How to Approach Binary File Format Analysis

Essential knowledge for reverse engineering

Andreas Pehnack

Summer 2015

Binary File Format Analysis

Get the habit of analysis - analysis will in time enable synthesis to become your habit of mind.

Frank Lloyd Wright

Why analyze Binary File Formats?

There are plenty of reasons why you could care about all the bits and bytes in binary files. Almost all files the average computer user reads and writes with applications like word processors, audio recorders, or video editing software, are binary files. However, computer specialists often need to dig deeper and want to extract, modify or simply understand the contents of such files.

Digital forensics experts often have to identify traces of suspicious activities in binary files.

Malware analysts need to dissect not only executable files in order to fully understand malicious software.

Broken files like interrupted audio or video recordings often can only be recovered with in-depth knowledge of the structure of these files.

Programmers of software that writes and reads binary files require a good understanding of how their data structures are represented in a file format. If something goes wrong a manual analysis of output files can be necessary.

Finally, reverse engineering and analysis of binary files can simply be fun and lets you for example modify high scores in saved games.

What will you get out of this book?

You will learn

- basics of binary file formats
- common patterns found in many different formats
- how to approach completely unknown file formats

It's helpful if you already know

- fundamentals of computer science like what a bit or byte is
- a programming language like C or Python

If you're already familiar with the fundamentals of binary file formats you can skip the first two chapters.

This book cares mainly about analysis of data files; information on how to decode executable file formats like PE (Portable Executable, Windows), MACH-O (Mac) or ELF (Linux) can be found at various places in the Internet.

Basics

Simple solutions seldom are. It takes a very unusual mind to undertake analysis of the obvious.

Alfred North Whitehead

Introduction

Whatever information some application software wants to make persistent in a file has to have some well-defined representation. And all these information have to be written according to some rules in order to allow reading the data back to memory.

For you this means that analysis of a binary file is mainly about understanding *file structure* and the *meaning of certain information*.

While binary files primarily consist of bytes, on file format level you mainly find text and numbers, often organized in structures. Learn here about the atoms, later we'll combine them to molecules.

Text Strings

Character Encoding

Since computers only know how to process numbers, characters have to be mapped to some numeric representation. This assignment is also called text encoding. The most encodings are based on US-ASCII with its 128 definitions (7 bits). In order to store text of languages like French or Russian so-called code pages were invented which consume the second 128 mappings of a byte.

Later Unicode was born which defines more than 120,000 characters. Additionally it allows to switch reading direction for languages like Hebrew or Arabic. Be aware that there are different ways Unicode text can be represented in a file. A very common one is UTF-8 that consumes up to 6 bytes per character but also UTF-16 can be found in different file formats.

File formats which have their origins in the IBM world often use EBCDIC encoded text. EBCDIC doesn't define the characters in consecutive order like ASCII which often causes software to fail when compiled on EBCDIC-based machines.

Storage

Apart from different character encodings there are some common ways how the text length is defined in a file. In many cases the representation in a file equals the storage in main memory.

Fixed Length

In many file formats the maximum length of strings is fixed. Typically the remaining space after the text to be stored is filled with zero bytes.

S	o	m	e		s	a	m	p	l	e		t	e	x	t	\0	\0	\0	\0	\0	\0

String with length 22 and 16 characters

Zero terminated

Since C is a very popular programming language for more than 40 years many file formats contain strings as they are stored in main memory by software developed in C. The end of such strings is marked with a single zero byte or two zero bytes for double-byte character encodings.

S	o	m	e		s	a	m	p	l	e		t	e	x	t	\0

String with length 17 and 16 characters

Pascal

Programming languages like Pascal, Modula or Delphi store the string length in the first byte of a string. Accordingly strings in file formats written by such software often follows the same convention. The first byte of a string contains the number of characters that follow.

16	S	o	m	e		s	a	m	p	l	e		t	e	x	t

Pascal string with number of following characters in the first byte

Numbers

While text strings play a significant role in binary files numbers are even more important. They are not only used to represent raw countable entities but can also stand for enumerations or masks.

Byte Order (Endianness)

A concept you definitely need to know when dealing with binary file formats is endianness. The term is related to the story of Gulliver's Travels written by Jonathan Swift.

For our purposes you simply have to keep in mind that the order of the bytes of numbers are reversed in some file formats (Little Endian). This is mainly the case in formats generated on computers with Intel processors.
Be aware that this normally doesn't play a role when programming in a higher-level language. However, if you look at a memory dump of number variables you notice the inverse byte order.
There are some file formats like TIFF which which can contain either little or big endian numbers, depending on some indicator in the file.

Example for a 32 bit / 4 byte number:

Byte Order \ Number	0x12345678 (decimal 305419896)	0x87654321 (decimal 2271560481)
Big Endian	12 34 56 78	87 65 43 21
Little Endian	78 56 34 12	21 43 65 87

Important: the byte order is reversed, **not** the bit order!

Integer numbers

There are basically two distinctions you have to make when dealing with an integer number: its size and whether it is signed or unsigned.
Typical integer sizes are 1, 2, 4 and 8 bytes. Formats aiming to minimize file sizes often contain numbers with arbitrary bit sizes.

Offsets

In binary file formats you'll often find numbers which are interpreted as file offsets. This can be considered a pointer to a certain position in a file where parsing should continue.
There are both absolute and relative offsets. Absolute means that the offset is based on the start of the file. Relative offsets are added to some other position in the file.

Floating-point numbers

In order to allow to represent very small and very large figures, floating point numbers were invented. Almost always they conform to the IEEE 754 standard, mostly with sizes 4 or 8 bytes. Half precision floats (2 bytes) or quadruple precision floats (16 bytes) are a rare species in binary files.

Flags

Flags are generally used to indicate if some "feature" is on or off. On file level this means that one bit represents the flag's value (zero means off/false, one means on/true). Often multiple related flags are combined in a number value.
Example of window style flags in Windows:

Name	Mask (hexadecimal)	Description
WS_EX_ACCEPTFILES	0x00000010	The window accepts drag-drop files.
WS_EX_CLIENTEDGE	0x00000200	The window has a border with a sunken edge.
WS_EX_CONTEXTHELP	0x00000400	The title bar of the window includes a question mark.
WS_EX_APPWINDOW	0x00040000	Forces a top-level window onto the taskbar when the window is visible.

In order to test if a flag bit in a number is zero or one, masks are used. A mask is a number that has the bit(s) set you want to test. If a binary AND operation of the mask and the value results in a value larger than zero, the flag is set.

Flag	Mask (binary)	Mask (hexadecimal)	Mask (decimal)
1	00000001	0x01	$1 = 2^0$
2	00000010	0x02	$2 = 2^1$
3	00000100	0x04	$4 = 2^2$
4	00001000	0x08	$8 = 2^3$
5	00010000	0x10	$16 = 2^4$
6	00100000	0x20	$32 = 2^5$
7	01000000	0x40	$64 = 2^6$
8	10000000	0x80	$128 = 2^7$

Mask values can also be used to set single bits in a number. Simply perform a binary OR operation of the mask and the value.

Value (binary)	Mask (binary)	AND result	OR result
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Structures

In programming languages structures are named collections of elements like numbers, strings or other objects. On file level things look a bit different. Since there is no independent name tag structures have to be identified by position or some element inside the structure.

Alignment / Padding

In programming languages like C, the elements of structures are aligned by default to byte positions which are multiple of 2, 4, 8, or 16 depending on the processor structure. Similarly, in file formats structures or elements of structures are sometimes aligned to multiple of 2 or 4 bytes.
Structure alignment can be found for example in IFF85 or TrueType file formats.
Usually the padding area before some aligned structure is filled with 0x00 bytes.

Binary File Formats

The greatest moments are those when you see the result pop up in a graph or in your statistics analysis - that moment you realise you know something no one else does and you get the pleasure of thinking about how to tell them.

Emily Oster

Typical Patterns

When looking at a bunch of binary file formats many of them share some basic concepts.
Always keep in mind that

- there is some order in which the file contents are read
- the more flexible a file format the more hints there are how to read it

The simplest possible file format contains only elements (numbers, strings, …) at fixed file positions. If there is for example an optional element there must be additional information in the file that indicates if the element is to be read or not.

Likewise for alternative structures — somewhere inside or outside of the structures there must be some hint which structure should be read.

Eye Catchers/Magic Numbers

In order to identify a file type not only by its file name extension many file formats start with a unique byte sequence. The UNIX file command is able to detect the format of files mostly depending on magic numbers.

Format	Text (ASCII)	Bytes
GIF	“GIF89a”	47 49 46 38 39 61
PDF	“%PDF”	25 50 44 46
ZIP	“PK”	50 4B

File Headers

Most binary file formats start with some structure called file header. Depending on the specification a magic number may be part of the the header or is followed by the header.
Typical contents of file headers are version numbers, file offsets of other structures in the file, image width and height, or in general information necessary in order to read the rest of the file.
In some file formats where the writing software doesn't always know in advance what will be written to a file some structure can be found at the end of the file allows accessing the file contents. Examples are ZIP or PDF.

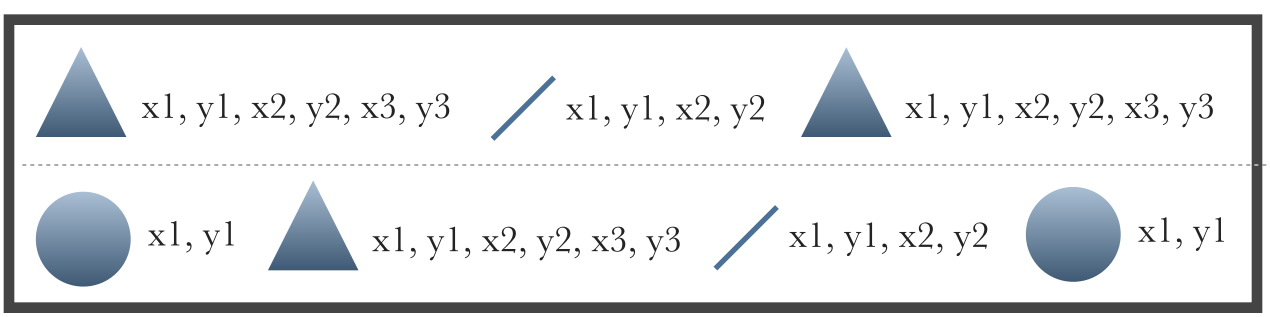
Structure Lengths

Depending on the data some structure comprises it may have a fixed or variable byte length.
Structure length types:

Structure Length	Description
Fixed	The length is not mentioned in the file but known by the reading application
Length element	Inside or outside of the structure (e. g. in the file header) the length of the structure is stored in a number element
Delimiter	The structure itself or elements of it are continued until a certain byte sequence. Typical example of this case are zero-terminated strings.

Alternative Structures

Let's say a hypothetical file format can store structures for points, lines and triangles in any order. Apart from the raw coordinates the reading program must get some hint which object to read next.



A binary representation of the data could look like this (assuming that every number fits into one byte):

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	03	x1	y1	x2	y2	x3	y3	02	x1	y1	x2	y2	03	x1	y1	x2
1	y2	x3	y3	01	x1	y1	03	x1	y1	x2	y2	x3	y3	02	02	x1
2	y1	x2	y2	01	x1	y1										

In this case additional lengths of the structures is not necessary — a triangle always consists of three and a line of two points.

Unknown Formats

My mind rebels at stagnation. Give me problems, give me work, give me the most abstruse cryptogram, or the most intricate analysis, and I am in my own proper atmosphere. But I abhor the dull routine of existence. I crave for mental exaltation.

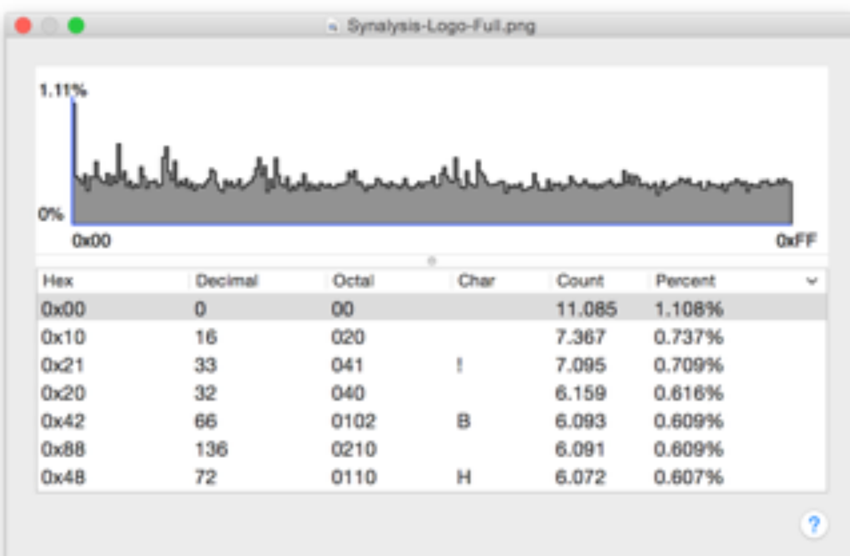
Arthur Conan Doyle

How to Start

Let's assume you want to decode a binary file and have no specification. Here I propose some simple steps you can take to learn more about such a file. So far the human brain is still one of the best instruments to detect patterns and make sense of them.

Histogram

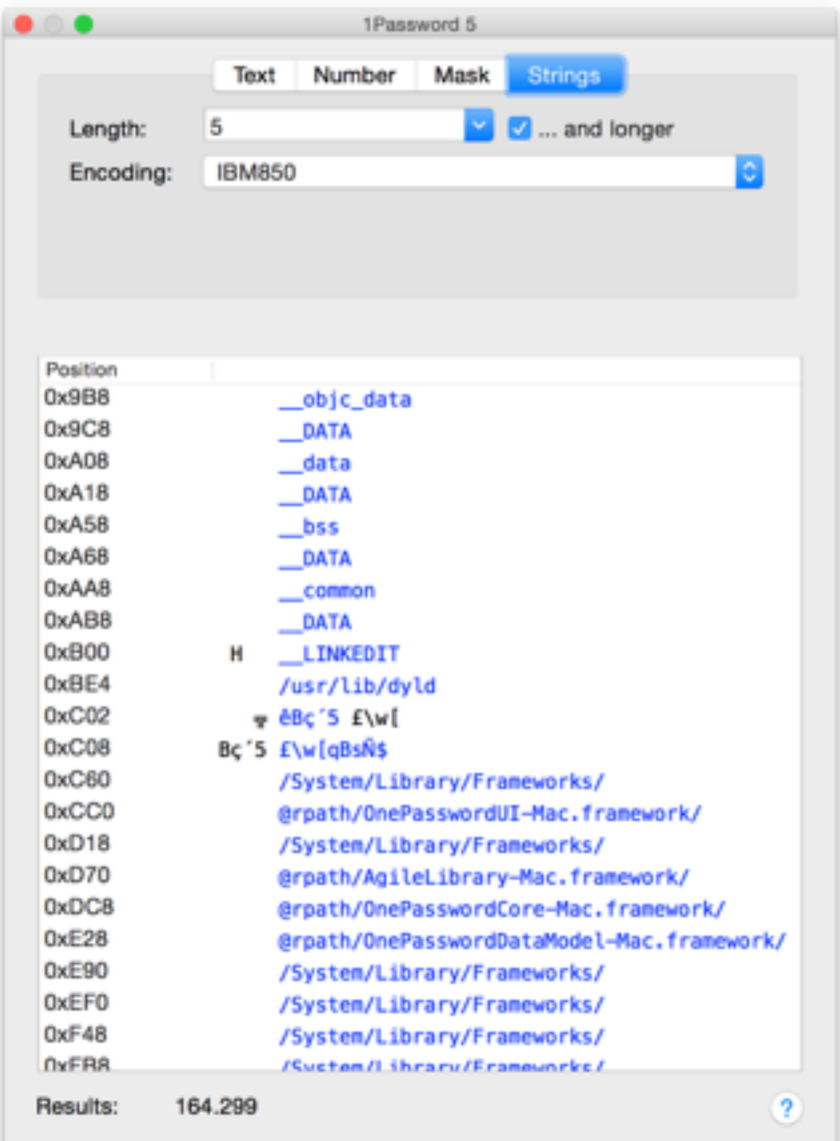
A handy tool to learn more about the characteristics of a file is the histogram. It basically shows how often each byte value (0-255) appears in a file.



What can a histogram tell you about a file? First, a very equal distribution of the byte counts indicates that the file is compressed or encrypted. Second, single peaks are hints which bytes play an important role in the file format. Often you can ignore a peak of 0x00 bytes because most strings or other data is padded with them.

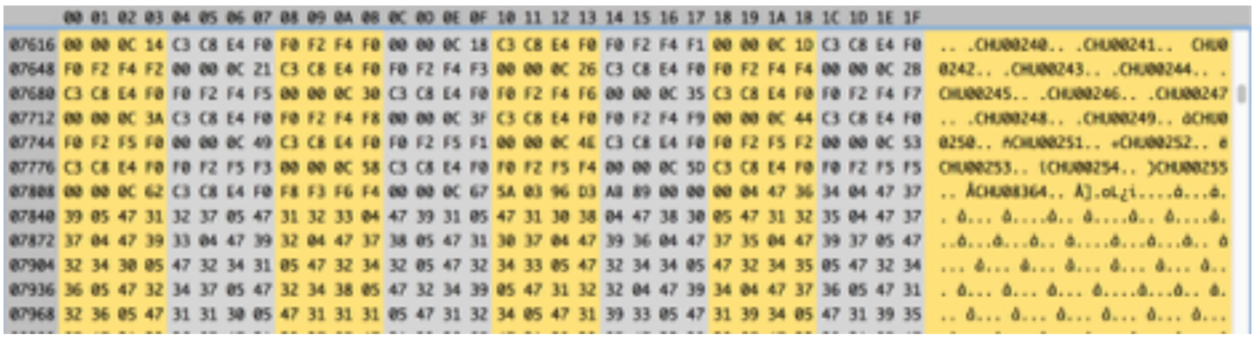
Strings

Another useful tool that gives an impression of how a file is structured is *strings*. You can use the command-line version on Unix or a GUI tool like Hexinator or Synalyze It!



Sections

Many file formats consist of different sections. Mostly you can identify them by scrolling quickly through a file in a hex editor and looking at the text column.



There are three possible ways these sections can be read by the application who "knows" the file format:

- The sections start at fixed positions which are always the same in all files of that format
- File offsets refer to the section starts, often stored in the file header
- The length of all preceding sections is known

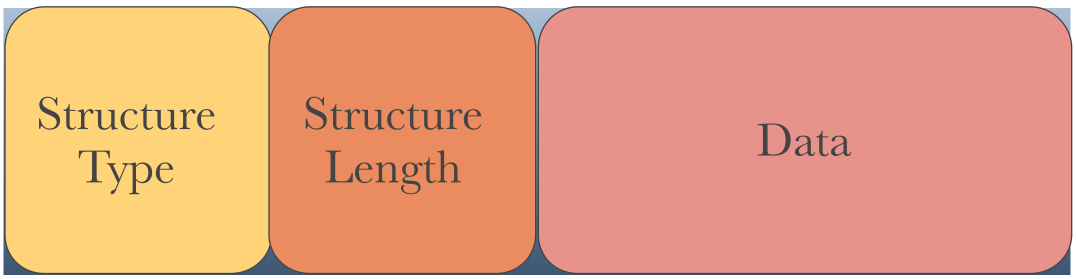
If you write down the start positions and lengths of all sections you can visually identify you can try to find this information later at other places in the file.

File Header

Since most file formats contain some kind of header that marks the starting point for decoding the rest of the file here is a good place to search for the file offsets you noted previously. Mostly file offsets are stored as 4-byte numbers, in newer formats the numbers may also take 8 bytes.
Also try to identify other information you know it's stored in the file. For example, if you know that the file contains an image of a certain width and height, try to find these values. Be aware that numbers may be stored in reverse byte order (little endian).

Structures

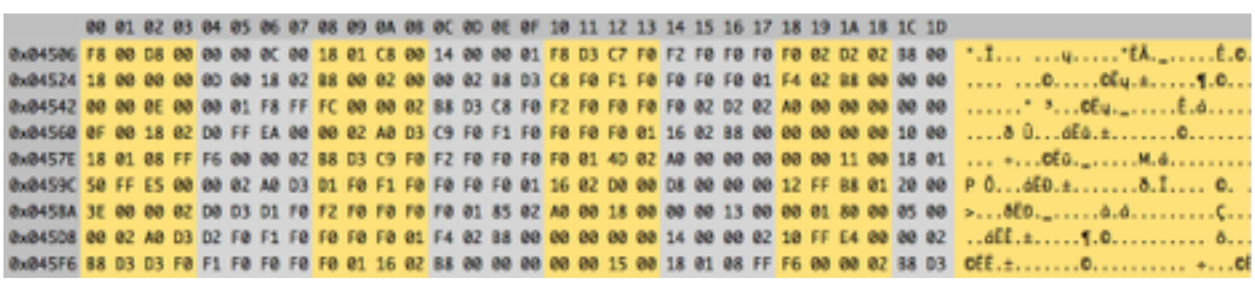
If you take a closer look at a section in a file you may notice that it consists of a series of similar structures. Often these structures follow some simple schema and start with an identifier and some length field that holds the structure length in bytes. In some cases there's only an identifier of the structure type and the length is not mentioned explicitly because that type of structure always has the same length.

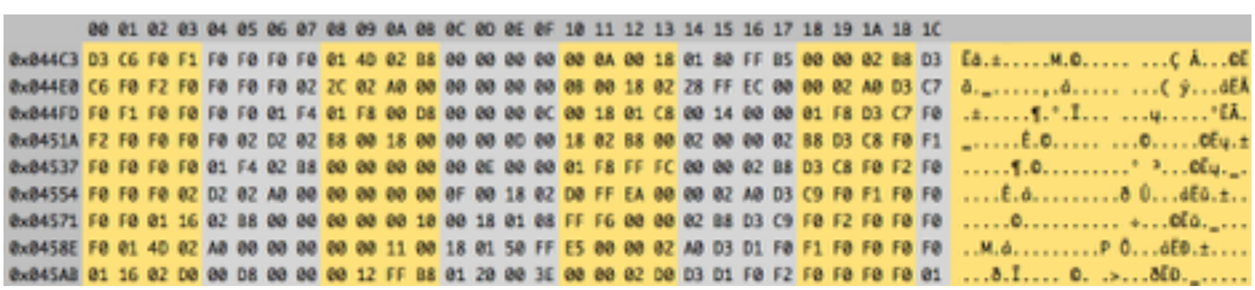


Sometimes the length field contains the length of the whole structure, sometimes only of the following data so don't look for exact numbers.

Here's how you can easily determine the size of fixed-length structures: With a hex editor that allows to set an arbitrary number of columns simply change the column count until similar patterns in the hex or text view are aligned vertically.

Now you can easily read the structure length from the headline of the hex view.





	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	
0x04408	F4	02	D0	00	00	00	00	00	03	00	00	01	E0	00	14	00	00	02	D0	D3	C2	F0	F2	F0	F0	F0	F0	02	¶.ð..........Ô.....ðË┬._......
0x04424	9B	02	A0	00	00	00	00	00	04	00	18	02	58	00	28	00	00	02	A0	D3	C3	F0	F1	F0	F0	F0	F0	01	ø.ó..........X.+...óË├.±......
0x04440	BC	01	F8	00	00	00	00	00	05	00	18	01	98	00	0C	00	00	01	F8	D3	C3	F0	F2	F0	F0	F0	F0	02	Ŷ.'..........ŷ. ...'Ë├._......
0x0445C	9B	02	B8	00	18	00	00	00	06	00	18	02	70	00	13	00	00	02	B8	D3	C4	F0	F1	F0	F0	F0	F0	01	ø.©..........p.....©Ë─.±......
0x04478	F4	02	D0	00	00	00	00	00	07	00	18	01	E0	FF	FC	00	00	02	D0	D3	C5	F0	F1	F0	F0	F0	F0	01	¶.ð..........Ô ³...ðË┼.±......
0x04494	BC	01	F8	00	00	00	00	00	08	00	18	01	98	00	0C	00	00	01	F8	D3	C5	F0	F2	F0	F0	F0	F0	02	Ŷ.'..........ŷ. ...'Ë┼._......
0x044B0	63	02	A0	00	00	00	00	00	09	00	18	02	58	FF	F3	00	00	02	A0	D3	C6	F0	F1	F0	F0	F0	F0	01	c.ó..... ...X Ж...óËã.±......
0x044CC	4D	02	B8	00	00	00	00	00	0A	00	18	01	80	FF	B5	00	00	02	B8	D3	C6	F0	F2	F0	F0	F0	F0	02	M.©..... ...Ç Å...©Ëã._......
0x044E8	2C	02	A0	00	00	00	00	00	0B	00	18	02	28	FF	EC	00	00	02	A0	D3	C7	F0	F1	F0	F0	F0	F0	01	,.ó..... ...(ý...óËÃ.±......

How to Start?

An absolute can only be given in an intuition, while all the rest has to do with analysis.

Henri Bergson

Now after you learned the basics of binary file formats you probably want to explore some real file. There are some tutorials that guide you step by step through the analysis of some files:
https://www.synalysis.net/tutorial-decode-a-png-file.html
https://hexinator.com/tutorial-decode-adobe-swatch-exchange-file/

Evaluation copies of the software used in the tutorials can be downloaded for free on these web sites:
https://hexinator.com (Windows, Linux) and
https://www.synalysis.net/ (OS X).

Feedback

If you want to give any feedback regarding this small book, please send an email to andreas@synalysis.com and let me know what you think.