutators, on the other hand, pay no attention to the file format, and just mutates the bytes using different criteria regardless of file structure. The Generic Full (gf and gfp) mutators are simple but stupid mutators. They simply mutate all the bytes in a file which often produces corrupt (invalid) files. The Generic 20 (g20 and g20p) mutators are based off successful fuzzers (like the one mentioned in lectures) which simply mutate a portion of the file (eg first 20% of the file). We decided to mutate a random section of the first 20% of the file. This would undoubtedly mutate bytes in various header fields which is the primary aim for this mutator. The reason that only a random section (of varying sizes) is mutated is in order to preserve various parts of the header (file format identifier, total size of file, total size of data segment, total number of data segment packets...etc). This type of mutation led to lots of interesting results. We often found we could keep a file valid throughout a long test using the **gfp** variant with a low probability ( $\leq 0.10$ ). The mutated files would often play the same video and audio up to a certain point before they simply froze (audio and video streams froze).



Other weird observations included skips in the tracks. For example, an audio file would play up to 0:04 seconds and would then skip to 0:07 seconds. Other times, produced files would have their video completely changed (ie lots of colourful, static frames) and audio would usually turn into various tones or white noise.

### American Fuzzy Lop

After extensive research of all possible media file fuzzers, this was our best option (even though it fuzzes using ALL types of files). Other fuzzers we tried included FuzzBox, notSPIKEfile and Basic Fuzzing Framework. However, we failed to get them working with FFmpeg. AFL's test harness and sophistication in analysis was to our liking, so we gave it a few sample media files, and let it run briefly over a week's duration (locally). Our usage was simple:

```
./afl-fuzz -i testcases -o sus_files -- ./ffmpeg -i @@ result.avi
```

Below is a screenshot of one session of AFL:

<b>process timing</b>		<b>overall results</b>	
run time : 0 days, 21 hrs, 39 min, 14 sec		cycles done : 0	
last new path : 0 days, 0 hrs, 0 min, 50 sec		total paths : 1537	
last uniq crash : none seen yet		uniq crashes : 0	
last uniq hang : 0 days, 6 hrs, 49 min, 3 sec		uniq hangs : 0	
<b>cycle progress</b>		<b>map coverage</b>	
now processing : 0 (0.00%)		map density : 65.5k (99.99%)	
paths timed out : 0 (0.00%)		count coverage : 2.18 bits/tuple	
<b>stage progress</b>		<b>findings in depth</b>	
now trying : bitflip 1/1		favored paths : 4 (0.23%)	
stage execs : 8142/60.6k (13.44%)		new edges on : 540 (31.09%)	
total execs : 79.5k		total crashes : 0 (0 unique)	
exec speed : 0.85/sec (zzzz...)		total hangs : 0 (0 unique)	
<b>fuzzing strategy yields</b>		<b>path geometry</b>	
bit flips : 0/0, 0/0, 0/0		levels : 2	
byte flips : 0/0, 0/0, 0/0		pending : 1537	
arithmetics : 0/0, 0/0, 0/0		pend fav : 4	
known ints : 0/0, 0/0, 0/0		own finds : 1537	
dictionary : 0/0, 0/0, 0/0		imported : n/a	
havoc : 0/0, 0/0		variable : 1537	
trim : 0.00%/1875, n/a			
[cpu:100%]			

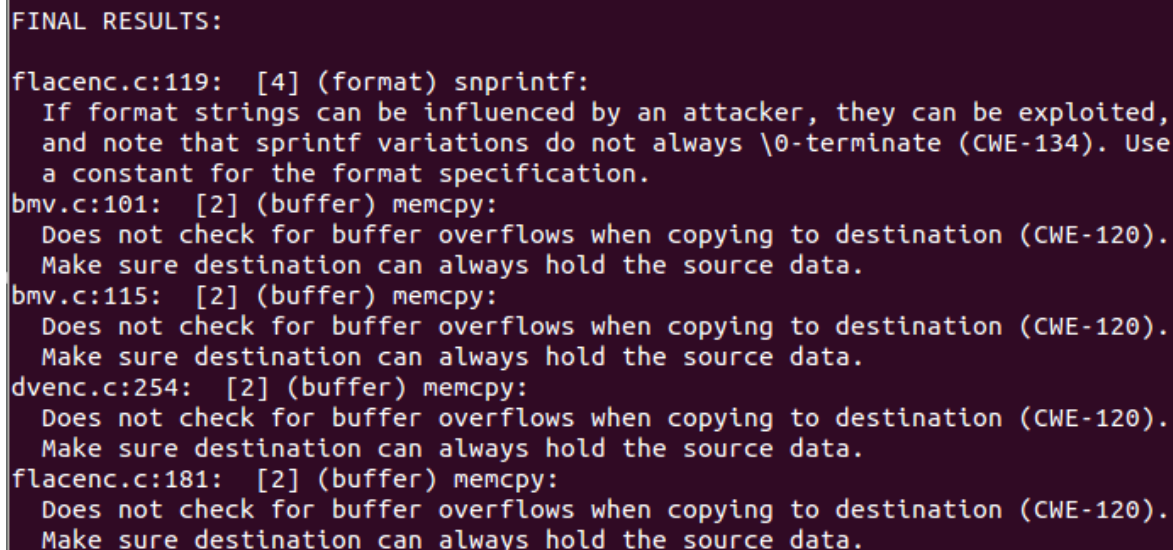
Due to the limitation of time and trying to covering other aspects of the exploitation process (developing our own fuzzer, or code reviewing), we didn't use AFL too long. However, if given the chance, we would let AFL fuzz longer (perhaps using some of our mutated files, as we only provided legitimate files to AFL).

# Code Review and Static Analysis

The source code for FFmpeg is long and complex (over 45 Mb worth of files, more than half of these were C files), and so it was difficult to read and understand it from scratch. Instead, we chose files from specific folders to review, namely **libavformat** and **libavcodec**, that contained the media file demuxers. Furthermore, we used tools to identify subsets of code that were recommended for further examination.

## FlawFinder

This was our initial tool for static code analysis. It is a tool for analysing C source code, available for Linux and Windows (we used Linux, since that was the platform on which we did all our testing). However, it is a naive static code analyser (powered by python scripts), as it only returned basic results that may contain security issues. In particular, the severity ranking returned depends on the library/function/coding pattern used, for example **snprintf** would be of severity level 4. This would be followed by a brief explanation of that line. A screenshot of FlawFinder in action with some of our chosen files is shown below:



```
FINAL RESULTS:

flacenc.c:119: [4] (format) snprintf:
  If format strings can be influenced by an attacker, they can be exploited,
  and note that sprintf variations do not always \0-terminate (CWE-134). Use
  a constant for the format specification.
bmv.c:101: [2] (buffer) memcpy:
  Does not check for buffer overflows when copying to destination (CWE-120).
  Make sure destination can always hold the source data.
bmv.c:115: [2] (buffer) memcpy:
  Does not check for buffer overflows when copying to destination (CWE-120).
  Make sure destination can always hold the source data.
dvenc.c:254: [2] (buffer) memcpy:
  Does not check for buffer overflows when copying to destination (CWE-120).
  Make sure destination can always hold the source data.
flacenc.c:181: [2] (buffer) memcpy:
  Does not check for buffer overflows when copying to destination (CWE-120).
  Make sure destination can always hold the source data.
```

Since the results are not very informative (as it essentially lists all the lines that contained those functions), and knowing that FlawFinder is a naive static analyser with a small set of functions that it will pass as a “potential threat”, the team decided not to proceed using the tool.

## Visual Code Grepper (VCG)

Our team used this tool (static C analyser) as it gives a list of results, along with a visual, that illustrates potential security issues. The results are prioritized and colour coded, according to the level of potential security threats (for example, red for high priority cases such as signed/unsigned comparison). Included in the results are also suspicious comments that may indicate incomplete code, or things to fix. Using these results, our team can get a good idea of which parts of the code to review. This tool is also slightly more sophisticated than FlawFinder, as it does a little more depth analysis (otherwise, it also lists a result based on the library/function/coding pattern used), hence we used this tool more than FlawFinder.

For our project, our input for VCG was simple - files that were core to encoding and decoding inputs into outputs. In particular, we examined the files from two folders, **libavformat** and **libavcodec**. Only \*.c files that were commonly changed and rarely changed were chosen as candidate files to be scanned using VCG.

Below you will find write ups for a subset of the files that we analysed this way. Each potential issue was inspected manually. As we inspected more and more files, we started to ignore 'medium' risk warnings as their usage was quite standard (eg most memcpy's seemed to be pretty similar in terms of arguments: destination, src, and some fixed number that depend on the size of a particular field in the media format).

## Code Analysis Write Up (only a handful of examples are shown)

**Source file:** libavformat/flvdec.c

**Name:** FLV demuxer

### **MEDIUM: goto**

Use of 'goto' function. The goto function can result in unstructured code which is difficult to maintain and can result in failures to initialise or de-allocate memory.

Line: 347 - Filename: O:\Dropbox\Security Workshop Project\possible\_files\flvdec.c  
goto finish;

**Response:** These 'medium' risk warning appear all the time. In this particular source file, goto is used to escape from while loops and inside conditional if statements to various labels. This is bad practice but there were no issues in this case. The labels that were jumped to include a **leave** label which skips a series of bytes in the stream (via **avio\_skip**) before returning. Another interesting label was the **finish** label which frees some memory (via **av\_freep**) before returning. I followed the variables that would eventually be free'd in the **parse\_keyframes\_index** function:

```
int64_t *times          = NULL;
int64_t *filepositions = NULL;
```

To my surprise, it was possible for times and file positions to be **NULL** upon reaching the **finish** label:

```
finish:
    av_freep(&times);
    av_freep(&filepositions);
    avio_seek(ioc, initial_pos, SEEK_SET);
    return ret;
```

However, after looking at the documentation for the **av\_freep** function, it becomes clear that it acts as a NOP (no operation) if a null pointer is passed as the argument. A false alarm.

#### MEDIUM: memcpy

Function appears in Microsoft's banned function list. Can facilitate buffer overflow conditions and other memory mis-management situations.

Line: 1066 - Filename: O:\Dropbox\Security Workshop Project\possible\_files\flvdec.c  
memcpy(side, flv->new\_extradata[stream\_type],

**Response: memcpy** is another of those results listed as "MEDIUM". In this particular case, the pointer **side** is being initialised with values from memory pointed to by **flv**, which is of type **FLVContext**. The code afterwards frees some of the data inside **FLVContext**, implying that the data is simply being moved to the **side** pointer for some conversion purpose. **memcpy** is infamous for buffer-overflow exploits, and so we conducted a check involving all three of its arguments.

The first argument is **side** itself - since this code is executed ONLY when **side** is not null, it should be fine. The second argument is **flv->new\_extradata[stream\_type]**. The **FLVContext flv** is initialised using an argument, which after tracing, that argument can never be null. The int **stream\_type** is also initialised (through multiple if statements). The third argument is **flv->new\_extradata[stream\_type]**. We traced through to the definition of the field definition, and finally determined that the value held in this variable is never bigger than the second argument previously described. Hence this memcpy instance is legit.

**Source file:** libavcodec/mjpeg2jpeg\_bsf.c

**Name:** MJPEG/AVI1 to JPEG/JFIF bitstream format filter

#### CRITICAL: Unsafe Use of memcpy Allows Buffer Overflow

The size limit is larger than the destination buffer, while the source is a char\* and so, could allow a buffer overflow to take place.

Line: 59 - Filename: D:\Desktop\Ffmpeg-master\libavcodec\mjpeg2jpeg\_bsf.c  
memcpy(buf, src, size);

Our global scan revealed a few critical errors! In this case, an unsafe usage of memcpy that could lead to a buffer overflow.

Upon reviewing the source code I see the function in question:

```
static uint8_t *append(uint8_t *buf, const uint8_t *src, int size)
{
    memcpy(buf, src, size);
    return buf + size;
}
```

This function copies **size** bytes from **src** to **buf** and then increments the buf pointer by that number of bytes (as buf is pointer to a **uint8\_t**).

In other areas of the code, we see:

```
static uint8_t *append_dht_segment(uint8_t *buf)
{
    buf = append(buf, dht_segment_head, sizeof(dht_segment_head));
    buf = append(buf, avpriv_mjpeg_bits_dc_luminance + 1, 16);
    buf = append(buf, dht_segment_frag, sizeof(dht_segment_frag));
    buf = append(buf, avpriv_mjpeg_val_dc, 12);
    *(buf++) = 0x10;
    buf = append(buf, avpriv_mjpeg_bits_ac_luminance + 1, 16);
    buf = append(buf, avpriv_mjpeg_val_ac_luminance, 162);
    *(buf++) = 0x11;
    buf = append(buf, avpriv_mjpeg_bits_ac_chrominance + 1, 16);
    buf = append(buf, avpriv_mjpeg_val_ac_chrominance, 162);
    return buf;
}
```

This code uses our append function to append the dht\_segment bit by bit. This was most likely done so so that the code is more pleasant to look at (as the headers and fragments are clearly defined). However, we must check that we do not call append with **size** values that add up to be more than the total size of **buf**.

So how much memory was reserved for **buf**?

```
output_size = buf_size - input_skip + sizeof(jpeg_header) + dht_segment_size;
```

The buffer size is provided as an argument in the **mjpeg2jpeg\_filter** function but will be at least 12 bytes. The size of **input\_skip** depends on the contents of a buffer that is provided as an argument too. The **jpeg\_header** is 20 bytes large and the **dht\_segment\_size** is defined to be 420.

So we must check the size values sent to the **append** function.

We see:

```
out = append(out, jpeg_header, sizeof(jpeg_header));
out = append_dht_segment(out);
out = append(out, buf + input_skip, buf_size - input_skip);
```

Looking at **append\_dht\_segment** we tally up the total number of bytes to get:

$5+16+29+12+1+16+162+1+16+162 = 420$  (🌿). This is the same size as the **dht\_segment\_size** so there is no misuse here.

The other two calls to **append** (1 before and 1 after the call to **append\_dht\_segment**) pass **sizeof(jpeg\_header)** and **(buf\_size - input\_skip)** number of bytes to the **append** function respectively. This equals the total amount of bytes that was originally reserved (**output\_size**) and so we conclude that there is no misuse of **memcpy** in this source file.

**Source file:** compat/avisynth/avisynth\_c.h

**Name:** Avisynth C Interface Version 0.20

**HIGH: Potentially Unsafe Code - LoadLibrary**

Line: 792 - D:\Desktop\Ffmpeg-master\compat\avisynth\avisynth\_c.h

The function searches several paths for a library if called with a filename, but no path. This can allow trojan DLLs to be deployed, regardless of the presence of the correct DLL. Manually check the code to ensure that the full path is specified.

```
library->handle = LoadLibrary("avisynth");
```

The issue here is a possible unsafe usage of the **LoadLibrary** function which could lead to a malicious DLL being used instead of the intended DLL.

The security issues are detailed here:

[https://msdn.microsoft.com/en-us/library/windows/desktop/ms684175\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms684175(v=vs.85).aspx)

In this case the **SearchPath** function was not used to retrieve a DLL address which is good. Furthermore, no assumptions are made about the operating system based on the **LoadLibrary** call. Therefore there are security issues here.

**LoadLibrary** is used many times in different source code files but upon checking, they are all used in the same (safe) way.

**Source file:** compat/strtod.c

**Name:** C99-compatible strtod() implementation

**SUSPICIOUS COMMENT: Comment Indicates Potentially Unfinished Code -**

Line: 82 - D:\Desktop\Ffmpeg-master\compat\strtod.c

FIXME this doesn't handle exponents, non-integers (float/double)

Suspicious comments are another branch of issues that VCG identifies. Essentially, it looks for comments that include various words like: **FIX**, **TODO**, **DOESN'T WORK**, **PROBLEM**.

The function in this source file attempts to convert a string to a double (**strtod**). Functionality is same as: <http://www.cplusplus.com/reference/cstdlib/strtod/>

In this case, the suspicious comment and relevant code is:

```
else if (!av_strncasecmp(nptr, "0x", 2) ||
        !av_strncasecmp(nptr, "-0x", 3) ||
        !av_strncasecmp(nptr, "+0x", 3)) {

    /* FIXME this doesn't handle exponents, non-integers(float/double)
     * and numbers too large for long long */
    res = strtoll(nptr, &end, 16);
}
```

This section of the code essentially checks to see if the string represents a value that begins with **0x** or **-0x** or **+0x** which would indicate the value is represented in hex. As it turns out, **strtoll** (<http://www.cplusplus.com/reference/cstdlib/strtoll/>) will not handle non-integers (float/doubles) and large numbers will cause the result (**res**) to represent the wrong number. Therefore, there are issues with this function **avpriv\_strtod** and we must ensure it is never called with a string that contains non-integers or very large numbers.

I used **grep** to simply search all source files to see where **avpriv\_strtod** was used. This function is used in two other locations (**libavutil/internal.h** (line 241) and **configure** (line 4584). In the first case we are simply leaving a comment in the generated object file so this can be ignored. The second instance actually defines **strtod** to use the **avpriv\_strtod** code. This is also apparent when looking at **strtod.c** as we **#undef strtod**.

**strtod** is essentially used in 11 different locations within various functions. I checked all function calls and almost all were simply converting numeric or hexadecimal representations to a double value. One case in **libavfilter/vf\_freio.c** seemed to convert a representation of a **VALUE\_MAX** (not an actual number) or **INFINITY** to a double which was also supported by the **avpriv\_strtod** function. Thus, I can conclude that there are no issues in the codebase. Although future calls to **avpriv\_strtod** could potentially be problematic and no documentation exists at the top of the function which details this precondition.

Suspicious comment issues were quite interesting to explore as they basically identified issues that others have found. However, these issues were often minor as they would have otherwise been patched by the developers who found them. In most cases they served to warn others to not call particular functions with certain arguments. That is, they detailed preconditions to various functions through comments.

**Source file:** libavcodec/dsicinvideo.c

**Name:** Delphine Software International CIN video decoder

**CRITICAL: Unsafe Use of memcpy Allows Buffer Overflow**

The size limit is larger than the destination buffer, while the source is a char\* and so, could allow a buffer overflow to take place.  
Line: 103 - Filename: D:\Desktop\FFmpeg-master\libavcodec\dsicinvideo.c  
`memcpy(huff_code_table, src, 15);`

In this case, VCG has flagged this usage of memcpy as a potentially critical issue because a magic number is used as an argument to memcpy (which tells memcpy how many bytes to copy). Now this is only an issue if we are copying in more bytes than the size of **huff\_code\_table**.

We look at the code in the **cin\_decode\_huffman** function and see:

```
unsigned char huff_code_table[15];
```

(snip)

```
memcpy(huff_code_table, src, 15);
```

As the size of the **huff\_code\_table** array is 15 bytes, there is no issue. However, this magic number should be avoided and the third argument to memcpy should be **sizeof(huff\_code\_table)**.

**Source file:** libavformat/wavdec.c

**Name:** WAV decoder

**STANDARD: Potentially Unsafe Code - strlen**

Line: 161 - C:\Users\Ian\Desktop\15s2\COMP9447\FFmpeg-master\libavformat\wavdec.c  
Function appears in Microsoft's banned function list. For critical applications, particularly applications accepting anonymous Internet connections or unverified input data, strlen and similar functions can become victims of integer overflow or 'wraparound' errors.  
`if (strlen(temp))`

As shown by VCG, the potential line **strlen(temp)** may be an issue, where temp is an array of 257 characters. **strlen** is infamous for when the argument is not null terminated by the user. However, previous to this line, we can be sure temp is null terminated:

```
temp[length] = 0
```

where length is an int passed as an argument. At this point, we are also sure that length is NOT larger than 257, because of the line:

```
av_assert0(length <= sizeof(temp));
```



The problem would therefore be whether **length** is a negative number. From the function where this line is located, if length was negative, then we would be accessing the memory located BEFORE the temp array (in which case, would be the function base pointer), as the array is the first local variable of the function.

However, from the same function, we can also be sure that the line **temp[length]** will NOT be executed if length is negative, because we have an if statement before this that reads in a number of bytes from the AVFormatContext into temp:

```
if ((ret = avio_read(s->pb, temp, length)) < 0) return ret;
```

By closely examining the function **avio\_read**, it does an internal check that the bytes to read (ie third argument, length) MUST be positive, otherwise returns an error value which is negative, hence executing the **return ret** after the IF statement.

This would mean **length** is a positive number between 1 and 257 at the **strlen** instruction, AND temp is null terminated. Hence strlen is being used properly, and (unfortunately for us) malicious input will be ineffective.

## Other Remarks

### signed/unsigned comparisons

We ignored the signed/unsigned comparison potential errors as VCG did not detect the issues correctly in most (if not all) cases. For example, the following code would be flagged by VCG:

```
for (i = 0; i < arraylen && avio_tell(ioc) < max_pos - 1; i++) { ... }
```

where **i** and **arraylen** are unsigned, **avio\_tell(ioc)** and **max\_pos** are signed.

Clearly, there is no issue here.

### goto, memcpy, strlen etc

These functions can potentially cause issues but are fine when used properly. FFmpeg source code files use goto a lot (too much for our liking) and even though the usage appears to be reasonable we would rather have them use if statements or other conditional checks, especially in some cases where a goto jumps down 400 lines!

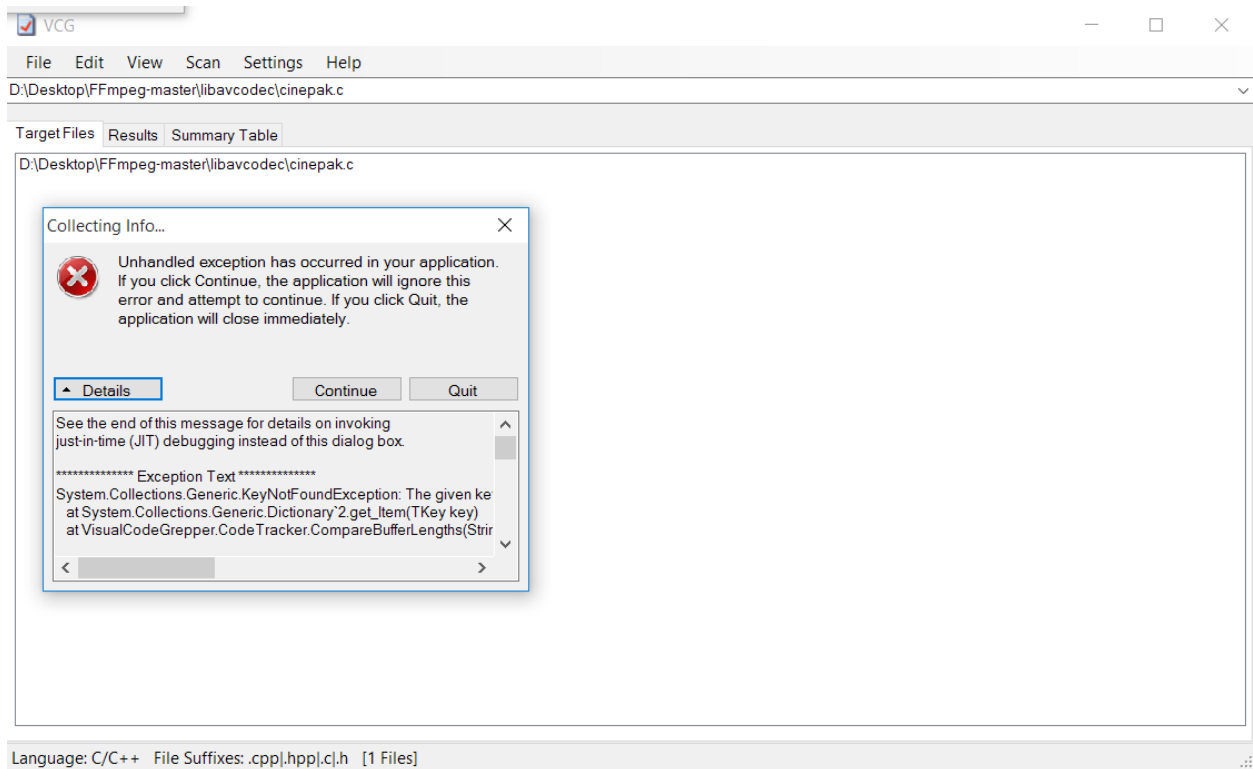
Here are a few more potential issues VCG found in the **libavformat** and **libavcodec** folders.

Priority	Severity	Title	Description
<input type="checkbox"/> 3	Medium	memcpy	Function appears in Microsoft's banned function list. Can facilitat...
<input type="checkbox"/> 3	Medium	goto	Use of 'goto' function. The goto function can result in unstructure...
<input type="checkbox"/> 3	Medium	goto	Use of 'goto' function. The goto function can result in unstructure...
<input type="checkbox"/> 3	Medium	memcpy	Function appears in Microsoft's banned function list. Can facilitat...
<input type="checkbox"/> 3	Medium	goto	Use of 'goto' function. The goto function can result in unstructure...
<input type="checkbox"/> 3	Medium	goto	Use of 'goto' function. The goto function can result in unstructure...
<input type="checkbox"/> 3	Medium	goto	Use of 'goto' function. The goto function can result in unstructure...
<input type="checkbox"/> 3	Medium	goto	Use of 'goto' function. The goto function can result in unstructure...
<input type="checkbox"/> 3	Medium	goto	Use of 'goto' function. The goto function can result in unstructure...
<input type="checkbox"/> 3	Medium	goto	Use of 'goto' function. The goto function can result in unstructure...
<input type="checkbox"/> 3	Medium	memcpy	Function appears in Microsoft's banned function list. Can facilitat...
<input type="checkbox"/> 3	Medium	memcpy	Function appears in Microsoft's banned function list. Can facilitat...
<input type="checkbox"/> 3	Medium	memcpy	Function appears in Microsoft's banned function list. Can facilitat...
<input type="checkbox"/> 3	Medium	memcpy	Function appears in Microsoft's banned function list. Can facilitat...
<input type="checkbox"/> 3	Medium	memcpy	Function appears in Microsoft's banned function list. Can facilitat...
<input type="checkbox"/> 3	Medium	memcpy	Function appears in Microsoft's banned function list. Can facilitat...
<input type="checkbox"/> 3	Medium	memcpy	Function appears in Microsoft's banned function list. Can facilitat...
<input type="checkbox"/> 3	Medium	memcpy	Function appears in Microsoft's banned function list. Can facilitat...
<input type="checkbox"/> 3	Medium	goto	Use of 'goto' function. The goto function can result in unstructure...
<input type="checkbox"/> 3	Medium	goto	Use of 'goto' function. The goto function can result in unstructure...
<input type="checkbox"/> 3	Medium	goto	Use of 'goto' function. The goto function can result in unstructure...
<input type="checkbox"/> 3	Medium	goto	Use of 'goto' function. The goto function can result in unstructure...

## VCG crashes

Ironically, we managed to crash the VCG file scanning thread.

During a global scan (where we scanned every file to review the results as group) we discovered one particular file (**libavcodec/cinepak.c**) would cause the VCG file scanning thread to freeze and terminate which would stop the current scan but would not terminate the program. We reported this bug as an issue in VCG's GitHub repo.



# Conclusion

## Overview of results

In the end our team wrote our own fuzzer with various mutators as well as a test harness to automate the fuzzing process. We did local fuzzing using our laptops and computers and some cloud fuzzing which ran 24/7 for many weeks.

We also experimented with many fuzzing suites and utilities which could perform 'smarter' fuzzing tests. Furthermore, we performed a lot of source code review where we tracked user input and function parameters to determine if demuxers incorrectly analysed file formats. Source code analysis tools were also used to find suspicious code which we also manually analysed.

Unfortunately, even though we performed a lot of fuzzing and source code analysis, we were unable to cause FFmpeg to abnormally crash and were unable to find any other software bugs. This meant that we could not write any exploits.

## What we learnt overall

Although we failed to find any software bugs, this project allowed us to develop skills required to analysis any test any piece of software. We learnt about the fuzzing process and how to write fuzzers (mutators + test harness). File format structure analysis through RIFFPad and HxD was also a valuable skill that was learnt - it allowed us to understand how files are represented in binary and how they are packed. This will certainly be useful in the future (especially to reverse engineer file formats we know nothing about). We also learnt about the source code reviewing process including common mistakes (integer promotions, off by one errors, buffer overflows, memcpy, memset, strlen, etc). We also learnt how to use many third party fuzzing/source code analysis tools that we will definitely use in the future (on our code and the code of others!). Our team also came to appreciate the usefulness of FFmpeg as a media converter application, this thing can literally support every format (including ones from 20+ years ago). We found that fuzzing commonly used multimedia file formats was limited in nature as some of these file formats were difficult to implement but little chance of exposing a bug.

Overall, this project improved drastically the skillset of our team members.

### Things we could have done better

There are various things we could have done better in this project. Setting up strict task management (perhaps using project management tools) would have allowed us to more clearly outline what tasks each person should focus on. This was an issue as some team members actually analysed the same source code files (derp!).

Obviously, fuzzing for a few weeks is not as beneficial as fuzzing for longer durations of time. Ideally, one would set up their fuzzer as soon as possible and get it function so that they could fuzz for a period of 8+ weeks. Also we could have put more time into developing smarter fuzzing mutators. We also set up American Fuzzy Lop (AFL) too late which limited our fuzzing time with it.

We could have used more sophisticated static analysis tools. However, there is only so much an automated tool can find, thus the necessity for even more manual code review than what we have.

Ultimately, we could have selected an easier and buggier piece of software! FFmpeg is actually fairly secure with the large number of developers who contribute to the open source project. Furthermore, third parties like Google who conduct massive scale fuzzing operations are much more likely to find bugs and flaws. Also, most issues in FFmpeg simply cause a crash and are often not exploitable in any way due to the nature of the program.

### Advice for those who tackle this in the future (future work)

- Pick a good topic! You are unlikely to find bugs in huge open source programs, especially if they are being fuzzed by others (like Google).
- Software with source code is great for this assessment. Otherwise you'll need to reverse engineer it from the binary, which is a PITA for large programs.
- American Fuzzy Lop is the best media file fuzzer. Try to get it working ASAP, and run it as long as possible. In general, try to set up fuzzers ASAP and start running them early.
- Cloud computing for fuzzing is much, much better than running a fuzzing process on your computer or laptop from time to time. It takes some effort to set up but can then be left to run 24/7.
- Expect to spend a LONG time on source code reviewing (especially if the software is relatively large). You will constantly have to trace arguments and definitions which means you end up understanding how many different source code files work. At the end of all of the analysis you should pretty much understand exactly how your program works.
- Don't solely rely on static code analysis tools, they can be inaccurate.
- Try to spend a third of the project time on fuzzing a two thirds on source code analysis. We decided to do a 50/50 split and realised we could have benefitted more by doing more source code analysis (at least in the case of FFmpeg).