Computer Foundations

Digital Forensics - ay 2022/2023

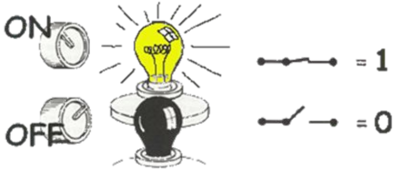
Enrico RUSSO <enrico.russo@dibris.unige.it>

Outline

- numbering formats and positional notation
- digital data and endianness
- characters encoding and strings
- data structure
- hexadecimal editors
- file structure
- data encoding with Base64

Hands-on session.

Number formats



System	Base	Symbols	Used by humans?	Used by CPU?	Used by Forensic Professionals?
Decimal	10	0, 1, … 9	Yes	No	Yes
Binary	2	0, 1	No	Yes	Yes
Hexa-decimal	16	0, 1, … 9, A, B, … F	No	No	Yes

https://github.com/frankwxu/digital-forensics-lab

Positional notation: decimal number

A decimal number is a series of symbols/digits (0-9), and each digit has a value.

Consider the series of digits 35,182. We calculate the corresponding decimal value using **positional notation**.

Index

4	3	2	1	0	
3	**5**	**1**	**8**	**2**	10

Radix

Most significant symbol

Position

Digit

Least significant symbol

A **positional system** is a numeral system in which the **contribution of a digit** to the **value of a number** is the **value** of the digit **multiplied by a factor** determined by the **position** of the digit.

Positional notation: decimal number

The factor is **radix$^{\wedge\text{position}}$**.

The decimal value is: (3x10,000)+(5x1000)+(1x100)+(8x10)+(2x1) = 35,182

10^4	10^3	10^2	10^1	10^0
3	**5**	**1**	**8**	**2**

$_{10}$

Positional notation enables a **general process** to determine the **decimal value** of **non-decimal numbers**.

Positional notation: binary numbers

A binary number has only **two digits** (1, 0).

We convert the binary number 1001 0011 to its decimal value. The radix is 2.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	**0**	**0**	**1**	**0**	**0**	**1**	**1**

2

The decimal value is

(1x128)+(0x64)+(0x32)+(1x16)+(0x8)+(0x4)+(1x2)+(1x1) = 147

Positional notation: hexadecimal numbers

A hexadecimal number has **16 digits** (the numbers 0 to 9 followed by the letters A to F).

We convert the hexadecimal number 0x8BE4 to its decimal value. The prefix ‘**0x**’ is used to **differentiate** it from a **decimal number**.

0xB = 11	
0xE = 14	

16^3	16^2	16^1	16^0
8	**B**	**E**	**4**

16

The decimal value is (8x4.096)+(11x256)+(14x16)+(4x1) = 35.812

Converting binary to hexadecimal

Converting between bin to hex and hex to bin is easy because it requires only **lookups**.

It requires **grouping bits in four**.

1001 0011 to hex

1001	0011
⇩	⇩
0x9	0x3

0x93 to bin

0x9	0x3
⇩	⇩
1001	0011

Binary	**Decimal**	**Hexadecimal**
0000	00	0
0001	01	1
0010	02	2
0011	03	3
0100	04	4
0101	05	5
0110	06	6
0111	07	7
1000	08	8
1001	09	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Linux facilities

Requires bc - “An arbitrary precision calculator language” (`apt install bc`)

- Dec to Hex: `echo 'obase=16;10'| bc`
- Dec to Bin: `echo 'obase=2;10' | bc`
- Hex to Dec: `echo 'ibase=16;A' | bc` or `echo $((0xA))`
- Bin to Dec: `echo 'ibase=2;1010' | bc` or `echo $((2#1010))`

Data Sizes

- to store digital data, we need to allocate a **location** on a storage device (e.g., disk or memory)
- a **byte** is the **smallest amount of space** that is typically allocated to data
- a byte can hold only 256 values, so **byte are grouped together** to store large numbers
- typical sizes include **2** (word)**, 4** (doubleword) or **8** (quadword) **bytes**
- computers **differ** in how they organize **multiple-byte** values

Endianness

Endianness is a term that describe the **order** in which a **sequence of bytes** (byte order) is **stored** in memory.

- **Little-endian**: is an order in which the "little end" (least significant value in the sequence) is stored first
- **Big-endian**: is an order in which the "big end" (most significant value in the sequence) is stored first (at the lowest storage address)

Most significant value

Least significant value

Data: 0x01020304

0x100	0x101	0x102	0x103	Location
0x04	0x03	0x02	0x01	

0x100	0x101	0x102	0x103	Location
0x01	0x02	0x03	0x04	

Endianness [2]

- **IA-32** based systems and their 64-bit counterparts use the **little-endian** ordering
  - we need to “rearrange” the bytes if we want the most significant byte to be the left-most number
- SUN SPARC and Motorola PowerPC systems use big-endian ordering

```
lscpu

Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         45 bits physical, 48
bits virtual
  Byte Order:            Little Endian
CPU(s):                  64
  On-line CPU(s) list:   0-63
Vendor ID:               GenuineIntel
  Model name:            Intel(R) Xeon(R) Gold
6252N CPU @ 2.30GHz
    CPU family:          6
    Model:               85
    Thread(s) per core:  1
    Core(s) per socket:  8
    Socket(s):           8
```

Strings and Character Encoding: ASCII

Computers store numbers but also letters and sentences. The most common technique is to encode characters using **ASCII** or **Unicode**.

- ASCII assigns a numerical value to the characters in **American English**
- It uses 7 bit and the largest defined value is 0x7E (an “extended” version can use 8 bit and incorporates non-English characters, cfr. ISO/IEC 8859-x)
- It has many values that are defined as control characters and are **not printables**, e.g., the 0x07 bell sound

ASCII table

ASCII control chars (not printables)

```
ascii -t
```

Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex	
0	00	NUL	16	10	DLE	32	20		48	30	0	64	40	@	80	50	P	96	60	`	112	70	p
1	01	SOH	17	11	DC1	33	21	!	49	31	1	65	41	A	81	51	Q	97	61	a	113	71	q
2	02	STX	18	12	DC2	34	22	"	50	32	2	66	42	B	82	52	R	98	62	b	114	72	r
3	03	ETX	19	13	DC3	35	23	#	51	33	3	67	43	C	83	53	S	99	63	c	115	73	s
4	04	EOT	20	14	DC4	36	24	$	52	34	4	68	44	D	84	54	T	100	64	d	116	74	t
5	05	ENQ	21	15	NAK	37	25	%	53	35	5	69	45	E	85	55	U	101	65	e	117	75	u
6	06	ACK	22	16	SYN	38	26	&	54	36	6	70	46	F	86	56	V	102	66	f	118	76	v
7	07	BEL	23	17	ETB	39	27	'	55	37	7	71	47	G	87	57	W	103	67	g	119	77	w
8	08	BS	24	18	CAN	40	28	(	56	38	8	72	48	H	88	58	X	104	68	h	120	78	x
9	09	HT	25	19	EM	41	29	)	57	39	9	73	49	I	89	59	Y	105	69	i	121	79	y
10	0A	LF	26	1A	SUB	42	2A	*	58	3A	:	74	4A	J	90	5A	Z	106	6A	j	122	7A	z
11	0B	VT	27	1B	ESC	43	2B	+	59	3B	;	75	4B	K	91	5B	[	107	6B	k	123	7B	{
12	0C	FF	28	1C	FS	44	2C	,	60	3C	<	76	4C	L	92	5C	\	108	6C	l	124	7C	|
13	0D	CR	29	1D	GS	45	2D	-	61	3D	=	77	4D	M	93	5D	]	109	6D	m	125	7D	}
14	0E	SO	30	1E	RS	46	2E	.	62	3E	>	78	4E	N	94	5E	^	110	6E	n	126	7E	~
15	0F	SI	31	1F	US	47	2F	/	63	3F	?	79	4F	O	95	5F	_	111	6F	o	127	7F	DEL

ASCII strings

- to store a sentence or a word using ASCII, we need to allocate **as many bytes** as there are **characters in a sentence or word**
- each **byte** stores the value of a **character**
- the **endianness** of a system **does not play a role** in how the characters are stored because these are separate 1-byte values (the first character is always the first allocated byte)
- the **series of bytes** in a word/sentence is called a **string**
- many times, the **string ends** with the **NULL** symbol, i.e., 0x00

Unicode

Unicode is a universal character encoding standard. It is aimed to include all the characters needed for **any writing system** or **language**.

- this standard includes roughly **100,000 characters** to represent characters of different languages.
- uses **4 bytes** to represent characters
- unicode characters can be **encoded/stored** with different methods (Unicode Transformation Format)
    - UTF-32 (4-bytes for each char, might waste a lot of space)
    - UTF-16 (most heavily used chars in 2-bytes, lesser-used in 4-bytes)
    - UTF-8 (uses 1,2,3 or 4 bytes to store chars, and the most frequently used ones use only 1 byte)

Unicode notations

- The Unicode Standard adopted the convention of "U+" followed by hexadecimal digits
  - The characters “U+” are an ASCIIfied version of the MULTISET UNION “⊎” U+228E
- Programming languages use their own notations. For example, the Python language defines the following string literals:
  - `u'xyz'` to indicate a Unicode string, a sequence of Unicode characters
  - `'\uxxxx'` to indicate a string with a unicode character denoted by four hex digits
  - `'\Uxxxxxxxx'` to indicate a string with a unicode character denoted by eight hex digits

UTF-8

UTF-8 is a **variable-length** character encoding standard

- it makes **processing** the data **more difficult** but it has the **least amount** of wasted **space**
- **ASCII is a subset of UTF-8**: a UTF-8 string that has only characters in ASCII uses only 1 byte per characters and has the same values as the equivalent ASCII string

First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4	Code points
U+0000	U+007F	0xxxxxxx				128
U+0080	U+07FF	110xxxxx	10xxxxxx			1920
U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx		[a]61440
U+10000	[b]U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx	1048576

https://en.wikipedia.org/wiki/UTF-8

Linux facilities

- **`xdd`**: creates a hex dump of a given file or standard input (and does the reverse)
- **`hexdump`**: is a built-in Linux utility to filter and display the contents of different files in hex, decimal, octal, or ASCII formats
- **strings**: print the strings of printable characters in files
  - prints the printable character sequences that are at least 4 characters long (or the number specified with an option) and are followed by an unprintable character.

UTF-8: example

😉 (U+1F609) is saved on winking_face.txt

1. `xxd -b winking_face.txt`, dump bits
    a. `00000000: 11110000 10011111 10011000 10001001          ....`
2. `hexdump winking_face.txt`, hexdump by default uses **words** instead of **bytes** and display them following endianness of the running architecture
    a. `0000000 9ff0 8998`
    b. `0000004`
3. hexdump -C `winking_face.txt` (canonical, use byte)

```
00000000  f0 9f 98 89                                       |....|
00000004
```

UTF-8 works on a **byte unit** and there is no endianness in it!

Byte Order Mark (BOM)

The byte order mark (BOM) is a particular usage of the special Unicode character, `U+FEFF BYTE ORDER MARK`, that appears at the start of a text stream and can signal several things to a program reading the text:

- the fact that the text encoding is Unicode
- the endianness, of the text stream in the cases of 16-bit and 32-bit encodings
- which Unicode character encoding is used

BOM use is **optional**.

UTF-32, big-endian	00 00 FE FF
UTF-32, little-endian	FF FE 00 00
UTF-16, big-endian	FE FF
UTF-16, little-endian	FF FE
UTF-8	EF BB BF

UTF-16 example: python

```
# python_encoding.py

# len: 15c + emoji
str = "utf-16 encoding😉".encode("utf-16")

with open("python_UTF16.txt", "wb") as f:
        f.write(str)

f.close()
```

```
#~ file python_UTF16.txt
python_UTF16.txt: Unicode text, UTF-16, little-endian text, with no line
terminators

#~ hexdump -C python_UTF16.txt
00000000  ff fe 75 00 74 00 66 00  2d 00 31 00 36 00 20 00  |..u.t.f.-.1.6. .|
00000010  65 00 6e 00 63 00 6f 00  64 00 69 00 6e 00 67 00  |e.n.c.o.d.i.n.g.|
00000020  3d d8 09 de                                       |=...|
```

Unicode Character “😉” (U+1F609)

UTF-16 Encoding: `0xD83D 0xDE09`

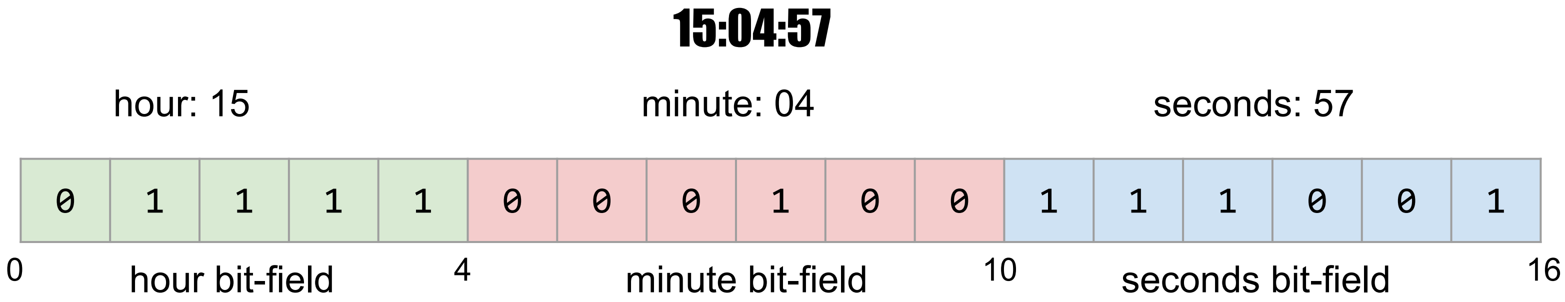
UTF-16 works on a **<u>word</u> unit** and there is endianness in it!

Data structures

Computers know the layout of the stored data because of **data structures**, i.e.,

> “rules that apply to groups of data so we can **understand what data means**”

- a data structure is broken up into **fields**
  - each field has a size and name
  - field’s information **is not saved** with the data



Data structures: example

A basic data structure for the house number and street name. If we want to store the address “1 Main St.”

Byte range	**Description**
0-1	2-byte house number
2-31	30-byte ASCII street name

- We write the number 1 to bytes 0-1
- We write “Main St.” in bytes 2-9
- The remaining bytes can be set to 0 since we do not need them

- We allocated 32 bytes of storage space, and it can be anywhere on the device
- The bytes offset is relative to the start of the space where they are allocated
- The order of the bytes in the **house number** depends on the **endian ordering** of the computer

Data structure: example

For reading data from the storage:

- determine where the data start
- refer to its data structure to find out where the needed values are

1 byte

byte offset (HEX) 16 bytes ASCII equivalent

```
(00) 00000000: 0100 4d61 696e 2053 742e 0000 0000 0000  ..Main St.......
(16) 00000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
(32) 00000020: 1900 536f 7574 6820 5374 2e00 0000 0000  ..South St......
(48) 00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

no printable

(output from raw data → xxd tool)

Data structure: example

```
(00) 00000000: 0100 4d61 696e 2053 742e 0000 0000 0000  ..Main St.......
(16) 00000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
(32) 00000020: 1900 536f 7574 6820 5374 2e00 0000 0000  ..South St......
(48) 00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
(64) 00000040
```

- Two entries: 0 - 31 and 32 - 63
- Bytes 0 - 1 (0x0100) and 32 - 33 (0x1900) are 2 bytes number field: data are from an Intel system, which is little-endian, and we have to switch the order to 0x0001 and 0x0019
- second fields, i.e., bytes 2 - 31 and 34 - 63, are ASCII strings not effected by the endian ordering

Flag values

A data type used to identify **if something exists** (e.g., whether a partition of the computer's storage devices is bootable or not).

- can be represented with either a 1 or a 0
- requires allocating a full byte but this wastes a lot of space because only a bit is needed
    - a more efficient method is to pack several of this binary condition into one value (namely flags)
    - each bit in the value corresponds to a feature or option
    - to read a flag value, we need to convert the number to binary and examine each bit. If the bit is 1, the flag is set

Flag values: example

- the original data structure had a field for the house number and a field for the street name
- we add an optional 16-byte city name after the street name
- because the city name is optional, we need a flag to identify if it exists or not
- the flag is in byte 31 and bit 0 is set when the city exists (i.e., 0000 0001)

Byte range	**Description**
0-1	2-byte house number
2-30	29-byte ASCII street name
31-31	Flags
32-47	16-byte ASCII city name (if flag is set)

Flags: example

```
(00) 00000000: 0100 4d61 696e 2053 742e 0000 0000 0000  ..Main St.......
(16) 00000010: 0000 0000 0000 0000 0000 0000 0000 0061  ...............a
(32) 00000020: 426f 7374 6f6e 0000 0000 0000 0000 0000  Boston..........
(48) 00000030: 1900 536f 7574 6820 5374 2e00 0000 0000  ..South St......
(64) 00000040: 0000 0000 0000 0000 0000 0000 0000 0060  ...............`
```

- The first data structure 0 - 47 has flags in byte 31 with a value of 0x61
- 0x6 and 0x1 correspond to the binary value 0110 0001 where the least significant bit is set, which is the flag for the city
- Bytes 32 - 47 contain the city name
- The second data structure 48 - 79 has flag field in byte 79 (0x60)
- 0x6 and 0x0 correspond to 0110 0000, and the city flag is no set

Reading data structures with python: `struct`

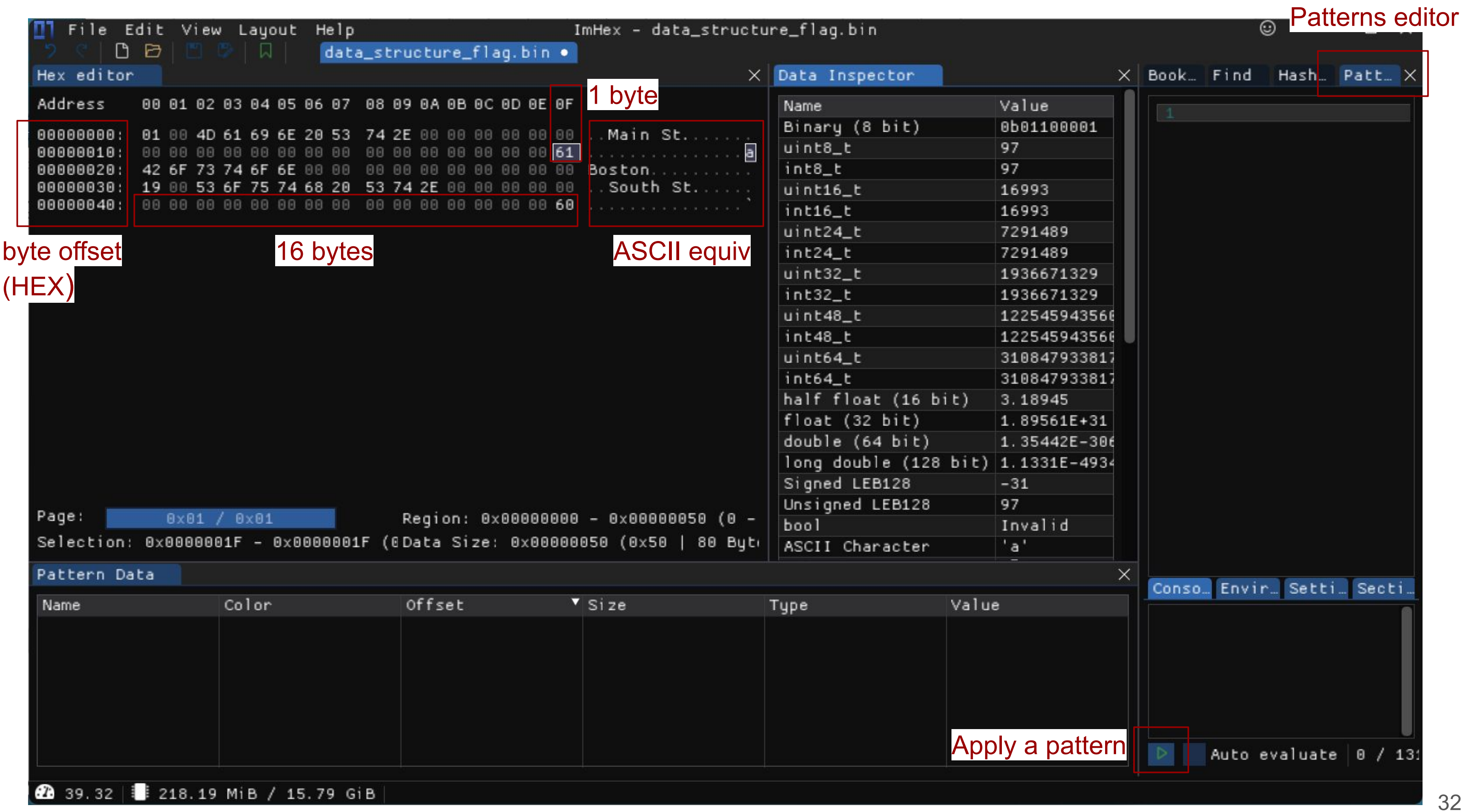
```
import struct

# https://docs.python.org/3/library/struct.html

# < little endian, H unsigned short (2 bytes),
# s string, c char (1 byte)
fmt = "<H29sc"
# unpack from binary buffer according to fmt
struct_unpack = struct.Struct(fmt).unpack_from

length = struct.calcsize(fmt)
print(f"Size of data structure: {length} bytes")
```

```
with open("data_structure_flag.bin", "rb")
 as f:
    data = f.read(length)
    while data:
        res = struct_unpack(data)
        print(f"- house number: {res[0]},
         street name: {res[1].decode()}")
        if res[2][0] & 0x01:
            city = f.read(16)
            print(f"  city:
             {city.decode()}")
        data = f.read(length)
```

Hex editor: ImHex

ImHex is a free and open source Hex Editor

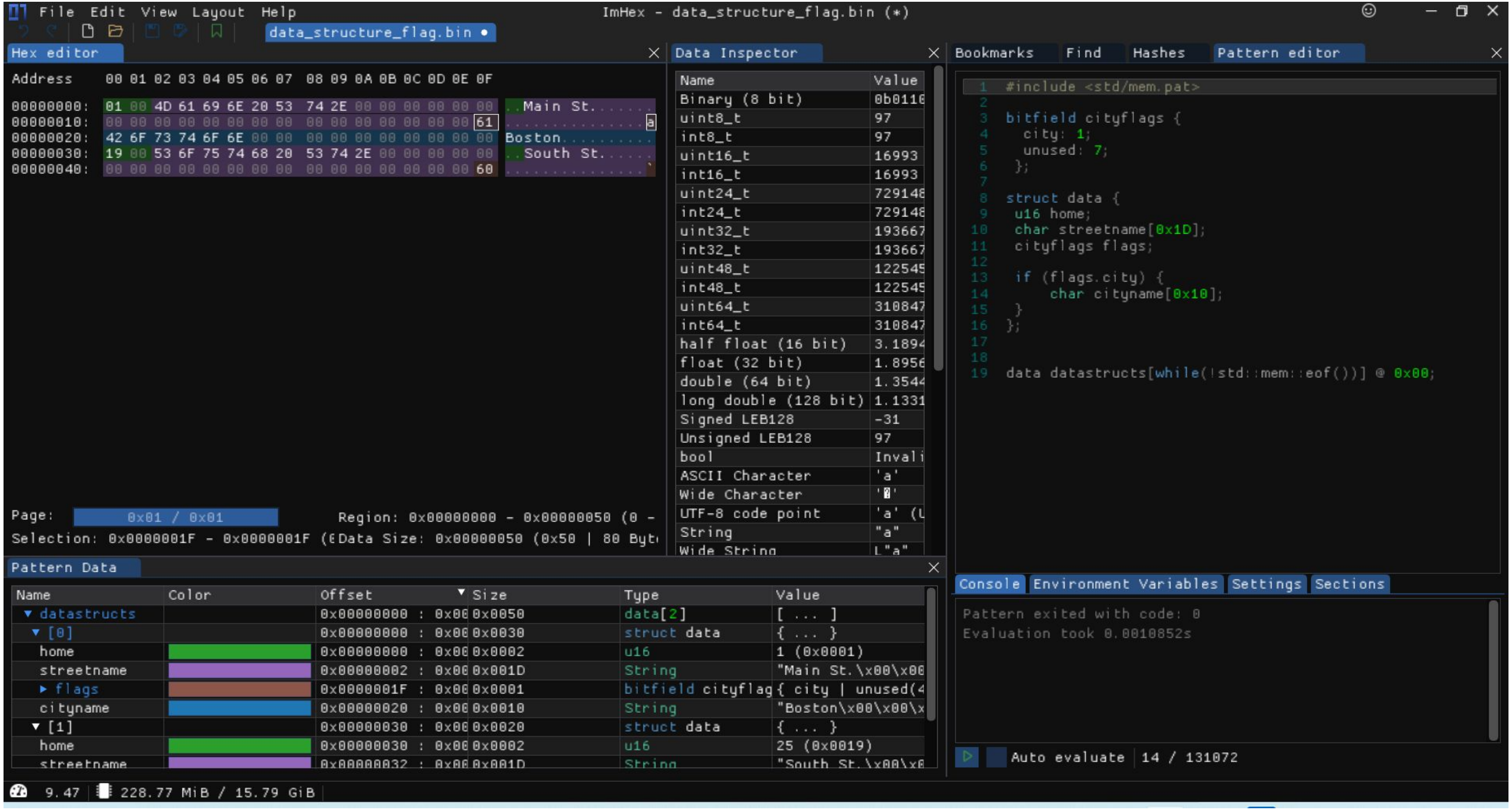
- available at https://imhex.werwolv.net/ (https://github.com/WerWolv/ImHex) for Linux, Windows, and MacOS
- it has many advanced features and, among them, a Custom C++-like pattern language for parsing and highlighting the content a file
    - Pattern Language documentation: https://imhex.werwolv.net/docs/


Patterns editor
File Edit View Layout Help
ImHex - data_structure_flag.bin
data_structure_flag.bin
Hex editor
Address 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000: 01 00 4D 61 69 6E 20 53 74 2E 00 00 00 00 00 00
00000010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 61
00000020: 42 6F 73 74 6F 6E 00 00 00 00 00 00 00 00 00 00
00000030: 19 00 53 6F 75 74 68 20 53 74 2E 00 00 00 00 00
00000040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 60
..Main St.......
...............a
Boston..........
..South St......
...............`
1 byte
byte offset (HEX)
16 bytes
ASCII equiv
Page: 0x01 / 0x01
Region: 0x00000000 - 0x00000050
Selection: 0x0000001F - 0x0000001F
Data Size: 0x00000050 (0x50 | 80 Byte
Data Inspector
Name Value
Binary (8 bit) 0b01100001
uint8_t 97
int8_t 97
uint16_t 16993
int16_t 16993
uint24_t 7291489
int24_t 7291489
uint32_t 1936671329
int32_t 1936671329
half float (16 bit) 3.18945
float (32 bit) 1.89561E+31
Signed LEB128 -31
Unsigned LEB128 97
bool Invalid
ASCII Character 'a'
Book... Find Hash... Patt...
Pattern Data
Name Color Offset Size Type Value
Conso... Envir... Setti... Secti...
Apply a pattern
Auto evaluate
39.32
218.19 MiB / 15.79 GiB

Pattern language: example

```
#include <std/mem.pat>

bitfield cityflags {
  city: 1;
  unused: 7;
 };

struct data {
 u16 home;
 char streetname[0x1D];
 cityflags flags;

 if (flags.city) {
     char cityname[0x10];
 }
};


data datastructs[while(!std::mem::eof())] @ 0x00;
```

Byte range	**Description**
0-1	2-byte house number
2-30	29-byte ASCII street name
31-31	Flags
32-47	16-byte ASCII city name (if flag is set)


File Edit View Layout Help
ImHex - data_structure_flag.bin (*)
data_structure_flag.bin
Hex editor
Data Inspector
Bookmarks Find Hashes Pattern editor
Address 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000: 01 00 4D 61 69 6E 20 53 74 2E 00 00 00 00 00 00 ..Main St......
00000010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 61 ....a
00000020: 42 6F 73 74 6F 6E 00 00 00 00 00 00 00 00 00 00 Boston..........
00000030: 19 00 53 6F 75 74 68 20 53 74 2E 00 00 00 00 00 ..South St.....
00000040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 60 ....`
#include <std/mem.pat>
bitfield cityflags {
city: 1;
unused: 7;
};
struct data {
u16 home;
char streetname[0x1D];
cityflags flags;
if (flags.city) {
char cityname[0x10];
}
};
data datastructs[while(!std::mem::eof())] @ 0x00;
Page: 0x01 / 0x01
Region: 0x00000000 - 0x00000050
Selection: 0x0000001F - 0x0000001F
Data Size: 0x00000050 (0x50 | 80 Byt
Pattern Data
Name Color Offset Size Type Value
datastructs 0x00000000 0x0050 data[2] [...]
[0] 0x00000000 0x0030 struct data { ... }
home 0x00000000 0x0002 u16 1 (0x0001)
streetname 0x00000002 0x001D String "Main St.\x00\x00
flags 0x0000001F 0x0001 bitfield cityflag { city | unused(4
cityname 0x00000020 0x0010 String "Boston\x00\x00\x
[1] 0x00000030 0x0020 struct data { ... }
home 0x00000030 0x0002 u16 25 (0x0019)
streetname 0x00000032 0x001D String "South St.\x00\x0
Console Environment Variables Settings Sections
Pattern exited with code: 0
Evaluation took 0.0010852s
Auto evaluate 14 / 131072
9.47 228.77 MiB / 15.79 GiB

File structures

Known file types leverage **documented data structures** to keep their data in storage. **Inferring** what **data structures** need to be applied to a binary file to parse and consume its content can depend on

- file extension
- the first few bytes of the file in question, a.k.a. "**magic bytes**" or "**magic numbers**" or "**signatures**" (see https://en.wikipedia.org/wiki/List_of_file_signatures)

<u>Digital Forensics perspective</u>: in some cases, files that are considered critical to the investigation **may not have an extension** or **have a wrong one** ⇒ *analyze magic bytes*.

File structures example: JPEG

JPEG Header

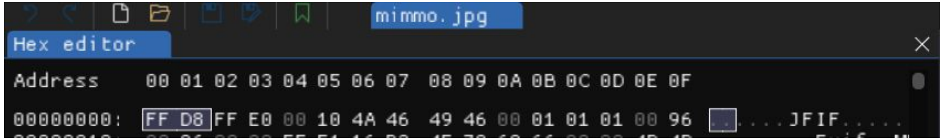
FF D8 …

JPEG Data structures

… E1 1F 8D 3E 67 CD E3 0F …

JPEG Footer

… **FF D9**



Linux facilities

- `file` helps determine the type of a file and its data by running a series of tests on the file data itself, without taking the file extension into account (see https://manpages.debian.org/testing/file/file.1.en.html):
  - filesystem tests, magic tests, and language tests (if a text-type file)
- `binwalk` (`apt install binwalk`): is a tool for searching a given binary image for embedded files and executable code (see https://manpages.ubuntu.com/manpages/trusty/man1/binwalk.1.html):
  - uses the same signatures of the file utility
  - executes **file carving** procedures, i.e., reassembling files from fragments in the absence of filesystem metadata

Base64 encoding

A binary-to-text **encoding** scheme that is designed to carry data stored in **binary formats** across channels that only reliably **support text content**.

- *World Wide Web*: enabling to embed image files or other binary assets inside textual assets such as HTML and CSS files
- *Email*: convert binary data into ASCII characters that can be sent via email attachments, as SMTP was originally designed to transport only 7-bit ASCII characters

Base64: how it works

Base64 encoding takes the original binary data and

- divides it into **tokens** of **3 bytes** (24 bits)
- these 3 bytes are then converted into **4 printable characters** from the ASCII standard (using an alphabet represented with a 6 bits index)
- the ASCII characters used for Base64 are the numbers **0-9**, the **alphabets** 26 **lowercase** and 26 **uppercase** characters plus two extra characters '**+**' and '**/**'
- **= padding** characters might be added to make the last encoded block contain 4 Base64 characters.

Index	Binary	Char	Index	Binary	Char	Index	Binary	Char	Index	Binary	Char
0	000000	A	16	010000	Q	32	100000	g	48	110000	w
1	000001	B	17	010001	R	33	100001	h	49	110001	x
2	000010	C	18	010010	S	34	100010	i	50	110010	y
3	000011	D	19	010011	T	35	100011	j	51	110011	z
4	000100	E	20	010100	U	36	100100	k	52	110100	0
5	000101	F	21	010101	V	37	100101	l	53	110101	1
6	000110	G	22	010110	W	38	100110	m	54	110110	2
7	000111	H	23	010111	X	39	100111	n	55	110111	3
8	001000	I	24	011000	Y	40	101000	o	56	111000	4
9	001001	J	25	011001	Z	41	101001	p	57	111001	5
10	001010	K	26	011010	a	42	101010	q	58	111010	6
11	001011	L	27	011011	b	43	101011	r	59	111011	7
12	001100	M	28	011100	c	44	101100	s	60	111100	8
13	001101	N	29	011101	d	45	101101	t	61	111101	9
14	001110	O	30	011110	e	46	101110	u	62	111110	+
15	001111	P	31	011111	f	47	101111	v	63	111111	/
Padding		=									

Base64 alphabet defined in RFC 4648 §4

Base64 example [1]

Input Data	A	B	C
Input Bits	01000001	01000010	01000011

Bit groups	010000	010100	001001	000011
Mapping	Q	U	J	D

Index	Binary	Char	Index	Binary	Char	Index	Binary	Char	Index	Binary	Char
0	000000	A	16	010000	Q	32	100000	g	48	110000	w
1	000001	B	17	010001	R	33	100001	h	49	110001	x
2	000010	C	18	010010	S	34	100010	i	50	110010	y
3	000011	D	19	010011	T	35	100011	j	51	110011	z
4	000100	E	20	010100	U	36	100100	k	52	110100	0
5	000101	F	21	010101	V	37	100101	l	53	110101	1
6	000110	G	22	010110	W	38	100110	m	54	110110	2
7	000111	H	23	010111	X	39	100111	n	55	110111	3
8	001000	I	24	011000	Y	40	101000	o	56	111000	4
9	001001	J	25	011001	Z	41	101001	p	57	111001	5
10	001010	K	26	011010	a	42	101010	q	58	111010	6
11	001011	L	27	011011	b	43	101011	r	59	111011	7
12	001100	M	28	011100	c	44	101100	s	60	111100	8
13	001101	N	29	011101	d	45	101101	t	61	111101	9
14	001110	O	30	011110	e	46	101110	u	62	111110	+
15	001111	P	31	011111	f	47	101111	v	63	111111	/
Padding		=									

Base64 example [2]

Input Data	A	B	C	E
Input Bits	01000001	01000010	01000011	01000101

Bit groups	010000	010100	001001	000011
Mapping	Q	U	J	D

Bit groups	010001	010000	-	-
Mapping	R	Q	=	=

Index	Binary	Char	Index	Binary	Char
0	000000	A	16	010000	Q
1	000001	B	17	010001	R
2	000010	C	18	010010	S
3	000011	D	19	010011	T
4	000100	E	20	010100	U
5	000101	F	21	010101	V
6	000110	G	22	010110	W
7	000111	H	23	010111	X
8	001000	I	24	011000	Y
9	001001	J	25	011001	Z
10	001010	K	26	011010	a
11	001011	L	27	011011	b
12	001100	M	28	011100	c
13	001101	N	29	011101	d
14	001110	O	30	011110	e
15	001111	P	31	011111	f
Padding		=			

Base64: Linux facilities

- `base64` encode/decode (`-d`) data and print to standard output

  `echo ABCE | base64` gives as output `QUJDRQo=`

Why does it differ from the previous result (`QUJDRQ==`)?

Base64: Linux facilities

- `base64` encode/decode (`-d`) data and print to standard output

  `echo ABCE | base64` gives as output `QUJDRQo=`

Why does it differ from the previous result (`QUJDRQ==`)?

`echo ABCE | xxd`

LF - Line Feed

`00000000: 4142 4345 0a                                ABCE.`

`echo -n ABCE | base64` returns *`QUJDRQ==`*

Exercises

LAB Setup

- Linux or a Virtual Machine with Linux (any distribution)
- Linux command line tools: strings, binwalk, dd, base64, xdd, hexdump …
- A Hex editor (the solutions use ImHex)

Challenge 1: spaghetti

Find the string containing the flag inside

https://github.com/enricorusso/DF_Exs/raw/main/data_organization/spaghetti.png

(the flag format is `flag{...}`)

Challenge 1: spaghetti (solution)

We can use the `strings` command to extract all the strings from the file and search for the ones containing “flag”:

```
strings spaghetti.png  | grep flag

    <rdf:li xml:lang="x-default">flag{hello_strings}</rdf:li>
```

Challenge 2: spaghetti with meatballs

Find the string containing the flag inside

https://github.com/enricorusso/DF_Exs/raw/main/data_organization/spaghetti-with-meatballs.png

(the flag format is `flag{...}`)

Challenge 2: spaghetti with meatballs (solution)

```
strings spaghetti-with-meatballs.png | grep -A 1 flag
     <rdf:li xml:lang="x-default">flag{
}</rdf:li>
```

returns an empty flag.

This is due to the default behavior of strings (see man)

`“[...] -e Select the character encoding of the strings that are to be found [...] s = single-7-bit-byte characters (ASCII, ISO 8859, etc., default), S = single-8-bit-byte characters [...]”`

that searches only for strings encoded with 7-bit. The flag contains UNICODE chars encoded with UTF-8.

```
strings -e S spaghetti-with-meatballs.png | grep flag

     <rdf:li xml:lang="x-default">flag{ᴄɪ̣ᴀᴏ}</rdf:li>
```

Challenge 2: spaghetti with meatballs [2]

```
<rdf:li xml:lang="x-default">flag{ᴄɪ̣ᴀᴏ}</rdf:li>
```

How many bytes does UTF-8 use to encode the flag?

Challenge 2: spaghetti with meatballs [2] (solution)

```
<rdf:li xml:lang="x-default">flag{ᴄᴉᴀᴏ}</rdf:li>
```

How many bytes does UTF-8 use to encode the flag?

```
strings -e S spaghetti-with-meatballs.png | grep  flag | cut -d { -f 2 | cut
-d} -f 1 | hexdump -C
```

```
00000000  e1 b4 84 e1 b4 89 e1 b4 80 e1 b4 8f 0a       |.............|
```

1110 0001 **10**11 0100 **10**11 0100

Byte 1	Byte 2	Byte 3	Byte 4
0xxxxxxx			
110xxxxx	10xxxxxx		
1110xxxx	10xxxxxx	10xxxxxx	

Anatomy of a BMP

A BMP file has the following format (see https://engineering.purdue.edu/ece264/17au/hw/HW15)

Header	54 bytes
Palette (optional)	0 bytes (for 24-bit RGB images)
Image Data	file size - 54 (for 24-bit RGB images)

The BMP is a **little endian** format.

Anatomy of a BMP: header

The header has 54 bytes, which are divided into the following fields.

```
typedef struct { // Total: 54 bytes
  uint16_t  type;             // Magic identifier: 0x4d42
  uint32_t  size;             // File size in bytes
  uint16_t  reserved1;        // Not used
  uint16_t  reserved2;        // Not used
  uint32_t  offset;           // Offset to image data in bytes from beginning of file (54 bytes)
  uint32_t  dib_header_size;  // DIB Header size in bytes (40 bytes)
  int32_t   width_px;         // Width of the image
  int32_t   height_px;        // Height of image
  uint16_t  num_planes;       // Number of color planes
  uint16_t  bits_per_pixel;   // Bits per pixel
  uint32_t  compression;      // Compression type
  uint32_t  image_size_bytes; // Image size in bytes
  int32_t   x_resolution_ppm; // Pixels per meter
  int32_t   y_resolution_ppm; // Pixels per meter
  uint32_t  num_colors;       // Number of colors
  uint32_t  important_colors; // Important colors
} BMPHeader;
```

Challenge 3: Dennis Ritchie headshot

Swap the width and height values of Dennis Ritchie (https://en.wikipedia.org/wiki/Dennis_Ritchie) and save the result as swap.bmp*.

This should look odd and horizontally streaky.

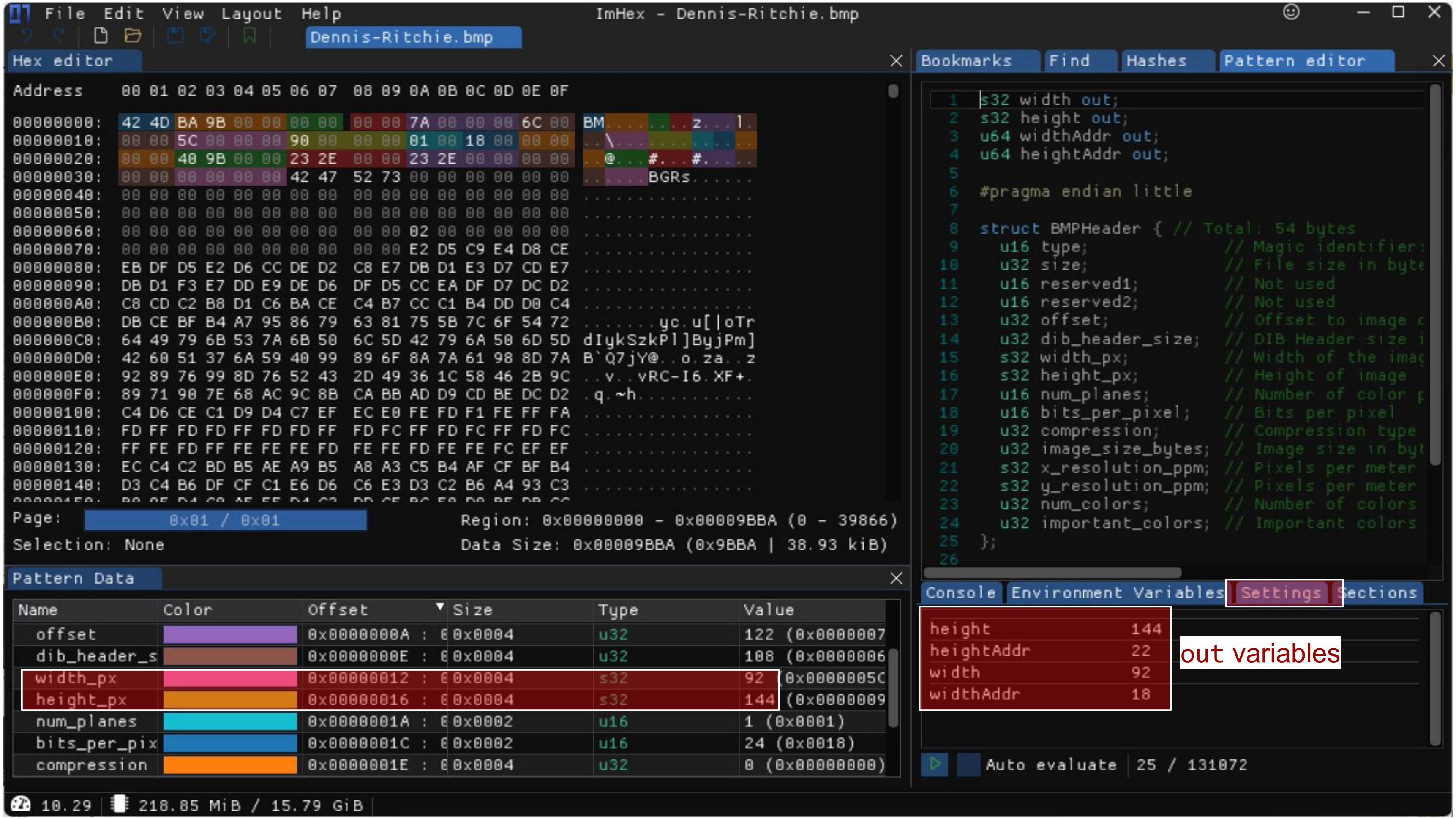
https://github.com/enricorusso/DF_Exs/raw/main/data_organization/Dennis-Ritchie.bmp

*https://www.cs.virginia.edu/luther/CSO1/S2022/lab02-hex-editor.html

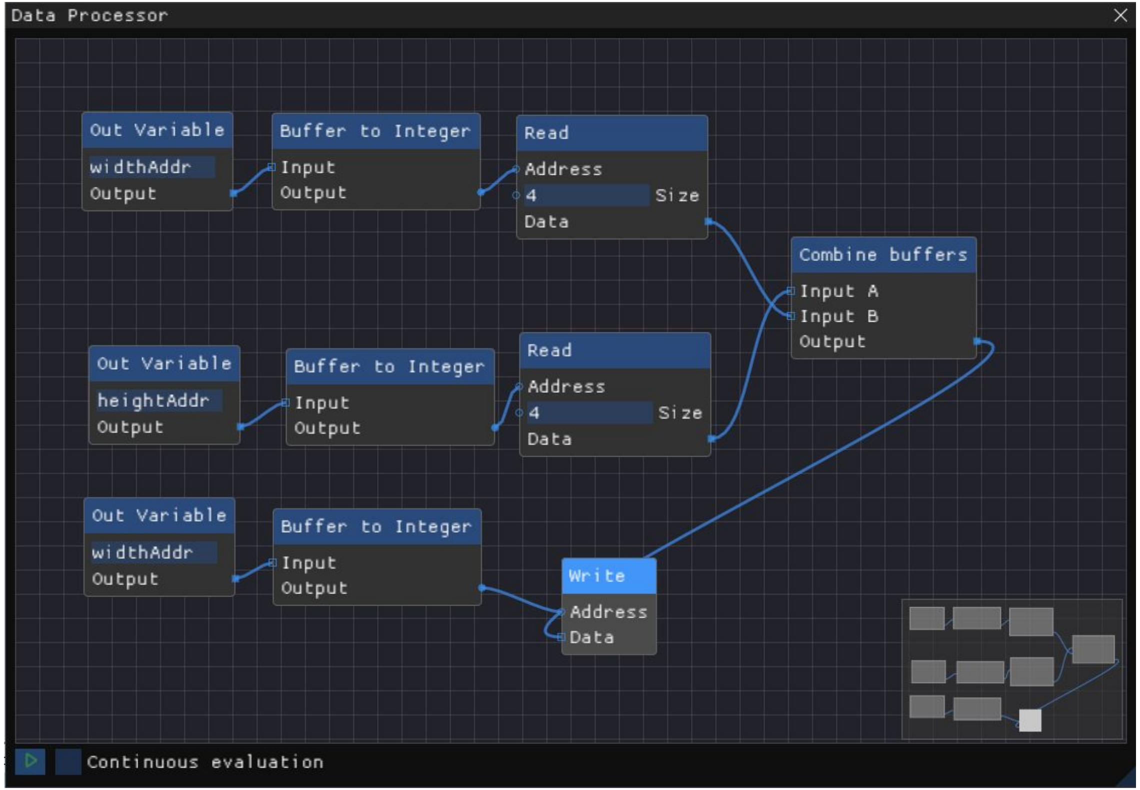
Challenge 3: Dennis Ritchie headshot (solution)

We use the pattern language from ImHex for the BMP Header data structure.

```
s32 width out;
s32 height out;
u64 widthAddr out;
u64 heightAddr out;

struct BMPHeader { // Total: 54 bytes
  u16 type;             // Magic identifier: 0x4d42
  u32 size;             // File size in bytes
  u16 reserved1;        // Not used
  u16 reserved2;        // Not used
  u32 offset;           // Offset to image data in bytes
from beginning of file (54 bytes)
  u32 dib_header_size;  // DIB Header size in bytes (40
bytes)
  s32 width_px;         // Width of the image
  s32 height_px;        // Height of image
  u16 num_planes;       // Number of color planes
  u16 bits_per_pixel;   // Bits per pixel
  u32 compression;      // Compression type
  u32 image_size_bytes; // Image size in bytes
  s32 x_resolution_ppm; // Pixels per meter
  s32 y_resolution_ppm; // Pixels per meter
  u32 num_colors;       // Number of colors
  u32 important_colors; // Important colors
};

BMPHeader header @ 0x00;
width = header.width_px;
height = header.height_px;
widthAddr = addressof(header.width_px);
heightAddr = addressof(header.height_px);
```


File Edit View Layout Help
ImHex - Dennis-Ritchie.bmp
Dennis-Ritchie.bmp
Hex editor
Address 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000: 42 4D BA 9B 00 00 00 00 00 00 7A 00 00 00 6C 00 BM.......z...l.
00000010: 00 00 5C 00 00 00 90 00 00 00 01 00 18 00 00 00 ..\.............
00000020: 00 00 40 9B 00 00 23 2E 00 00 23 2E 00 00 00 00 ..@...#...#.....
00000030: 00 00 00 00 00 00 42 47 52 73 00 00 00 00 00 00 ......BGRs......
00000040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060: 00 00 00 00 00 00 00 00 00 00 02 00 00 00 00 00
00000070: 00 00 00 00 00 00 00 00 00 00 E2 D5 C9 E4 D8 CE
00000080: EB DF D5 E2 D6 CC DE D2 C8 E7 DB D1 E3 D7 CD E7
00000090: DB D1 F3 E7 DD E9 DE D6 DF D5 CC EA DF D7 DC D2
000000A0: C8 CD C2 B8 D1 C6 BA CE C4 B7 CC C1 B4 DD D0 C4
000000B0: DB CE BF B4 A7 95 86 79 63 81 75 5B 7C 6F 54 72 ....yc.u[|oTr
000000C0: 64 49 79 6B 53 7A 6B 50 6C 5D 42 79 6A 50 6D 5D dIykSzkPl]ByjPm]
000000D0: 42 60 51 37 6A 59 40 99 89 6F 8A 7A 61 98 8D 7A B`Q7jY@..o.za..z
000000E0: 92 89 76 99 8D 76 52 43 2D 49 36 1C 58 46 2B 9C ..v..vRC-I6.XF+.
000000F0: 89 71 90 7E 68 AC 9C 8B CA BB AD D9 CD BE DC D2 .q.~h...........
00000100: C4 D6 CE C1 D9 D4 C7 EF EC E0 FE FD F1 FE FF FA
00000110: FD FF FD FD FF FD FD FF FD FC FF FD FC FF FD FC
00000120: FF FE FD FF FE FE FE FD FE FE FD FE FE FC EF EF
00000130: EC C4 C2 BD B5 AE A9 B5 A8 A3 C5 B4 AF CF BF B4
00000140: D3 C4 B6 DF CF C1 E6 D6 C6 E3 D3 C2 B6 A4 93 C3
Page: 0x01 / 0x01
Region: 0x00000000 - 0x00009BBA (0 - 39866)
Selection: None
Data Size: 0x00009BBA (0x9BBA | 38.93 kiB)
Bookmarks Find Hashes Pattern editor
s32 width out;
s32 height out;
u64 widthAddr out;
u64 heightAddr out;
#pragma endian little
struct BMPHeader { // Total: 54 bytes
u16 type; // Magic identifier:
u32 size; // File size in byte
u16 reserved1; // Not used
u16 reserved2; // Not used
u32 offset; // Offset to image c
u32 dib_header_size; // DIB Header size i
s32 width_px; // Width of the imag
s32 height_px; // Height of image
u16 num_planes; // Number of color p
u16 bits_per_pixel; // Bits per pixel
u32 compression; // Compression type
u32 image_size_bytes; // Image size in byt
s32 x_resolution_ppm; // Pixels per meter
s32 y_resolution_ppm; // Pixels per meter
u32 num_colors; // Number of colors
u32 important_colors; // Important colors
};
Pattern Data
Name Color Offset Size Type Value
offset 0x0000000A 0x0004 u32 122 (0x0000007
dib_header_s 0x0000000E 0x0004 u32 108 (0x0000006
width_px 0x00000012 0x0004 s32 92 (0x0000005C
height_px 0x00000016 0x0004 s32 144 (0x0000009
num_planes 0x0000001A 0x0002 u16 1 (0x0001)
bits_per_pix 0x0000001C 0x0002 u16 24 (0x0018)
compression 0x0000001E 0x0004 u32 0 (0x00000000)
Console Environment Variables Settings Sections
height 144
heightAddr 22
width 92
widthAddr 18
out variables
Auto evaluate 25 / 131072
10.29 218.85 MiB / 15.79 GiB

Challenge 3: Dennis Ritchie headshot (solution with DP)



swap.bmp

Anatomy of a JPEG

JPEG is a commonly used method of lossy compression for digital images.

- uses different data structures that are headed by common tags (namely, markets or segments) followed by size, and then the specific data
- all tags start with the value `0xFF`. If the value `0xFF` is ever needed, it must be escaped by immediately following it with `0x00` (byte stuffing)

Magic number

```
xxd -g 1 -u  hideme_master.jpg | grep --color=always -C999 FF | head -12
00000000: FF D8 FF E0 00 10 4A 46 49 46 00 01 01 01 00 78  ......JFIF.....x
00000010: 00 78 00 00 FF DB 00 43 00 02 01 01 02 01 01 02  .x.....C........
00000020: 02 02 02 02 02 02 02 03 05 03 03 03 03 03 06 04  ................
00000030: 04 03 05 07 06 07 07 07 06 07 07 08 09 0B 09 08  ................
00000040: 08 0A 08 07 07 0A 0D 0A 0A 0B 0C 0C 0C 0C 07 09  ................
00000050: 0E 0F 0D 0C 0E 0B 0C 0C 0C FF DB 00 43 01 02 02  .............C...
00000060: 02 03 03 03 06 03 03 06 0C 08 07 08 0C 0C 0C 0C  ................
00000070: 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C  ................
00000080: 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C  ................
00000090: 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C FF C0  ................
000000a0: 00 11 08 01 72 01 B8 03 01 22 00 02 11 01 03 11  ....r...."......
000000b0: 01 FF C4 00 1F 00 00 01 05 01 01 01 01 01 01 00  ................
```

JPEG segments

Information on some JPEG segments. The data is always **big endian**.

TLA	Name	HEX	Size	Required	Notes
SOI	start of image	`0xFF 0xD8`	This tag does not have a size	Yes	This tag must be the first one in the file.
SOF0	start of frame (baseline Discrete Cosine Transform - DCT)	`0xFF 0xC0`	Variable size. Typically `0x00 0x11` (17 bytes)	Yes (but see notes)	SOF0 can be replaced with SOF1 (`0xFFC1`, extended sequential DCT), SOF2 (`0xFFC2`, progressive DCT), etc.
…					
EOI	end of image	`0xFF 0xD9`	This tag does not have a size	Yes	This tag must be the last one in the image

(see https://www.ccoderun.ca/programming/2017-01-31_jpeg/)

JPEG segments: SOF0

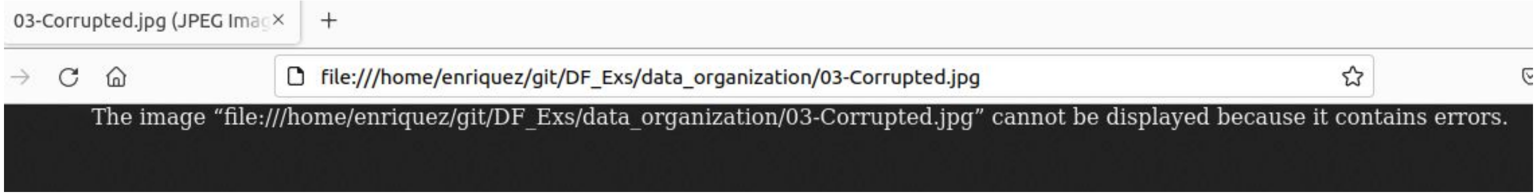
SOF0 (start of frame) - 0xFFC0

```
0xFF, 0xC0,                 // SOF0 segment
0x00, 0x11,                 // length of segment depends on the number of
components
0x08,                       // bits per pixel
0x00, 0x95,                 // image height
0x00, 0xE3,                 // image width
0x03,                       // number of components (should be 1 or 3)
0x01, 0x22, 0x00,           // 0x01=Y component, 0x22=sampling factor,
quantization table number
0x02, 0x11, 0x01,           // 0x02=Cb component, ...
0x03, 0x11, 0x01            // 0x03=Cr component, ...
```

Challenge 4: Corrupted File

During a transmission, one of our files got corrupted. Take a look and see if you can do something about it.

the flag format is ISC{…}*



https://github.com/enricorusso/DF_Exs/raw/main/data_organization/03-Corrupted.jpg

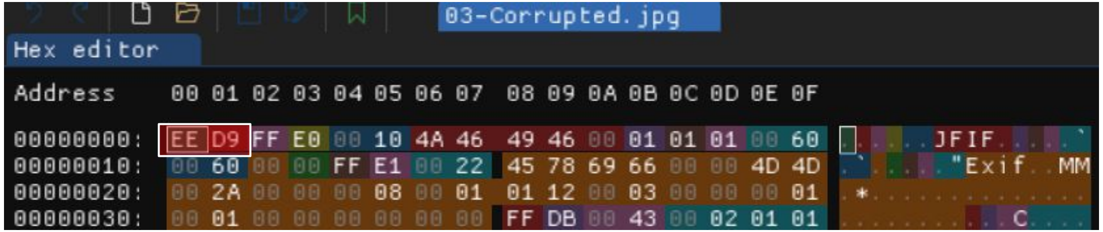
*Introduction to Computer Security course labs (https://ocw.cs.pub.ro/courses/isc/labs/09)

Challenge 4: Corrupted File (solution)

Although the file extension is useful for figuring out which data structures to apply, it is necessary to refer to the magic number.

`file 03-Corrupted.jpg` returns `03-Corrupted.jpg: data`

The magic number is probably wrong.



ISC{no_more_ideas_for_flags}

According to SOI segment, a JPEG file starts with `0xFF 0xD8`

Challenge 5: steganography

What the people will see

HIDE ME HIDE ME HIDE ME HIDE ME HIDE ME

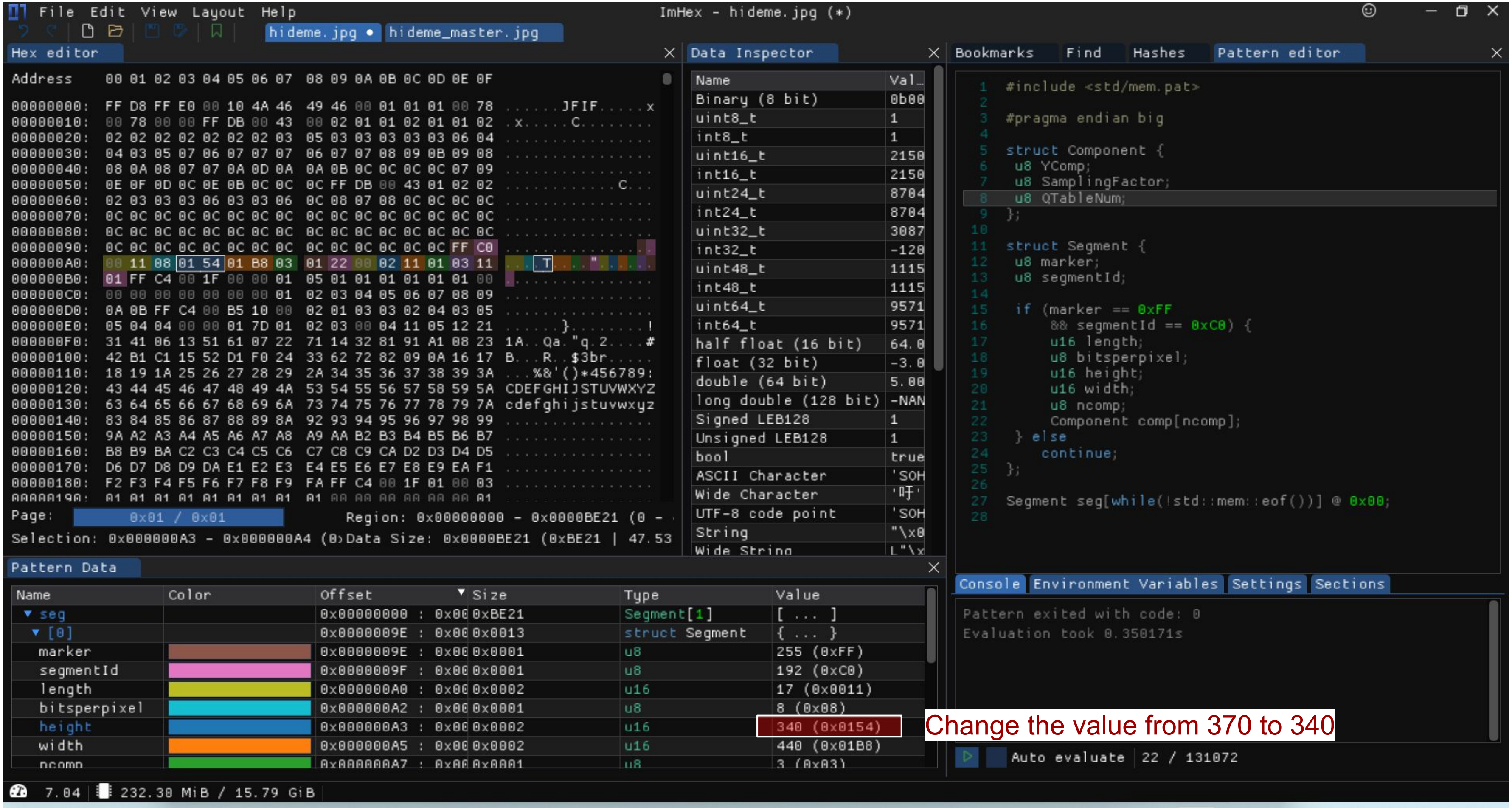
What you need to hide

https://github.com/enricorusso/DF_Exs/raw/main/data_organization/hideme_master.jpg

Challenge 5: steganography (solution)

We reduce the height of the image by changing the data structure in the SOF0 segment. We use the pattern language from ImHex to find it.

```
#include <std/mem.pat>

#pragma endian big

struct Component {
 u8 YComp;
 u8 SamplingFactor;
 u8 QTableNum;
};

struct Segment {
 u8 marker;
 u8 segmentId;
if (marker == 0xFF
     && segmentId == 0xC0) {
     u16 length;
     u8 bitsperpixel;
     u16 height;
     u16 width;
     u8 ncomp;
     Component comp[ncomp];
 } else
    continue;
};

Segment seg[while(!std::mem::eof())] @ 0x00;
```

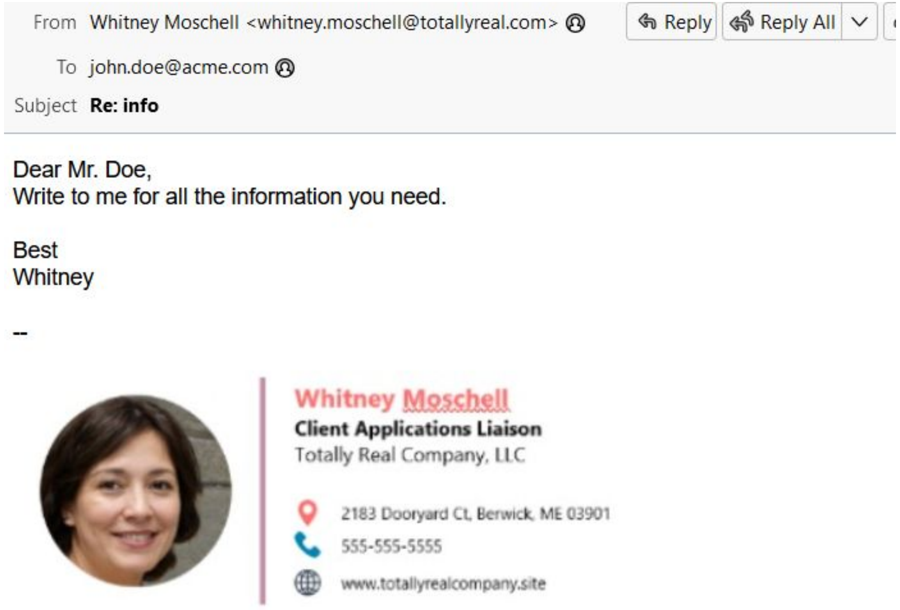

File Edit View Layout Help
ImHex - hideme.jpg (*)
hideme.jpg
hideme_master.jpg
Hex editor
Data Inspector
Bookmarks
Find
Hashes
Pattern editor
#include <std/mem.pat>
#pragma endian big
struct Component {
u8 YComp;
u8 SamplingFactor;
u8 QTableNum;
};
struct Segment {
u8 marker;
u8 segmentId;
if (marker == 0xFF
&& segmentId == 0xC0) {
u16 length;
u8 bitsperpixel;
u16 height;
u16 width;
u8 ncomp;
Component comp[ncomp];
} else
continue;
};
Segment seg[while(!std::mem::eof())] @ 0x00;
Page: 0x01 / 0x01
Region: 0x00000000 - 0x0000BE21
Selection: 0x000000A3 - 0x000000A4
Data Size: 0x0000BE21 (0xBE21 | 47.53
Pattern Data
Name Color Offset Size Type Value
seg 0x00000000 0xBE21 Segment[1] [...]
[0] 0x0000009E 0x0013 struct Segment { ... }
marker 0x0000009E 0x0001 u8 255 (0xFF)
segmentId 0x0000009F 0x0001 u8 192 (0xC0)
length 0x000000A0 0x0002 u16 17 (0x0011)
bitsperpixel 0x000000A2 0x0001 u8 8 (0x08)
height 0x000000A3 0x0002 u16 340 (0x0154)
width 0x000000A5 0x0002 u16 440 (0x01B8)
ncomp 0x000000A7 0x0001 u8 3 (0x03)
Console
Environment Variables
Settings
Sections
Pattern exited with code: 0
Evaluation took 0.350171s
Auto evaluate 22 / 131072
7.04
232.30 MiB / 15.79 GiB
Change the value from 370 to 340

Challenge 6: data exfiltration

Someone tells us that Ms. Whitney is exfiltrating sensitive data from our company.

Find hidden data in her email.

(Sysadmin noticed that the file size of the signature image has changed)



https://github.com/enricorusso/DF_Exs/raw/main/data_organization/mail.eml

Challenge 6: data exfiltration (solution)

The filename "`signature-new.jpg`" related to the signature image suggests that it has been modified.

We can analyze the jpg file by extracting the base64 data enclosed between the boundaries identified with "--=_bcJPkVe2nbj_lquOawM3OTf--".

```
grep -A100000 _bcJPkVe2nbj_lquOawM3OTf  mail.eml | grep -A100000
-i "content-type: image/jpeg" | tail -n +6 | cut -d- -f 1 | base64
-d > signature-new.jpg
```

`signature-new.jpg` is the name of the extracted JPEG file.

Challenge 6: data exfiltration (solution)

Analyzing the signature with `binwalk` shows that the image file contains also a ZIP archive

```
binwalk -e signature-new.jpg
```

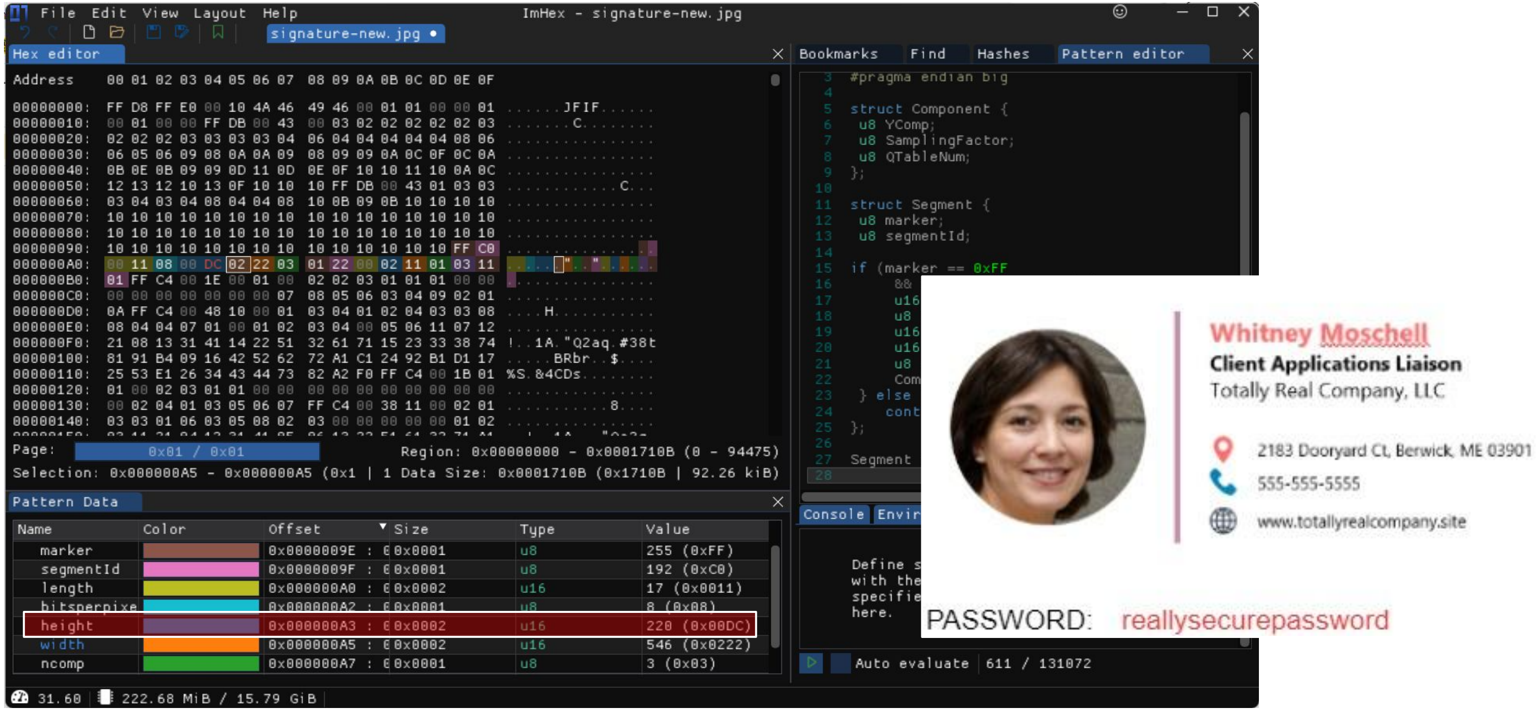
```
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             JPEG image data, JFIF standard 1.01
19580         0x4C7C          Zip archive data, encrypted at least v2.0 to extract, compressed size:
74701, uncompressed size: 151156, name: dummy_data.csv
94453         0x170F5         End of Zip archive, footer length: 22
```

-e option extracts data in the folder named “`_signature-new.jpg.extracted`”.

The ZIP file is password protected!

Challenge 6: data exfiltration (solution)

We change the height of the image from 197 (0x5C) px to 220 (0xDC) px.



Challenge 6: data exfiltration (solution)

We unzip the archive using the password **`reallysecurepassword`**

```
unzip 4C7C.zip
Archive:  4C7C.zip
[4C7C.zip] dummy_data.csv password:
replace dummy_data.csv? [y]es, [n]o, [A]ll, [N]one, [r]ename: A
  inflating: dummy_data.csv
```

The csv file contains the exfiltrated data.

Challenge 7: hidden file

There is something wrong with the size of this image. Is there anything else there?

the flag format is ISC{…}*

https://github.com/enricorusso/DF_Exs/raw/main/data_organization/05-Idea.jpg

*Introduction to Computer Security course labs (https://ocw.cs.pub.ro/courses/isc/labs/09)

Challenge 7: hidden file (solution)

We can use `binwalk` but ”`-e`” option is buggy sometimes..

```
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             JPEG image data, JFIF standard 1.01
33519         0x82EF          7-zip archive data, version 0.4
```

For extracting the 7-zip archive we can:

1. use `binwalk --dd='.*' 05-idea.jpg` to extract all of the possible files by their offset
2. use `dd` (convert and copy a file) directly: `dd bs=1 if=05-Idea.jpg of=file.7z skip=33519 count=9999999`

The 7z archive (`7z x file.7x`) contains the image `flag4.jpg` with the flag

ISC{fileception_is_real}