How does Zoom Store Recordings?

Reverse Engineering C++ and Custom File Formats

Leonardo Galli

flagbot (CTF@VIS)

October 8, 2020

About Me

- Finishing my Bachelor of Computer Science at ETH
- Member of flagbot since over two years
- President of flagbot since over a year

Leonardo Galli
leonardo.galli@vis.ethz.ch

About flagbot

- VIS committee and ETH's Capture the Flag team
  - CTFs are team-based cybersecurity competitions, often involving real-world attacks
- Ranked 1st place in Switzerland in 2019 and 2020[1]
- Most recent: 5th place in 0CTF (Tencent) Finals
  - Teamed up with polygl0ts (EPFL), the cr0wn[2], excusemewtf? and secret club
- Playing CTFs on weekends
- Weekly meetings on Monday at 19:00 over Zoom, open to anyone
  - Discussion of challenges and lectures aimed at beginners (recordings available on `flagbot.ch/material`)

Contact: ctf@vis.ethz.ch
More Information: `flagbot.ch`

[1] According to `ctftime.org`
[2] 1st-placed UK team

Introduction

Premise

- At the beginning of the year, needed to shift from in-person meetings to online
- Wanted to record lectures for uploading to our website
- Audio mixing was a big problem
- Zoom allows you to export every person as a separate audio file
- Unfortunately, they remove any periods of silence longer than a few seconds
  - Nightmare to try and synchronize

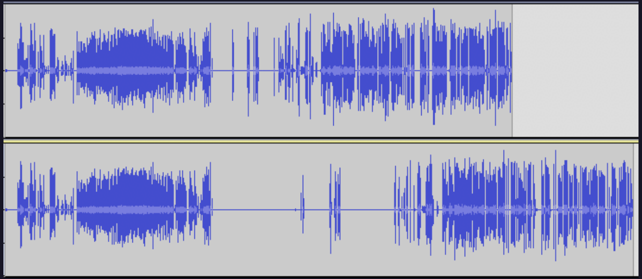


Figure: Top: audio as output by Zoom, bottom: audio as extracted by me.

Idea

- Zoom stores recordings in temporary files
  - These are called `double_click_to_convert_0{1,2,3}.zoom`
- **Goal:** Figure out how recordings are stored in those files
  - **Nice to have:** Extract higher-quality video recordings
- **Approach:** Reverse-engineer Zoom's transcoder app and accompanying libraries
- **Side effects:** Learn more about Zoom's software architecture and the H.264 standard

Table of Contents

Introduction
Reversing Tactics

Static vs. Dynamic Analysis

- Two major approaches to reversing
- Usually want to use a combination of both
- Not just for reversing compiled applications, but also most other code
    - Can be applied to JavaScript, Python, etc.

Static Analysis

- Look at application through a decompiler / disassembler
  - **disassembler:** Tool for analyzing the machine code of an application
  - **decompiler:** Tool for converting disassembly to high-level source code
  - Popular free tools: Ghidra, radare2 (+ Cutter), IDA freeware
  - Similar tools exist for non-compiled languages
- Figure out types, function signatures, purpose and more
- Can quickly get complicated
- Analyze supporting files with other tools and try to figure out their purpose
  - `binwalk` to extract possible files in a larger collection
  - Often, custom file formats are identifiable in just a hex viewer

Dynamic Analysis

- Try to gain insight into the application by analyzing it **at runtime**
- Attach a debugger and step through functions, analyzing memory contents
  - Often, static tools can also do dynamic analysis
- Inject code and hook functions
  - Can be easier than scripting a debugger
- Symbolically execute parts of the application (or even the whole thing)[3]
- Often underused, even though it can be a lot faster
  - Especially helpful with C++ VTables

[3] `angr.io` is a popular tool for this.

Introduction
C++ Instance Methods and VTables

Example Classes

```cpp
class Animal {
    void eat();
    virtual bool tryPet();
};

void Animal::eat()
{
    // Something useful
}

bool Animal::tryPet()
{
    // not all animals can be pet
    return false;
}
```

```cpp
#include "animal.h"

class Dog : public Animal {
    bool tryPet() override;
};

bool Dog::tryPet()
{
    return true;
}
```

Instance Methods in C++

```
// Actual function as emitted by the compiler
void Animal::eat(Animal* this)
{


}

Dog* dog = new Dog();
Animal::eat((Animal*)dog); // compiled from dog->eat();
```

- Compiler just adds the `this` parameter
- Function calls work normally (just like they would in C)
- Not a big impact on reversing

Virtual Instance Methods in C++

```
void petOrError(Animal* animal)
{
    if (!animal->tryPet()) {
        printf("Failed to pet animal!");
    }
}
```

- Above code causes a problem for a naive compiler
- How to know which implementation of `tryPet` to call?
- Use **virtual function tables** (vtables) for dynamic dispatch

vtables

```
struct Animal {
    Animal_vtable* vtable;
    // other properties
};
// Shared among all instances of same type
struct Animal_vtable {
    // Dog::tryPet() for Dog instances, Animal::tryPet() for Animal ones
    bool (*tryPet)(Animal* this);
};

Animal* dog = (Animal*)new Dog();
dog->vtable->tryPet(dog); // compiled from dog->tryPet();
```

- Store information about location of virtual functions on object itself
- Much harder to reverse engineer

Reverse Engineering Process

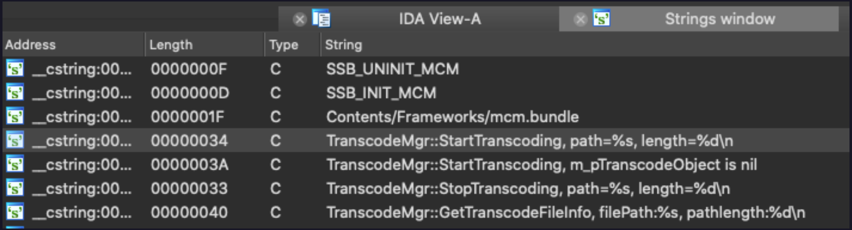
Overview

- Combined both dynamic and static analysis
- Used decompiler and hex viewer most frequently
- Reconstructed many class hierarchies in the decompiler
  - Used debugger to figure out relevant classes and functions
- Verified findings with python scripts, debugger and binary hooking
- Sometimes, educated guesses were enough

Reverse Engineering Process
Initial Reconnaissance

Finding a Starting Point

- Suspiciously small main application (`Transcode.app`)
- Unable to find filenames, but found interesting logging statements at least
- Static analysis revealed that actual work is quickly delegated to `mcm.bundle`
  - mcm library is very opaque: almost no exports, imports or symbols[4]



IDA View-A | Strings window

Address	Length	Type	String
__cstring:00...	0000000F	C	SSB_UNINIT_MCM
__cstring:00...	0000000D	C	SSB_INIT_MCM
__cstring:00...	0000001F	C	Contents/Frameworks/mcm.bundle
__cstring:00...	00000034	C	TranscodeMgr::StartTranscoding, path=%s, length=%d\n
__cstring:00...	0000003A	C	TranscodeMgr::StartTranscoding, m_pTranscodeObject is nil
__cstring:00...	00000033	C	TranscodeMgr::StopTranscoding, path=%s, length=%d\n
__cstring:00...	00000040	C	TranscodeMgr::GetTranscodeFileInfo, filePath:%s, pathlength:%d\n

Figure: No sight of the filenames, but interesting strings nonetheless.

[4] think (function) names

Dynamic Analysis to the Rescue

- Start `Transcode.app` under a debugger and pause in the middle
  - Hopefully, call stack will hint to where we want to start investigating
- At first, call stack looked useless, but after switching to different threads I spotted an interesting call stack
  - Contains functions referencing files as well as video decoding

Address	Module	Function
00007FFF73D1D882	libsystem_kernel...	__psynch_cvwait+A
0000000115817E0A	zlt	0000000115817E0A
000000011581830B	zlt	000000011581830B
00000001159489C2	zlt	_DestroyGltInterface+A36B6
0000000115948645	zlt	_DestroyGltInterface+A3339
0000000110EDA30D	zmb	zMediaBox::zmbCVideoDec::GetOutputBs(zmbVideoSample &,int,int)+75
0000000110EE405D	zmb	zMediaBox::zmbCMultiChannelVideoConvertOpt2::DeliveAs(zmbVideoSample &,zMediaBox::zmbCMulti...
0000000110EE22CE	zmb	zMediaBox::zmbCMultiChannelVideoConvertOpt2::ProcessOutput(zmbVideoSample &)+674
0000000110ED0158	zmb	zMediaBox::zmbCVideoTranscodeMc::ReceiveSample(zMediaBox::sample_if *)+4B8
0000000110ECF6B5	zmb	zMediaBox::zmbCVideoTranscodeMc::NeedForSample(zmbChannelInfo &)+EB
0000000110EC201D	zmb	zMediaBox::zmbCNodePortBase::Notify(int,void *)+31
0000000110ECD478	zmb	zMediaBox::zmbCFileSource2Mc::NeedForSample(zMediaBox::trc_rt_info_ext_t &,zMediaBox::trc_rt_info_...
0000000110ECC15A	zmb	zMediaBox::zmbCFileSource2Mc::run(void)+714
0000000110EED09F	zmb	zMediaBox::runable_t::routine(void *)+15
00007FFF73DDE103	libsystem_pthre...	_pthread_start+8E
00007FFF73DD9B86	libsystem_pthre...	_thread_start+A

Figure: Interesting call stack of one thread.

Opening the zmb Framework

- Although it looked promising under the debugger, there could be complications
- Most functions fully retained their names alongside argument types[5]
- Heavy use of C++ throughout the binary
- Looked for the filenames in the strings of the binary and started reversing from there

IDA View-A | Strings window

Address	Length	Type	String
__cstring:00...	0000001B	C	double_click_to_convert_01
__cstring:00...	0000001B	C	double_click_to_convert_02
__cstring:00...	0000001B	C	double_click_to_convert_03
__cstring:00...	0000000F	C	audio_only.m4a
__cstring:00...	0000000C	C	audio_only_
__cstring:00...	00000009	C	chat.srt

Figure: Finally, we found the filenames.

[5] The types themselves were lost, though.

Reverse Engineering Process

Example: Reversing the File Header

Initial Decompiled Function

```
__int64 __fastcall zMediaBox::io_read_mgr_t::io_read_mgr_t(zMediaBox::io_read_mgr_t *this, int *a2, const char *a3)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  *((_DWORD *)this + 16) = 0;
  *((_DWORD *)this + 16) = ((unsigned __int16)zMediaBox::thread_tool_t::thread_mutex_create(
                                                (pthread_mutex_t *)this,
                                                (_opaque_pthread_mutex_t *)a2) & 0xFFFCu) < 0x64;
  *((_DWORD *)this + 24) = 0;
  *((_QWORD *)this + 11) = 0LL;
  *((_QWORD *)this + 10) = 0LL;
  *((_QWORD *)this + 9) = 0LL;
  *(_QWORD *)((char *)this + 100) = 0xFFFFFFFF00000000LL;
  v4 = operator new(0x30uLL);
  result = zMediaBox::io64_read_t::io64_read_t((zMediaBox::io64_read_t *)v4, a3);
  *((_QWORD *)this + 9) = v4;
  if ( !*(_DWORD *)(v4 + 40) || !*(_QWORD *)(v4 + 8) )
  {
    *a2 = 47513717;
    return result;
  }
  v21 = 0x400000DC601LL;
  v28 = 0LL;
  v27 = 0LL;
  v26 = 0LL;
  v25 = 0LL;
  v24 = 0LL;
  v23 = 0LL;
  v22 = 0LL;
  v17 = 0x84AD52E22C05F158LL;
  v20 = 0LL;
  v19 = 0LL;
  v18 = 0LL;
  v6 = zMediaBox::io64_read_t::read((zMediaBox::io64_read_t *)v4, (unsigned __int8 *)&v17, 0x60uLL);
  *a2 = v6;
  result = (unsigned __int16)v6 & 0xFFFC;
  if ( (unsigned int)result > 0x63 )
    return result;
  v8 = v17 == 0x84AD52E22C05F158LL;
  v29 = (__int64)this + 100;
  zMediaBox::version_mgr_t::set(
    (char *)this + 100,
    (unsigned int)v21,
    v7,
    (unsigned int)v17 ^ 0x2C05F158 | HIDWORD(v17) ^ 0x84AD52E2);
```

Decompiled Function after Cleanup and Annotation

```c
__int64 __fastcall zMediaBox::io_read_mgr_t::io_read_mgr_t(io_read_mgr *this, _opaque_pthread_mutex_t *a2, const char *filename)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  *(_DWORD *)&this->is_good = 0;
  *(_DWORD *)&this->is_good = ((unsigned __int16)zMediaBox::thread_tool_t::thread_mutex_create(&this->mutex, a2) & 0xFFFCu) < 0x64;
  *(_DWORD *)&this->version_good = 0;
  this->size_of_file = 0LL;
  this->start_of_data_offset = 0LL;
  this->io64_com = 0LL;
  this->version_info = (version_info)0xFFFFFFFF00000000LL;
  io_read = (zMediaBox::io64_read_t *)operator new(48LL);
  result = (__int64)zMediaBox::io64_read_t::io64_read_t(io_read, filename);
  this->io64_com = (zMediaBox::io64_com_t *)io_read;
  if ( !io_read->file_state || !io_read->file_fd )
  {
    LODWORD(a2->__sig) = 0x2D50075;
    return result;
  }
  *(_QWORD *)&header_data[32] = 0x400000DC601LL;// version
  *(_QWORD *)&header_data[88] = 0LL;
  *(_QWORD *)&header_data[80] = 0LL;
  *(_QWORD *)&header_data[72] = 0LL;
  *(_QWORD *)&header_data[64] = 0LL;
  *(_QWORD *)&header_data[56] = 0LL;
  *(_QWORD *)&header_data[48] = 0LL;
  *(_QWORD *)&header_data[40] = 0LL;
  *(_QWORD *)header_data = 0x84AD52E22C05F158LL;
  *(_QWORD *)&header_data[24] = 0LL;
  *(_QWORD *)&header_data[16] = 0LL;
  *(_QWORD *)&header_data[8] = 0LL;
  v6 = zMediaBox::io64_read_t::read(io_read, (unsigned __int8 *)header_data, 96uLL);// read header
  LODWORD(a2->__sig) = v6;
  result = (unsigned __int16)v6 & 0xFFFC;
  if ( (unsigned int)result > 0x63 )
    return result;
  v7 = *(_QWORD *)header_data == 0x84AD52E22C05F158LL;
  a1 = &this->version_info;
  zMediaBox::version_mgr_t::set(&this->version_info, *(unsigned int *)&header_data[32]);
  v8 = this->version_info.number;
  if ( v8 < 0 )
  {
    *(_DWORD *)&this->version_good = 0;
  }
  else
```

Relevant Parts of Function

```
__int64 zMediaBox::io_read_mgr_t::io_read_mgr_t(io_read_mgr *this, ...)
{
    // Initializing a bunch of variables related to reading
    char header_data[96];
    memset(header_data, 0, 96);
    *(_QWORD *)&header_data[32] = 0x400000DC601LL;// version
    *(_QWORD *)header_data = 0x84AD52E22C05F158LL;// packet delimiters
    // read header
    zMediaBox::io64_read_t::read(io_read, header_data, 96uLL);
    zMediaBox::version_mgr_t::set(&this->version_info,
                                  *(unsigned int *)&header_data[32]);
    // Make sure version info is ok!
    if (*(_QWORD *)header_data != 0x84AD52E22C05F158LL) return -1;
    this->data_start = *(int *)&header_data[36];
    return zMediaBox::io64_com_t::seek(this->io64, this->data_start, 0);
}
```

Reverse Engineering Process

Further Investigation

File Format

- Starting from the previous function, slowly restored class hierarchies and found locations where file contents are used
- Quickly located functions relevant to parsing the files
  - Only used for very basic parsing: splits file into packets
  - General pattern would have also been easily spotted with a hex viewer
- By reversing even more of the class hierarchies, certain fields of the packets became apparent
- Allowed me to differentiate between different types of packets and dump their data contents

Audio Format

- Finally able to dump audio information
  - What format is used to store the audio, though?
- Concatenated all audio data and loaded it into Audacity (8-bit PCM, Stereo): Initial Result
  - Left side is actually somewhat understandable
- Inspecting `Transcode.app`'s output reveals mono audio with a 32 kHz sample rate
  - Sounds worse than before!
  - However, the length is exactly double that of the transcoding result
- Loading it again with 16-bit PCM, Mono, yields: Correct Output

Matching Audio to Names

- Every audio sample has an attached *name identifier*: a simple integer
- Spent a lot of time reversing data structures (such as maps) to figure out where the mapping from number to name is
- Running the second file through a hex viewer immediately reveals where it comes from:
  - Follows the same general packet-oriented structure as the other files
  - Contains names in plain text in packets with corresponding numbers

```
000002d0: 58f1 052c 4000 0000 0800 0000 0000 0000  X..,@...........
000002e0: 0000 0000 0004 0001 0000 0000 0000 0000  ................
000002f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000300: 0000 0000 0000 0610 0000 0000 0e00 0000  ................
00000310: 0100 0000 4c65 6f6e 6172 646f 2047 616c  ....Leonardo Gal
00000320: 6c69 e252 ad84 58f1 052c 4000 0000 0800  li.R..X..,@.....
00000330: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000340: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000350: 0000 700d 0000 a005 0000 0000 0210 0000  ..p.............
00000360: 0000 0000 0000 0000 0000 e252 ad84 58f1  ...........R..X.
00000370: 052c 4000 0000 0800 0000 0000 0000 2000  .,@...........  .
00000380: 0000 0304 0001 0000 0000 7c8b e301 0000  ..........|.....
00000390: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000003a0: 0000 0000 0110 0000 0000 0000 0000 0000  ................
000003b0: 0000 e252 ad84                           ...R..
```

Figure: Hexdump of a test recording

What About Video?

- Proved to be quite a bit of a challenge
- Looking at only video data in a hex viewer suggested some form of H.264 encoding
  - Network Abstraction Layer Unit[6] start code prefixes (`0x00 0x00 0x00 0x01`) are plenty
- Running the video data through `ffmpeg` resulted in nothing useful:

Initial Result

[6] NALUs abstract the underlying storage of bits in a "network-friendly" manner

zlt Framework

- Video decoding implemented in zlt framework
- Full of virtual method calls and over 400 classes
- Almost no symbols, exports or imports
- Preliminary dynamic analysis did not reveal anything obvious

```
Offset|Size|struct sc_cabac_decoder
      |    |{
  0000|0008|  Vtable_sc_cabac_decoder *__vftable;
  0008|0008|  _BYTE gap_8[8];
  0010|0008|  CDecBitstream2 *bitstream;
  0018|0004|  int field_18;
  001C|0004|  int codIRange;
  0020|0004|  int codIOffset;
  0024|0004|  int stuff3;
  0028|0008|  __int64 field_28;
  0030|0008|  __int64 field_30;
  0038|0008|  _BYTE is_not_pcm[8];
  0040|0008|  __int64 field_40;
  0048|0008|  __int64 field_48;
  0050|0004|  _BYTE field_50[4];
  0054|0004|  signed int is_si_slice;
      |0058|};
```

Figure: Example of reversed C++ class

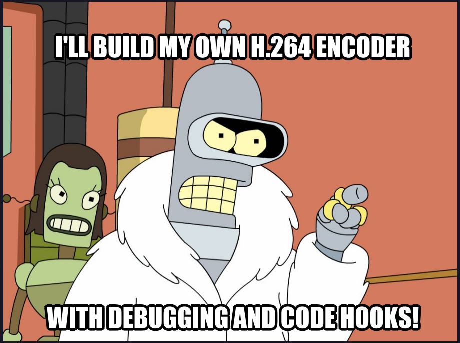
zlt Virtual Functions Example

```
__int64 __fastcall ns_avc::zltCResidualCABACParser::sub_1334CA(ns_avc::zltCResidualCABACParser *this, int a2)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v20 = *((_QWORD *)this + 3);
  v13 = *(unsigned __int8 *)(v20 + 1997);
  v14 = *(_QWORD *)(**((_QWORD **)this + 4) + 64LL);
  v2 = 0;
  v3 = 0LL;
  v4 = 0LL;
  result = 0LL;
  do
  {
    v6 = v13;
    if ( _bittest(&v6, v2) )
    {
      if ( a2 )
      {
        v7 = byte_192ED0[v4];
        v15 = byte_192E70[v4];
        v8 = *((_QWORD *)this + 6);
        result = (*(__int64 (__fastcall **)(_QWORD, __int64, __int64, _QWORD, _QWORD))(***((_QWORD ***)this + 5) + 112LL))(
                   **((_QWORD **)this + 5),
                   v8 + v3,
                   5LL,
                   (unsigned int)v4,
                   *((unsigned int *)this + 4));
        *(_BYTE *)(v20 + v15) = *(_BYTE *)(v8 + v3 + 9);
        v9 = *(_BYTE *)(v8 + v3 + 9);
        v10 = v14;
        *(_BYTE *)(v14 + v7 + 5) = v9;
        *(_BYTE *)(v14 + v7 + 4) = v9;
        *(_BYTE *)(v14 + v7 + 1) = v9;
      }
      else
      {
        v21 = byte_192E70[v4];
        v16 = *((_QWORD *)this + 6);
        v17 = byte_192ED0[v4];
        (*(void (__fastcall **)(_QWORD, __int64, __int64, __int64, _QWORD))(***((_QWORD ***)this + 5) + 112LL))(
          **((_QWORD **)this + 5),
          v16 + v3,
          2LL,
          v17,
          *((unsigned int *)this + 4));
```

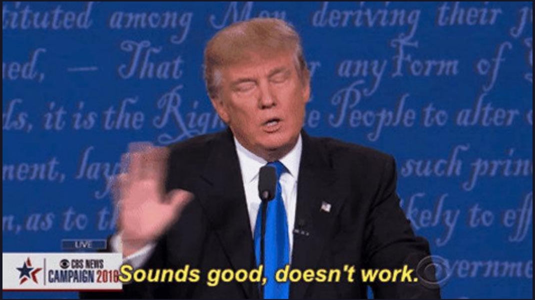
DIY H.264 Decoder

- H.264 specification is very difficult to understand
- ffmpeg's implementation has no comments and does not follow the specification closely
  - Debugging and changing ffmpeg would be difficult (or so I thought)
- **Idea:** Let's build our own decoder made for debugging!



DIY H.264 Decoder

- **Bad Idea:** ~~Let's build our own decoder made for debugging!~~
- Even just parsing H.264 is extremely complicated
- Lots of intricacies and weird stuff going on



Back to zlt

- Took another look at the zlt framework
- Managed to reverse engineer quite a lot of their code
  - At first, relied way too much on static analysis
  - Using a debugger helped immensely
- Found some interesting things while poking around
- Video is already stored in bad quality, so that proved a bit pointless

Findings

Findings
Recording Files

Basic File Layout

- First, a file header containing information like the version and offset of actual data
- Data part of file is split into many small "packets":
  - Delimited by `0x2C05F158` (header) and `0x84AD52E2` (trailer)
- Every packet has:
  - `int32_t type`: specifies type of packet (e.g. video, audio)
  - `int32_t prop_size`: specifies size of property data
  - `int32_t data_size`: specifies size of actual data

0400	58	f1	05	2c	ff	ff	ff	ff	00	00	00	00	f2	69	e3	01
0410	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0420	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0430	00	00	00	00	e2	52	ad	84	58	f1	05	2c	04	00	00	00
0440	00	00	00	00	84	71	e3	01	00	00	00	00	c0	00	00	00
0450	0a	00	00	00	00	00	00	00	00	00	00	00	80	02	00	00
0460	18	00	00	00	00	00	00	00	00	00	00	00	00	7d	00	00
0470	02	04	00	01	00	00	00	00	00	00	00	00	00	00	00	00
0480	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0490	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

File Purposes

- `double_click_to_convert_01.zoom` contains *screenshare*, *webcam*, *avatar* and *cursor* sample packets
- `double_click_to_convert_02.zoom` contains all command packets
- `double_click_to_convert_03.zoom` contains the audio sample packets

Types of Samples

- *Audio*, *Screen Share* and *Webcam* were already discussed
- *Cursor* stores a bmp of the current cursor alongside its screen position
- *Avatar* stores a bmp of the avatar of a person

Figure: Example of a cursor image.

In-Depth Format Description

A more in-depth format description as well as tools for extracting media are available on my GitHub page.

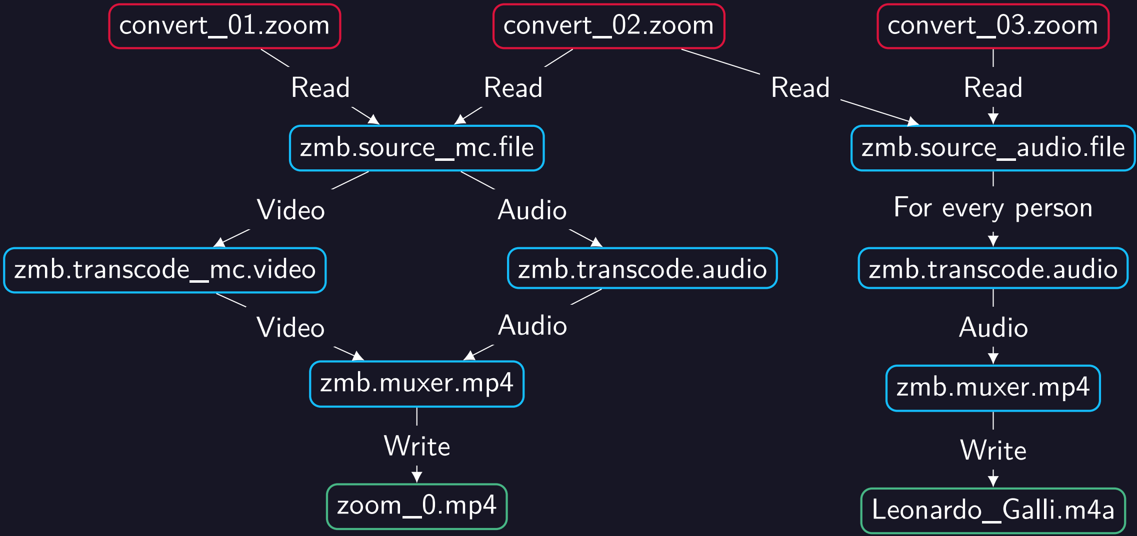
Findings

Other Interesting Bits

Software Architecture

- Organization extremely modular
  - Some parts of the modularization seem unnecessary
  - Most exported C++ classes have C wrappers for no discernable reason
- zmb uses a pipeline architecture
  - Individual operations (e.g. reading a file, converting video) are nodes in a graph
  - Nodes communicate between each other
  - Does not seem to be used much, except for outputting audio tracks by person

Transcoding Pipeline



Transcoding Engine

- zlt seems to implement their own version of an H.264 encoder / decoder
- One small bug in the H.264 implementation:
  - `write_bits(3, &flag)` instead of `write_bits(1, &flag)` in one header
  - First hurdle trying to decode the H.264 stream using other programs
- There seems to be a boolean flag to enable / disable doing wildly non-spec-compliant things
  - Makes reversing and reading the data a lot harder
  - Can force H.264 by selecting: “Optimize Screen Share for Video Clip”
- Fully functional hardware decoding support found in zlt
  - Likely not used due to aforementioned spec non-compliance

Useful Links

- Go library and tool for working with these files: `github.com`

Tools

- Disassemblers: Cutter (`cutter.re`), Ghidra (`ghidra-sre.org`), IDA Freeware (`www.hex-rays.com`)
- Binwalk (`github.com`)

Other

- flagbot homepage: `flagbot.ch`
- H.264 Specification (`www.itu.int`)
- angr for symbolic execution (`angr.io`)

Questions?