

universidade
de aveiro

Computer Systems Forensic Analysis AFSC

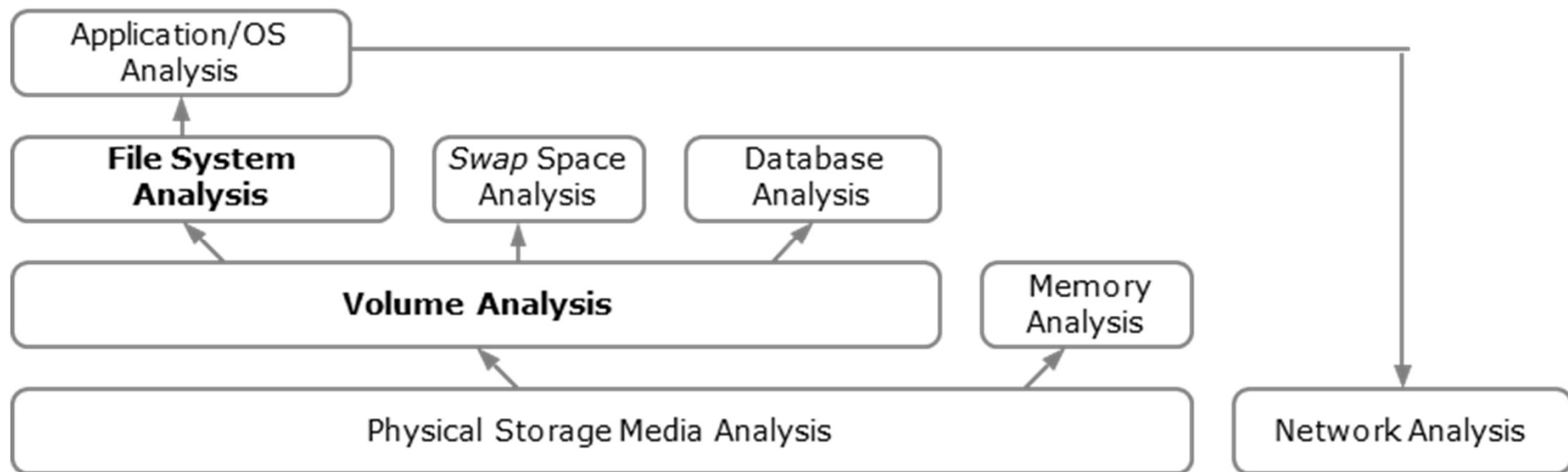
Data Organization

Artur Varanda

School Year 2021-2022

Data Organization

Data storage systems have several layers



Data Organization

Layer 1 – requires specialized laboratories

communication devices: Ethernet, 3G, UMTS, ...

physical store mediums: hard disks, memory chips, CD-ROMs, ...

Layer 2 – reading logical data (streams of 0s and 1s)

volatile memory (RAM) – data typically organized by processes

non-volatile storage – data typically organized into volumes

typically organized into volumes (partitions, RAID arrays, ...)

analyze data at the volume level to find possible hidden data

Data Organization

Layer 3

file system (most common content)

temporary space: swap space in Linux or pagefile in Windows

direct database (without traditional file system), such as *Google file system*, ...

Layer 4

Operating systems (Windows, Mac OSX, Linux, Android, iOS, ...)

Applications (operating system dependent)

Focus of this course

File System Analysis

File system analysis:

collection of data structures that allow an application to create, read, and write files

Analyse file system to:

- find files

- recover deleted files

- find hidden data

the result can be:

- file content

- data fragments

- metadata associated with files

File system

organizes data inside a volume

associate file names to file content

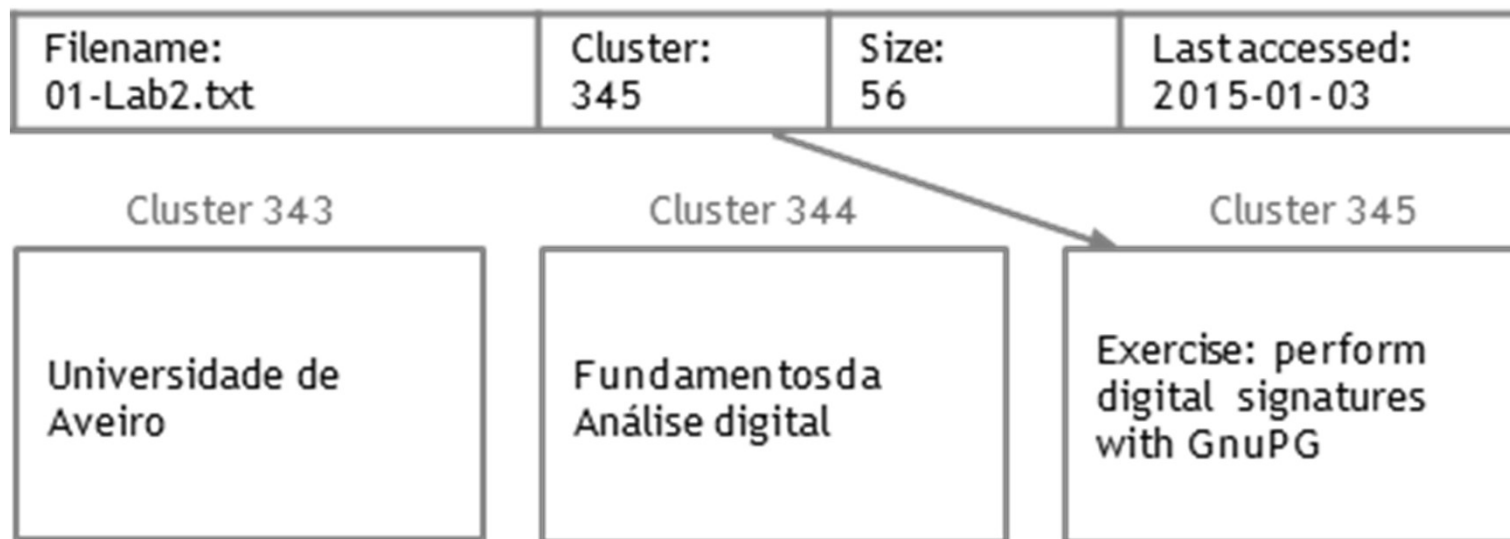
essential data: file names and content location

trustful data – however content may be invalid, *e. g.* deleted files

non essential data: last access time – even if it is wrong the file content still is valid

we may not be able to trust non essential data, *e. g.* system time may be inaccurate, the user may have changed the time, *etc*

we should try to find additional data sources to support an incident hypothesis



File content analysis:

data structure depends on the application or OS that created the file analysis tools may vary accordingly to the application that created the file

HTML is data structure differs from .jpeg

analysis of configuration files is important to determine what programs were running

Data analysis process from the physical level to the application level:



NUMERICAL REPRESENTATION

Numbers can be represented in several ways:

decimal – human system (10 fingers)

10 symbols: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

binary – computational representation (2 levels of voltage)

2 symbols: [0, 1]

hexadecimal – compact representation of binary numbers

16 symbols: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F]

there are more numerical systems *e. g.* **octal**

Each symbol has a value depending on its position, *e. g.* $35\,812_d$:

$$3 \times 10^4 + 5 \times 10^3 + 8 \times 10^2 + 1 \times 10^1 + 2 \times 10^0 =$$

$$3 \times 10\,000 + 5 \times 1\,000 + 8 \times 100 + 1 \times 10 + 2 \times 1 =$$

$$30\,000 + 5\,000 + 800 + 10 + 2 = 35\,812$$

most significant symbol \rightarrow leftmost value: 3

less significant symbol \rightarrow rightmost value: 2

BINARY NUMBERS

Example: Convert $1001\ 0011_b$ to decimal:

$$1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 =$$

$$1 \times 128 + 0 \times 64 + 0 \times 32 + 1 \times 16 + 0 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 =$$

$$128 + 0 + 0 + 16 + 0 + 0 + 2 + 1 = 147_d$$

which is the **most** significant digit? which is the **less** significant digit?

Generic formula to convert from any base system to decimal:

$$s_p \times b^p + \dots + s_1 \times b^1 + s_0 \times b^0$$

s – symbol value

b – original base (binary, octal, hexadecimal, ...)

p – symbol position, begins at zero and increases from right to left

Convert to decimal $8BE4_h$, which can also be represented as $0x8BE4$

$$A = 10, B = 11, C = 12, D = 13, E = 14, F = 15$$

$$8 \times 16^3 + 11 \times 16^2 + 14 \times 16^1 + 4 \times 16^0 =$$

$$8 \times 4096 + 11 \times 256 + 14 \times 16 + 4 \times 1 =$$

$$32768 + 2816 + 224 + 4 = 35812_d$$

How do we convert from binary to hexadecimal?

CONVERTING BETWEEN BINARY AND HEXADECIMAL

Conversion Table

Decimal	Binary	Hexadecimal	Decimal	Binary	Hexadecimal
0	0000	0x0	8	1000	0x8
1	0001	0x1	9	1001	0x9
2	0010	0x2	10	1010	0xA
3	0011	0x3	11	1011	0xB
4	0100	0x4	12	1100	0xC
5	0101	0x5	13	1101	0xD
6	0110	0x6	14	1110	0xE
7	0111	0x7	15	1111	0xF

CONVERTING BETWEEN BINARY AND HEXADECIMAL

Example:

$$1001\ 0011_b = 0x93$$

direct conversion:

$$1001_b = 0x9$$

$$0011_b = 0x3$$

1 Byte = 8 bits = 2 hexadecimal digits

hexadecimal \rightarrow binary compact representation

FLOATING POINT NUMBERS

Floating point number

- format IEEE 754 standard
- exponent in excess allows direct comparisons of floating-point numbers
- mantissa (or significand):
 - normalized → the binary digit 1 to the left of the comma is omitted
 - in additions and subtractions is denormalized, but there is a gradual loss of accuracy
- single precision: 32 bits — exponent in excess of 127 → $[2^{-126}, 2^{+127}]$
- double precision : 64 bits — exponent in excess of 1023 → $[2^{-1022}, 2^{+1023}]$
- conversion tools: <http://www.h-schmidt.net/FloatConverter/IEEE754.html> ; <https://www.binaryconvert.com>

Signal	Exponent	Mantissa (or significand)
1 bit	8 bits	23 bits

Signal	Exponent	Mantissa (or significand)
1 bit	11 bits	52 bits

ENDIANNESS – STORAGE ORDER

Binary data unit:

1 Byte (B) = 8 bits (b): $2^8 = 256$ possible values to store large numbers it is necessary to group bytes

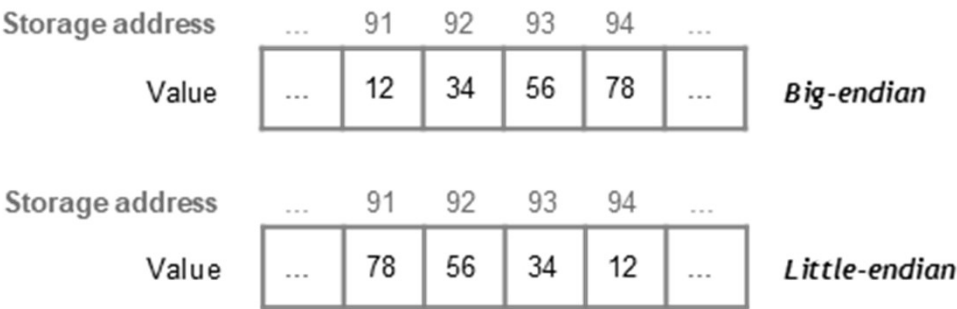
typically 2, 4 or 8 bytes (16, 32 or 64 bits)

Problem: systems differ in how they store multi-byte values

big-endian: place the **most** significant byte in the 1st (or lower) storage address

little-endian: place the **less** significant byte in the 1st (or lower) storage address

Example: 0x12 34 56 78



Big-endian processors

- SPARC, PowerPC, MIPS, Motorola 68k, Alpha, ...

Little-endian processors

- it seems that there are some optimization advantages in pipelined architectures
- z80, VAX, x86, x86-64, amd64, ...

Programmable Big/Little-endian

- ARM, ...

Data networks – *network order*

- IP: **Big-endian**, but there are some exceptions
- very important to guarantee systems interoperability

More info: <http://en.wikipedia.org/wiki/Endianness>

CHARACTER ENCODING – ASCII

Advantages:

- is the simplest way to encode the characters
- doesn't have *endianness* problems – uses 1 byte at a time original version uses only 7 bits – 128 different characters
 - ✓ ASCII table: <https://www.asciitable.com/asciifull.gif>
- it takes up less storage space than unicode

Disadvantages:

- very limited capacity to represent non-English characters
- there are several extended versions – 8 bits (ISO 8859)
 - ✓ the best known is *Latin-1 (ISO 8859-1)* (<http://cs.stanford.edu/people/miles/iso8859.html>)

Example

Address	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Hex.	...	43	61	70	69	74	E3	6F	20	41	6D	E9	72	69	63	61	...
Text		C	a	p	i	t	ã	o		A	m	é	r	i	c	a	

Advantages:

- the latest version has more than 137 000 characters
- covers 146 modern and historic scripts, as well as multiple symbol sets

Disadvantages:

- there are several implementations
 - ✓ UTF-8: compatible with ASCII, has variable length from 1 to 4 bytes and is used in *nix systems, www, HTML, *etc*
 - not subject to endianness problems – it reads 1 byte at a time
 - ✓ UTF-16: has variable length from 2 to 4 bytes, is used in Windows, Mac OS, Java, .Net, KDE, *etc*
 - ✓ UTF-32: fixed length of 4 bytes
- more complex processing

Example: UTF-8 vs Latin-1

Address	10	11	12	13	14	15
Hex.	...	4F	6C	C3	A1	...
UTF-8		O	l	á		

Address	10	11	12	13	14	15
Hex.	...	4F	6C	E1
Latin-1		O	l	á		

Data structures:

- specifies how data is placed
- it is like a map
- data structure itself is not recorded

Data structure:

```
typedef struct{
    char name[32];           // 32 bytes
    char postcode[16];       // 16 bytes
    uint32_t partner_num;    // 4 bytes
    float quota;             // 4 bytes
    char empty[8];           // 8 bytes to lineup the structure to multiples of 16 bytes
};                           // Total = 64 bytes
```

Data in hexadecimal:

0000000:	4d69 6775 656c 2046 7261 6465 0000 0000	Miguel Frade....
0000010:	0000 0000 0000 0000 0000 0000 0000 0000
0000020:	3234 3131 2d39 3031 0000 0000 0000 0000	2411-901.....
0000030:	b300 0d00 c976 9e3f 0000 0000 0000 0000v.?.....

What is the partner number (in hex.)? Is it 0xb300 0d00 or 0x000d 00b3?

What is the quota value (in hex.)? Is it 0xc976 9e3f or 0x3f9e 76c9?

DATA STRUCTURES – EXAMPLE

Original source code

```
strcpy(partners[0].name, "Miguel Frade");  
strcpy(partners[0].postalcode, "2411-901");  
partners[0].partner_num=852147;  
partners[0].quota=1.238;
```

Question

Is this system *Little-endian* or *Big-endian*?

if it is *Big-endian*, then:

partner number: 0xb300 0d00 = 3 003 124 992; quota: 0xc976 9e3f = -1 010 147, 94

if it is *Little-endian*, then:

partner number: 0x000d 00b3 = 852 147; quota: 0x3f9e 76c9 = 1, 238

There are many different ways to storage date and time:

- as a string:

- easy to read by humans,
- but hard for computers' operations, for example to compare dates in a database
- many different representations and some times language dependent, *e. g.*:
 - Wednesday, January 9, 11:13:48 UTC 2019
 - 2018-12-23 08:23:55
 - 23-12-2018 08:23:55 (PT)
 - 12-23-2018 08:23:55 (US)
 - ...

- as binary represented with numbers or hexadecimal

- difficult to read by humans, but easier for computers' operations
- unfortunately, not all software uses the same representation

How time is counted:

Unix time = POSIX time = UNIX Epoch time (<https://www.epochconverter.com>)

- number of elapsed seconds since 1970-01-01 00:00:00
- Unix, Linux, **Firefox**, Java, JavaScript, Perl, PHP, Python, Ruby, Tcl, *etc*

400-year Gregorian calendar cycle

- number of microseconds since 1601-01-01 00:00:00
- **Google Chrome**, Windows 32 and 64 bits, NTFS, Cobol, *etc*

Examples of date and time storage formats:

Software	Representation	Example	Software	Representation	Example
Win 64 bits BE	16 Hex chars	01 CC A6 3C 91 B9 72 00	Firefox SQLite	16 digits	1 321 653 236 000 000
Win 64 bits LE	16 Hex chars	00 72 B9 91 3C A6 CC 01	Chrome SQLite	17 digits	12 966 126 836 000 000
Unix numeric	10 digits	1 321 653 236	Cookie hi:low	18:19 digits	3 923 586 186:30 188 999
Unix numeric m. seconds	13 digits	1 321 653 236 000 000	DOS wFAT	8 Hex chars	F0 48 64 40
Unix 32 bits BE	8 Hex chars	4E C6 D3 F4	GSM	14 digits	99 309 251 619 580
Unix 32 bits LE	8 Hex chars	F4 D3 C6 4E	Samsung Swift	8 Hex chars	E7 8C 94 7D
HFS+ 32 bits BE	8 Hex chars	CA EC 84 74	Nokia Series 40	14 Hex chars	07 D9 04 11 13 27 09
HFS+ 32 bits LE	8 Hex chars	74 84 EC CA			
Mac Absolute time	9 digits	343 346 036			

Chrome SQLite/1000000 = Unix time + 11644473600 sec

(1970-01-01 00:00:00) - (1601-01-01 00:00:00) = 11644473600 sec

Date and time convert tools

MFT Stampede <https://mft-stampede.software.informer.com/download/>

- ✓ Windows GUI tool, very easy to use

Nirsoft tool *BrowsingHistoryView* https://www.nirsoft.net/utils/browsing_history_view.html

- ✓ supports browsing history of Internet Explorer, Mozilla Firefox, Google Chrome, and Safari

SQL – cool for automation of tasks <https://sqlitebrowser.org/dl/>

- Firefox SQLite database overview of the visited sites:
 - Ubuntu -> [user_home_directory]/.mozilla/firefox/xxxxxxxx.default/places.sqlite
 - Windows -> [user_home_directory]\AppData\Roaming\Mozilla\Firefox\Profiles\xxxxxxxx.default\places.sqlite

```
SELECT datetime(moz_historyvisits.visit_date/1000000, 'unixepoch', 'localtime') AS Date_Time, moz_places.url
FROM moz_places, moz_historyvisits
WHERE moz_places.id = moz_historyvisits.place_id
ORDER BY Date_Time ASC
```

- Google Chrome SQLite database overview of the visited sites:

Ubuntu -> [user_home_directory]/.config/google-chrome/Default/databases

Windows -> [user_home_directory]\AppData\Local\Google\Chrome\User Data\Default\History

```
SELECT datetime(((visits.visit_time/1000000)-11644473600), 'unixepoch', 'localtime') AS Date_Time, urls.url, urls.title
FROM urls, visits
WHERE urls.id = visits.url
ORDER BY Date_Time ASC
```

Binary-to-text encoding

- is encoding of data in plain text, or in other words it is an encoding of binary data in a sequence of printable characters
- the encoding is necessary for transmission of data when the channel does not allow binary data (*e. g.* email)
- encoding inflates the original data size, the inflate rate depends on the used technique encoding is a reversible operation
- it can also be applied to plain text

Not to be confused with encrypting, **it's not** encryption:

- encryption requires a key, usually secret
- encoding doesn't depend on a key

COMMON ENCODING TECHNIQUES

Common encoding techniques:

hexadecimal (also known as base16)

- ✓ used chars: [0..9] and [A..F] (or [a..f])
- ✓ hash values (MD5, SHA256, etc) are usually displayed in hexadecimal
- ✓ example: "Hello World" → 48656c6c6f20776f726c64

base64

- ✓ used chars: [A..Z], [a..z], [0..9], and [+, /]
- ✓ base64 string size must be a multiple of 4, so char = can be used at the end as padding
- ✓ used on: email servers (MIME), OpenPGP, etc
- ✓ example: "Hello World" → SGVsbG8gV29ybGQ=
<https://cryptii.com/pipes/binary-to-base64> <https://www.browserling.com/tools/base64-encode>

base58:

- ✓ similar to base64, but modified to avoid both non-alphanumeric characters and letters which might look ambiguous when printed
- ✓ used on bitcoins,
- ✓ example: bitcoin public key 1ZNz2KdM8epACBA5bjgKQbRyaGcDt3XV2
- ✓ bitcoin private key: Ky1ZcCSMziFtdxfDEjANw3PZUZQQljh6hKpX1CinVtJscnAFnvcn
<https://learnmeabitcoin.com/technical/base58> https://en.bitcoin.it/wiki/Base58Check_encoding

Exercises

Please do the following exercises:

Lab 1 – Endianness, read binary file

Lab 2 – Identify different character encodings

Lab 3 – Character encoding conversions

ANY QUESTIONS?

