howest
university of applied sciences

Practical Reverse Engineering and Malware Analysis (PREMA)

Lecture 1
2025-2026

howest
university of applied sciences

Recap & terminology

Recap bit/byte/hex

- What is a bit?
- What is a byte?
- How many hexadecimal values are needed to represent a byte?
- How many bits are needed to represent a hexadecimal value?

- A8 B7 D3
  - How many bytes?
  - How many bits?

howest
university of applied sciences

From site/app/game to bit

How does it all work (top-bottom)

- Human-level product, the end result we use today
  - Website, software, mobile phone app, video game…
- These are typically build using frameworks, engines, libraries, SDKs…
- Which are created in programming languages
  - Some are considered “high”, others “low”
- To go from source code → to programs we need compilers, linkers, bundlers
- When we have programs, they need to get executed
  - Operating System, Runtime environments, CPU, registers, Memory
- These are created by logic gates on hardware (transistors, capacitors)
- The hardware is created using real physical elements that use currents to represent bits
- Electricity

We could dive into

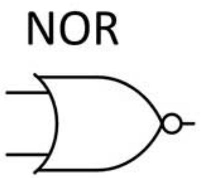
- Analysing JavaScript frameworks
- Learn about all the logical gates in detail to create other gates (example later)
- Learn how we use logical gates to store data (=have memory)
- Learn about transistors
- Learn about the physics of transistors and the elements like why Silicon is an interesting element for chips
- Learn about electronics and electricity

→ We stick to the internals of computer science, as this is where cybersecurity lies

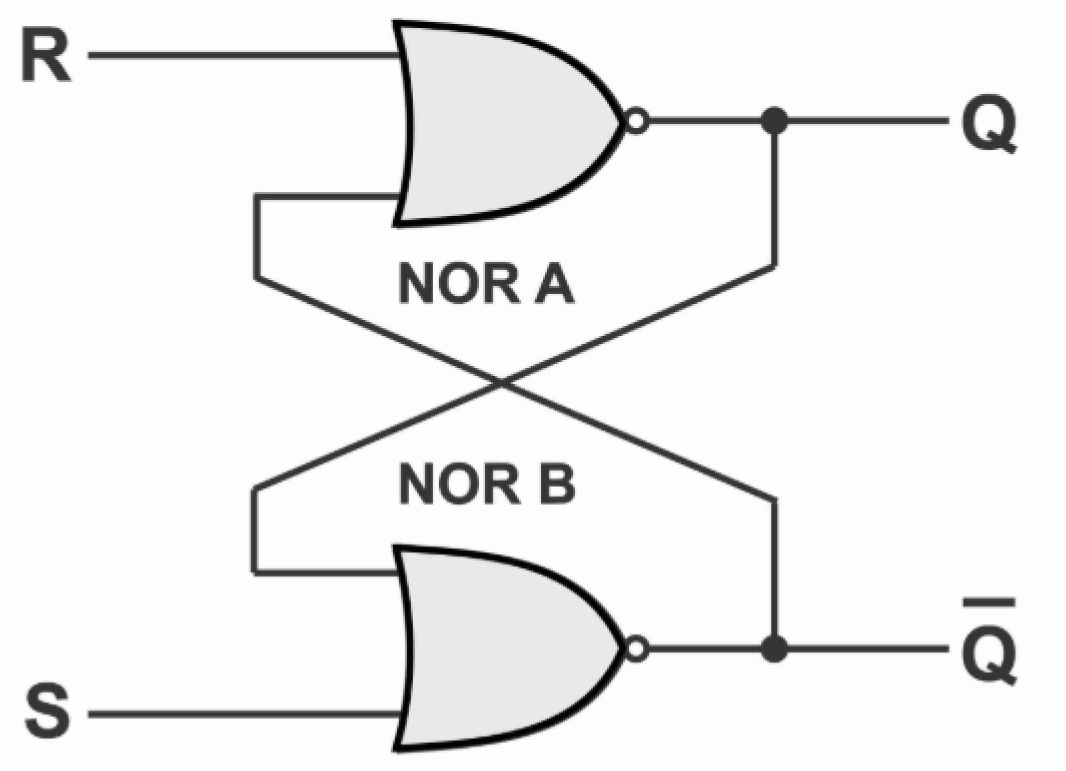
How does it all work (top-bottom)

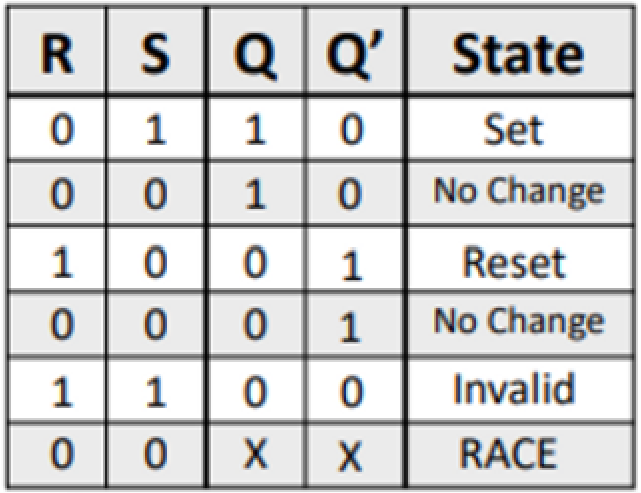
- Human-level product, the end result we use today
  - Website, software, mobile phone app, video game…
- These are typically build using frameworks, engines, libraries, SDKs…
- Which are created in programming languages
  - Some are considered “high”, others “low”
- To go from source code → to programs we need compilers, linkers, bundlers
- When we have programs, they need to get executed
  - Operating System, Runtime environments, CPU, registers, Memory
- These are created by logic gates on hardware (transistors, capacitors)
- The hardware is created using real physical elements that use currents to represent bits
- Electricity

Small intermezzo (no exam material)



A	B	Output
0	0	1
1	0	0
0	1	0
1	1	0



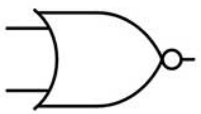


R	S	Q	Q’	State
0	1	1	0	Set
0	0	1	0	No Change
1	0	0	1	Reset
0	0	0	1	No Change
1	1	0	0	Invalid
0	0	X	X	RACE

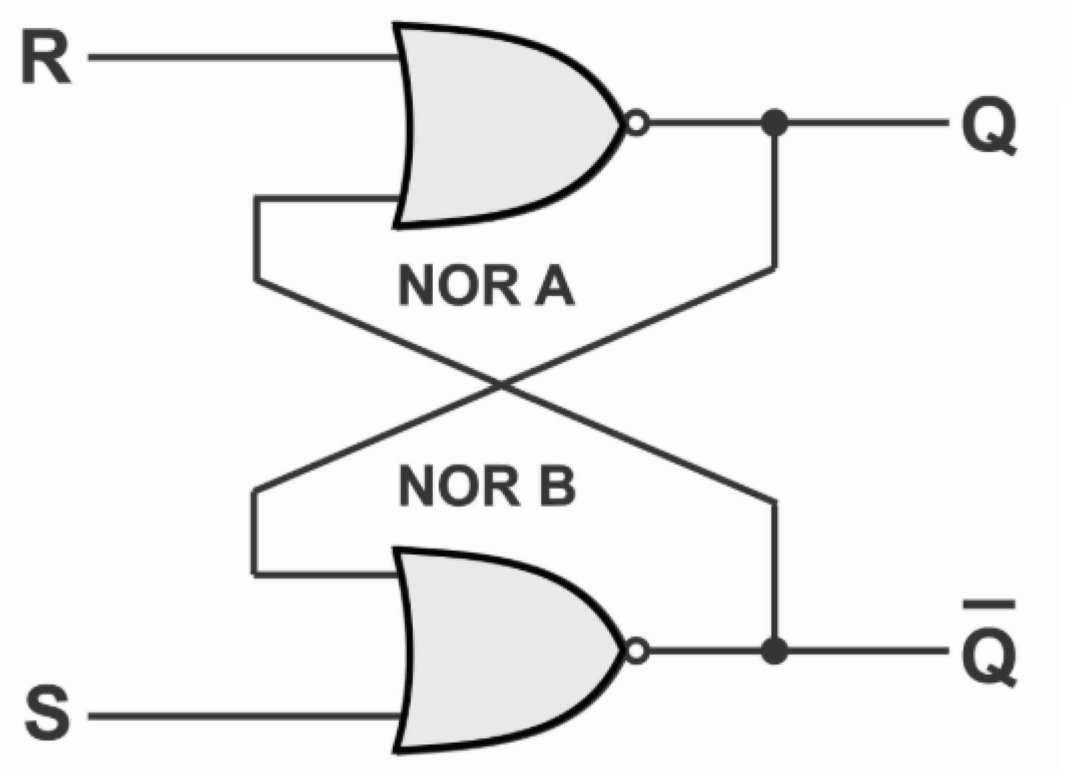
In other words, a way to “store” a bit ☺

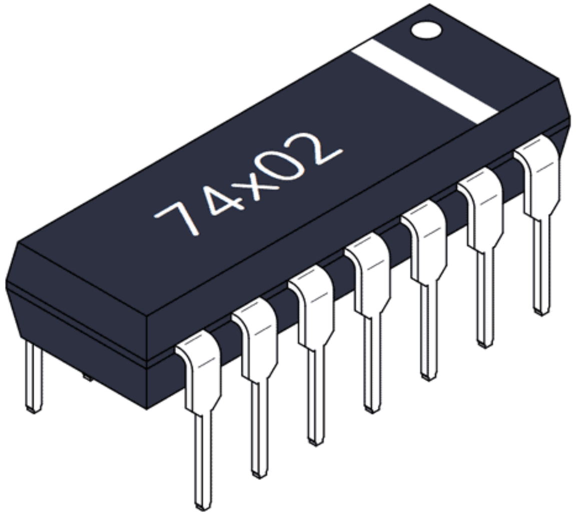
Small intermezzo (no exam material)

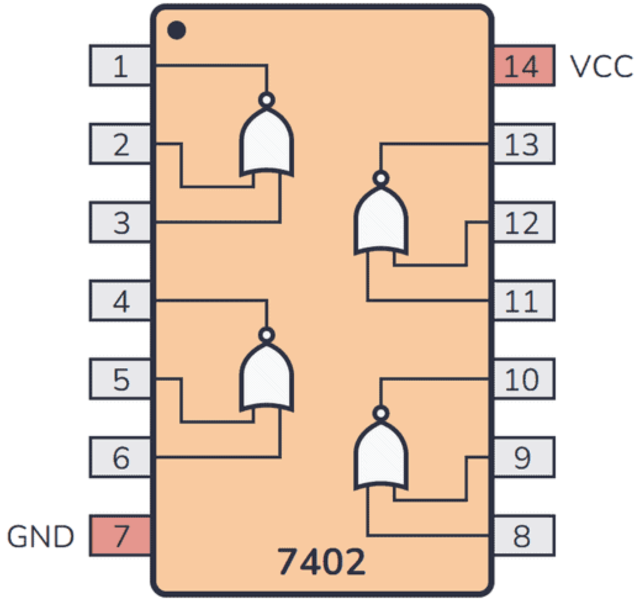
NOR



A	B	Output
0	0	1
1	0	0
0	1	0
1	1	0







Example (top-down approach): Website

- A website is created using HTML / CSS / JS (maybe frameworks & libraries)
  - Other stuff is also important: CDN, Load balancing, caching, authentication, protocol(s)…
- The site is being hosted on a server, that runs a program (webserver software)
- The site is being rendered in a browser (client), that is also a program running on a pc
- The browser is programmed in a programming language (C++, Rust for Firefox for example)
- File on disk + when used it is a process in memory

Example (top-down approach): Video Game

- Hogwarts legacy is created in Unreal Engine
- Unreal Engine is written in C++
- Even on a console like a PlayStation (5) for example
  - The end result are once again files
  - And processes running on the operating system of the PlayStation

It seems that at some point

- Everything is a **file**
- Everything is a **process** running on “something”
- Let’s focus on analysing these things! ☺

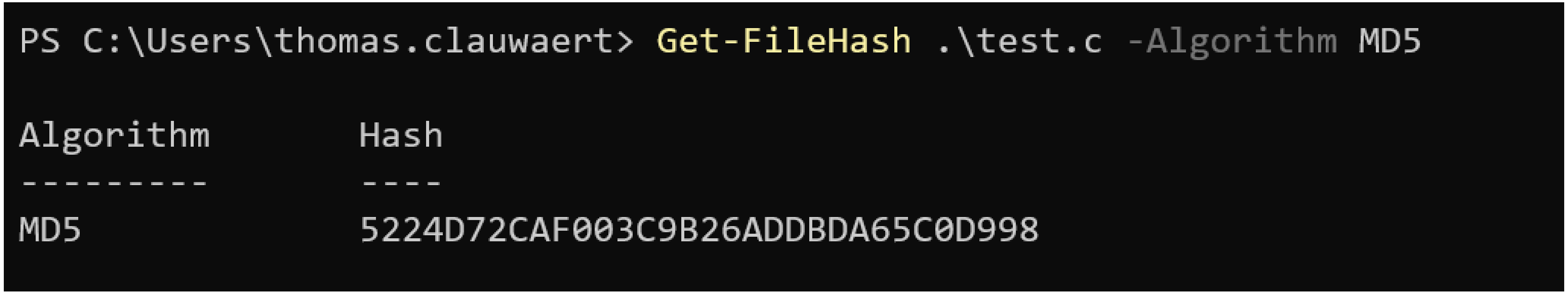
howest
university of applied sciences

File vs Process

A “File”

Just what is a “file”?

- Is it the same as a process?
- Where is a file stored?
- Is a file “running”?
- What is the hash of a file?
- Will the hash of a file change, if we modify the name of the file?



```
PS C:\Users\thomas.clauwaert> Get-FileHash .\test.c -Algorithm MD5

Algorithm       Hash
---------       ----
MD5             5224D72CAF003C9B26ADDBDA65C0D998
```

A “Process”

- Needs an executable file to be loaded from disk into **memory!**
- Often needs other processes to work together
- Something related: threads

→ We will get more into technical details in future lectures about this!

An “Executable” vs a “Process”

In other words:

- An executable is a file that contains executable code (= instructions for the OS (and in the end the CPU) to execute)
- a file resides **statically** on **disk**
- a process is **running** in **memory**

Note: In this course we will use executable, program, binary as synonyms

“Magic Bytes”

https://en.wikipedia.org/wiki/List_of_file_signatures

The first “few” bytes of a file that often determine what type of file this is.

Examples:

- 42 4D → BMP
- 89 50 4E 47 0D 0A 1A 0A → PNG
- FF D8 FF → *JPG*
- 4D 5A → DOS MZ Executable (.exe)

“Magic Bytes”

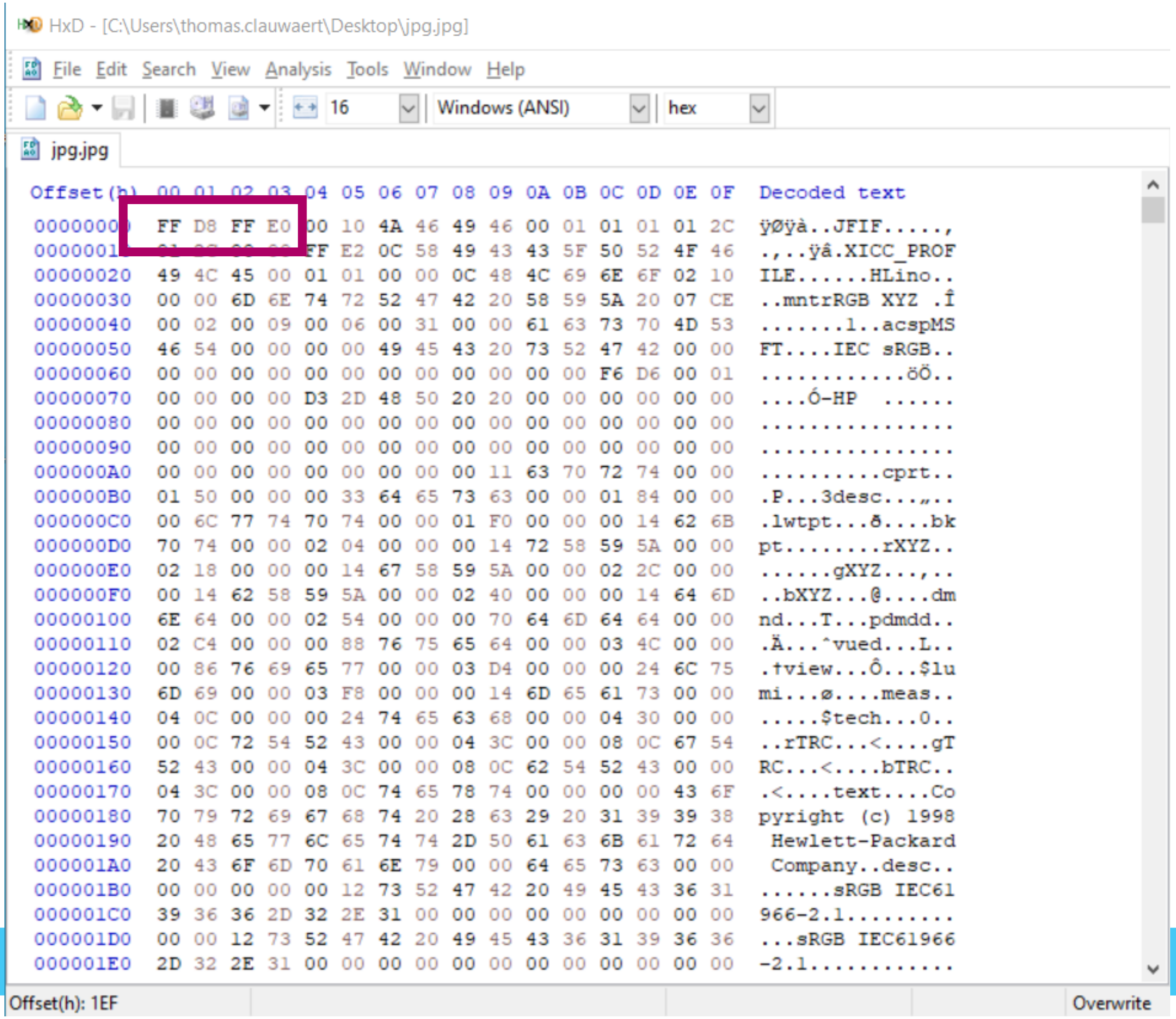
The **file** utility in Linux uses these magic bytes to know what software should be used to open the file.

Windows uses extensions (.exe, .pdf, .docx) to decide what software to use.

Question:

Will the magic bytes change of file.docx if we change the extension to file.pdf?

Hex Editor



HxD - [C:\Users\thomas.clauwaert\Desktop\jpg.jpg]
File Edit Search View Analysis Tools Window Help
16
Windows (ANSI)
hex
jpg.jpg

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000000  FF D8 FF E0 00 10 4A 46 49 46 00 01 01 01 01 2C  ÿØÿà..JFIF.....,
00000010  01 2C 00 00 FF E2 0C 58 49 43 43 5F 50 52 4F 46  .,..ÿâ.XICC_PROF
00000020  49 4C 45 00 01 01 00 00 0C 48 4C 69 6E 6F 02 10  ILE......HLino..
00000030  00 00 6D 6E 74 72 52 47 42 20 58 59 5A 20 07 CE  ..mntrRGB XYZ .Î
00000040  00 02 00 09 00 06 00 31 00 00 61 63 73 70 4D 53  .......1..acspMS
00000050  46 54 00 00 00 00 49 45 43 20 73 52 47 42 00 00  FT....IEC sRGB..
00000060  00 00 00 00 00 00 00 00 00 00 00 00 F6 D6 00 01  ............öÖ..
00000070  00 00 00 00 D3 2D 48 50 20 20 00 00 00 00 00 00  ....Ó-HP  ......
00000080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000A0  00 00 00 00 00 00 00 00 00 11 63 70 72 74 00 00  ..........cprt..
000000B0  01 50 00 00 00 33 64 65 73 63 00 00 01 84 00 00  .P...3desc...„..
000000C0  00 6C 77 74 70 74 00 00 01 F0 00 00 00 14 62 6B  .lwtpt...ð....bk
000000D0  70 74 00 00 02 04 00 00 00 14 72 58 59 5A 00 00  pt........rXYZ..
000000E0  02 18 00 00 00 14 67 58 59 5A 00 00 02 2C 00 00  ......gXYZ...,..
000000F0  00 14 62 58 59 5A 00 00 02 40 00 00 00 14 64 6D  ..bXYZ...@....dm
00000100  6E 64 00 00 02 54 00 00 00 70 64 6D 64 64 00 00  nd...T...pdmdd..
00000110  02 C4 00 00 00 88 76 75 65 64 00 00 03 4C 00 00  .Ä...ˆvued...L..
00000120  00 86 76 69 65 77 00 00 03 D4 00 00 00 24 6C 75  .†view...Ô...$lu
00000130  6D 69 00 00 03 F8 00 00 00 14 6D 65 61 73 00 00  mi...ø....meas..
00000140  04 0C 00 00 00 24 74 65 63 68 00 00 04 30 00 00  .....$tech...0..
00000150  00 0C 72 54 52 43 00 00 04 3C 00 00 08 0C 67 54  ..rTRC...<....gT
00000160  52 43 00 00 04 3C 00 00 08 0C 62 54 52 43 00 00  RC...<....bTRC..
00000170  04 3C 00 00 08 0C 74 65 78 74 00 00 00 00 43 6F  .<....text....Co
00000180  70 79 72 69 67 68 74 20 28 63 29 20 31 39 39 38  pyright (c) 1998
00000190  20 48 65 77 6C 65 74 74 2D 50 61 63 6B 61 72 64   Hewlett-Packard
000001A0  20 43 6F 6D 70 61 6E 79 00 00 64 65 73 63 00 00   Company..desc..
000001B0  00 00 00 00 00 12 73 52 47 42 20 49 45 43 36 31  ......sRGB IEC61
000001C0  39 36 36 2D 32 2E 31 00 00 00 00 00 00 00 00 00  966-2.1.........
000001D0  00 00 12 73 52 47 42 20 49 45 43 36 31 39 36 36  ...sRGB IEC61966
000001E0  2D 32 2E 31 00 00 00 00 00 00 00 00 00 00 00 00  -2.1............
```

HxD: https://mh-nexus.de/en/hxd/

Visual studio code can do this as well, and many others

On Linux: “od” & “xxd” and vim/nano ;)

howest
university of applied sciences

A bit of history: ASCII / Unicode

Encodings

Who recognises this?

V2VsY29tZSB0byB0aGlzIGxlY3R1cmUgb2YgUFJFTUEhCg==

Why was this created?

What is the main goal?

Base64

Binary to text encoding!

It takes “bits” and transforms into text, limited to a unique set of 64 characters

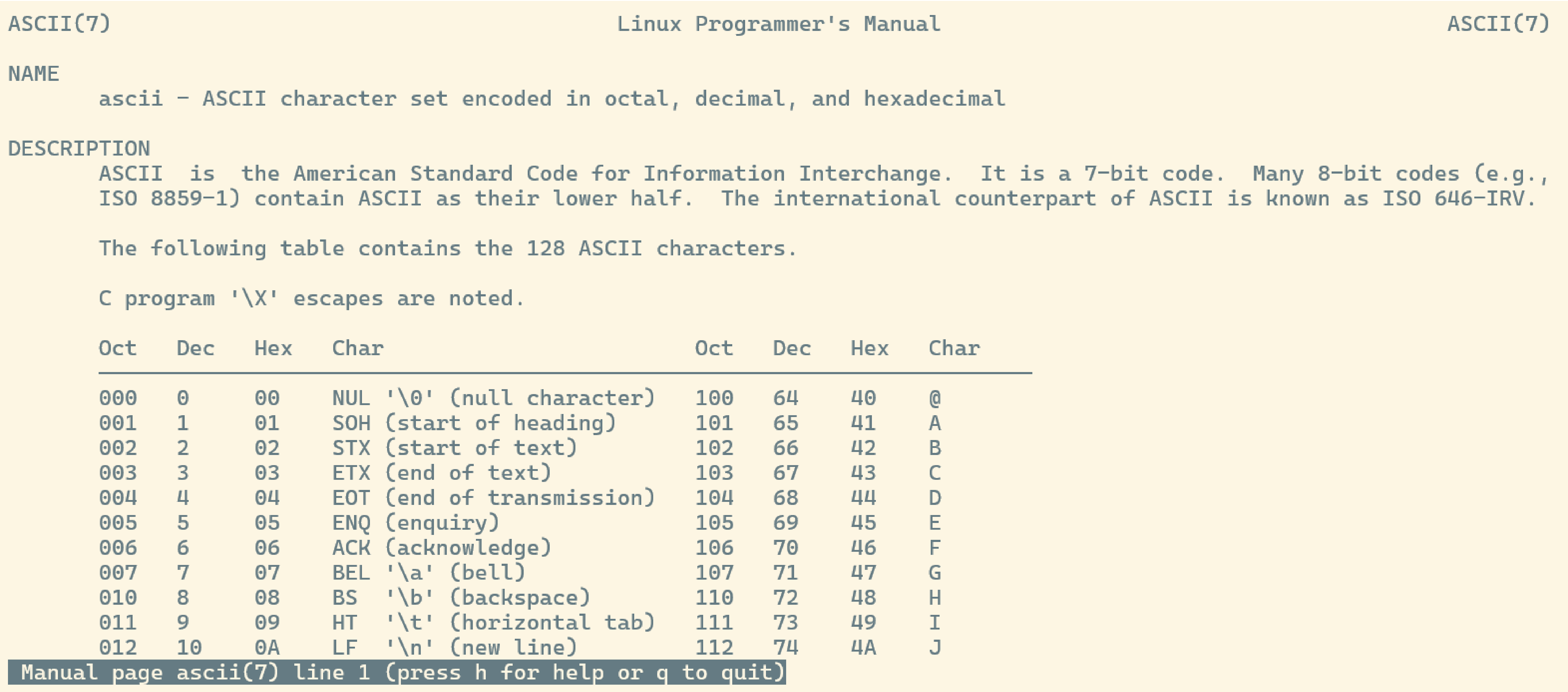
This was invented to properly “write down” or “transfer” data (bytes) that doesn’t necessarily have properly formatted characters in the English language for example.

But how and who decided how characters (the English alphabet) would be stored on a computer using 0’s and 1’s?

ASCII

- American Standard Code for Information Interchange = ASCII is an acronym
- +/- a mapping from the common characters (English alphabet) AND special buttons from Teletype devices to a number (decimal, hexadecimal)
- ASCII table, is the overview of the standard: On Linux: man ascii
- Uses 7 bits, so having 1 byte allows you to store a character

ASCII table



```
ASCII(7)                       Linux Programmer's Manual                      ASCII(7)

NAME
       ascii - ASCII character set encoded in octal, decimal, and hexadecimal

DESCRIPTION
       ASCII  is  the American Standard Code for Information Interchange.  It is a 7-bit code.  Many 8-bit codes (e.g.,
       ISO 8859-1) contain ASCII as their lower half.  The international counterpart of ASCII is known as ISO 646-IRV.

       The following table contains the 128 ASCII characters.

       C program '\X' escapes are noted.

       Oct   Dec   Hex   Char                        Oct   Dec   Hex   Char
       ────────────────────────────────────────────────────────────────────────
       000   0     00    NUL '\0' (null character)   100   64    40    @
       001   1     01    SOH (start of heading)      101   65    41    A
       002   2     02    STX (start of text)         102   66    42    B
       003   3     03    ETX (end of text)           103   67    43    C
       004   4     04    EOT (end of transmission)   104   68    44    D
       005   5     05    ENQ (enquiry)               105   69    45    E
       006   6     06    ACK (acknowledge)           106   70    46    F
       007   7     07    BEL '\a' (bell)             107   71    47    G
       010   8     08    BS  '\b' (backspace)        110   72    48    H
       011   9     09    HT  '\t' (horizontal tab)   111   73    49    I
       012   10    0A    LF  '\n' (new line)         112   74    4A    J
 Manual page ascii(7) line 1 (press h for help or q to quit)
```

ASCII

- Worked, everything was fine if you were a native English speaker.

- But what about ç, è, ß, おはよう 🔥 etc?

- Even by using 1 extra byte, this wouldn’t be a good solution.
  - Multiple encodings started to exist depending on where you lived in the world
  - Data transferred from America to Asia for example became tricky
  - Then the internet came

Unicode (U+2665) → ♥

- The standard today, it uses “codepoints” (not characters) → the number
- It found a way to “keep” ASCII to a certain way
- To allow more characters to be able to be represented in Unicode
- To not massively increase storage needs because of extra bytes that were needed
- ‘A’ is 41 (in hex) in ASCII and has U+0041 as Unicode

UTF-8

- UTF-8 is an example of a character encoding that uses Unicode.

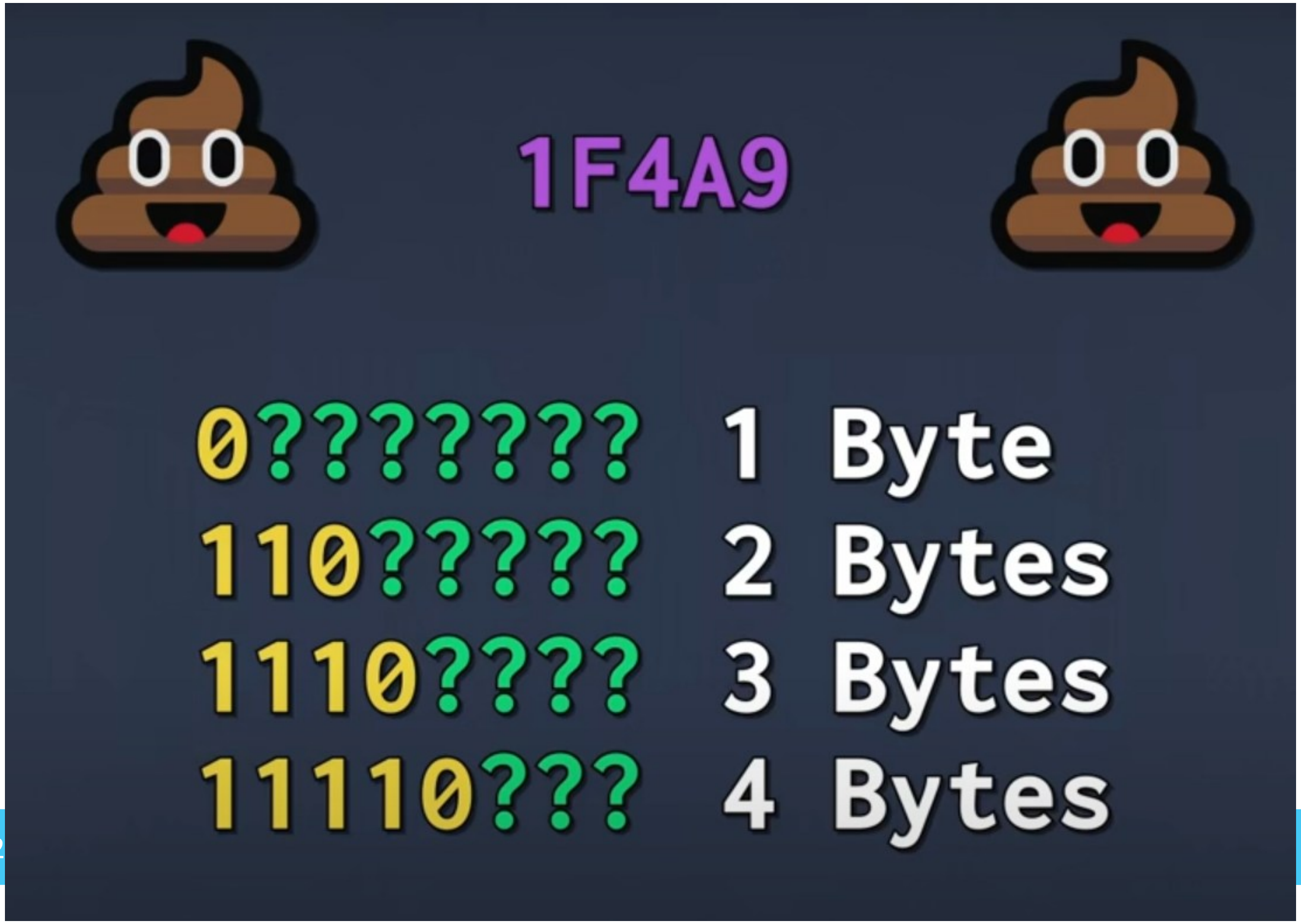
- There are exist others for example UTF-16 or UTF-32

- Variable length encoding
  - “The lower you are in the Unicode number, the less bytes you will need to store”
    - ASCII characters → Remain the same in number of bytes = 1 byte (so in other words backwards compatible) ☺
    - A recent emoji → Will require more bytes

- Most widely used today (It is estimated that at least 98% of all web pages are using this encoding.)

Example (no exam material)

If you want a very good explanation about ASCII, Unicode, UTF-8, give the following video a watch: https://www.youtube.com/watch?v=GMF2Z1EZHXk

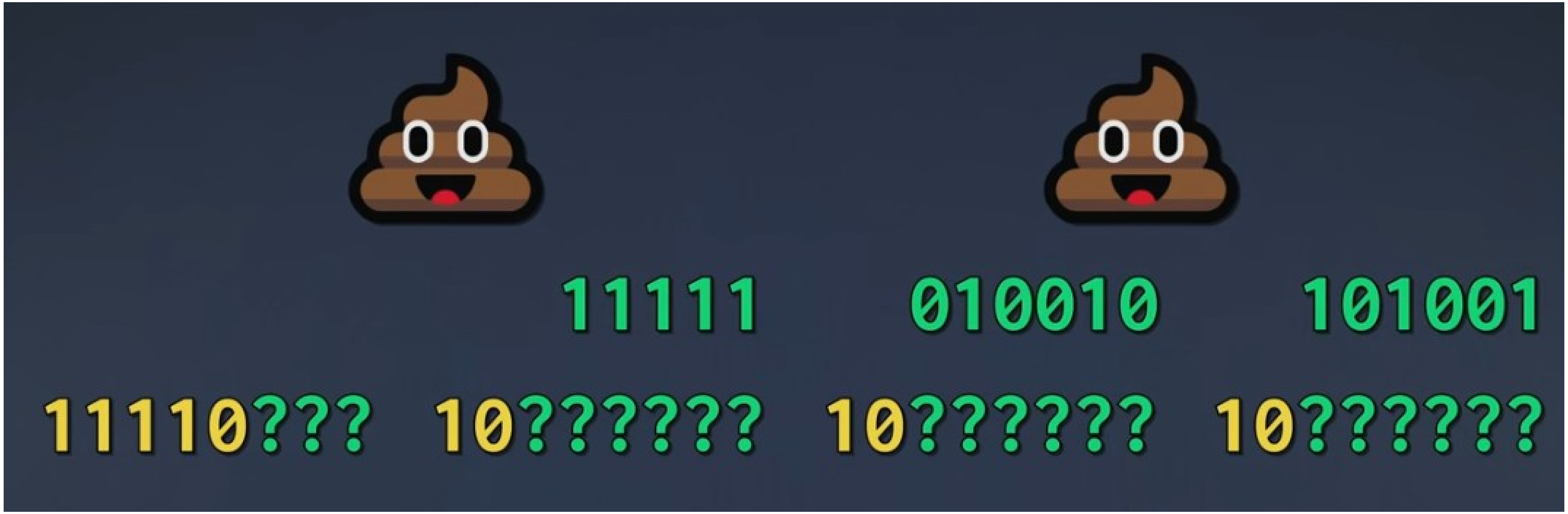


1F4A9

0???????	1 Byte
110?????	2 Bytes
1110????	3 Bytes
11110???	4 Bytes

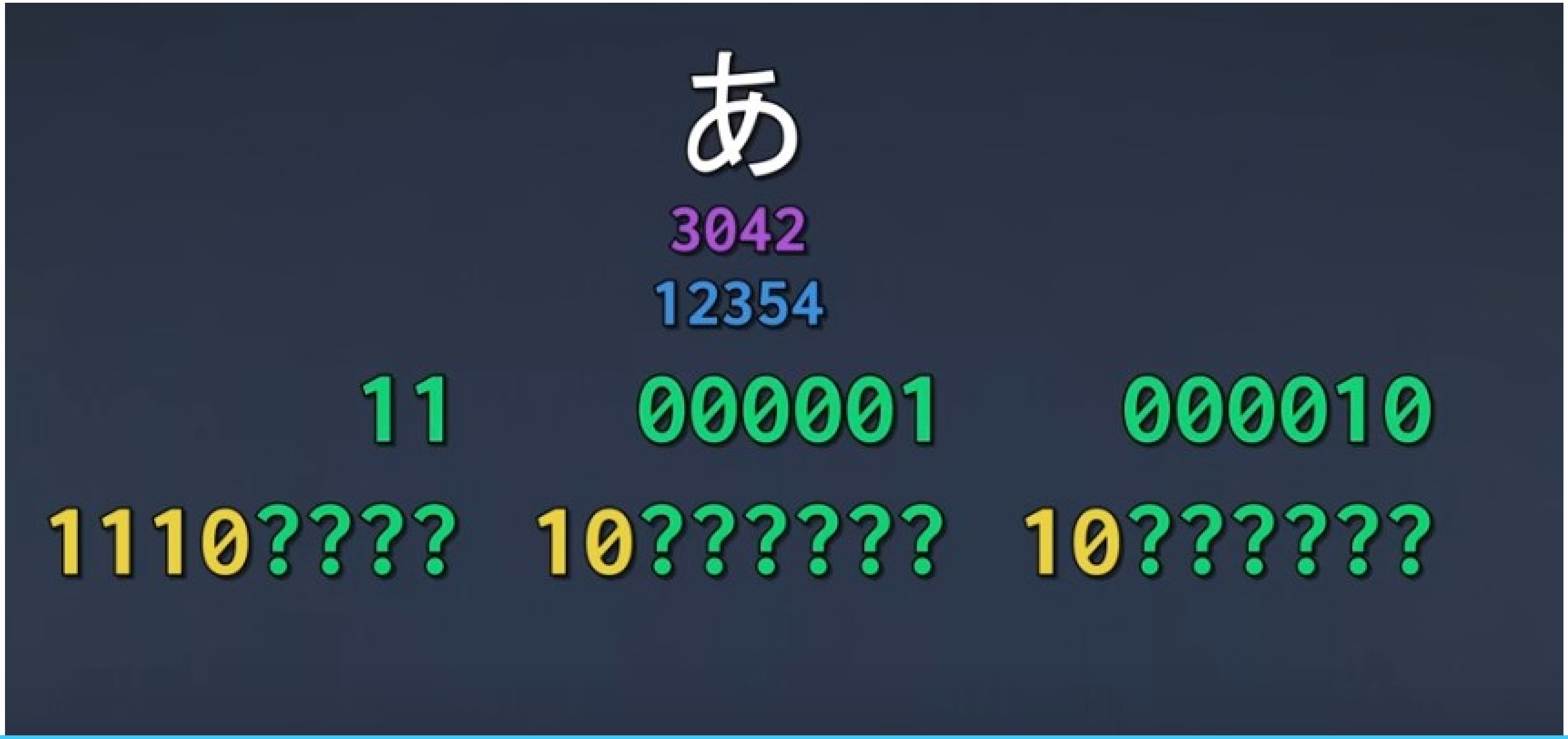
Example (no exam material)

If you want a very good explanation about ASCII, Unicode, UTF-8, give the following video a watch: https://www.youtube.com/watch?v=GMF2Z1EZHXk

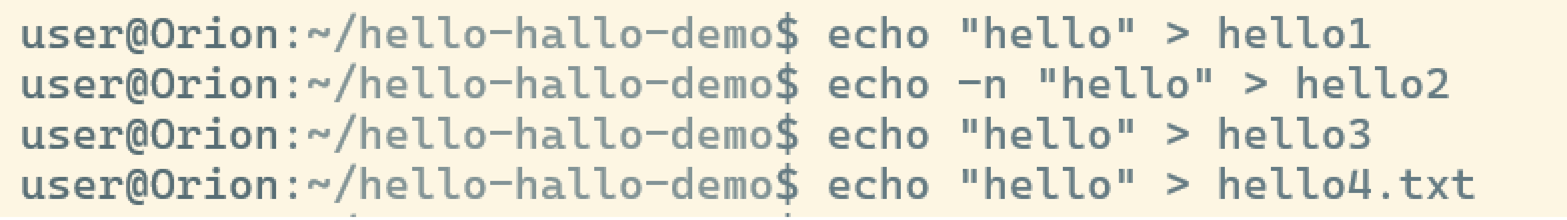


Example (no exam material)

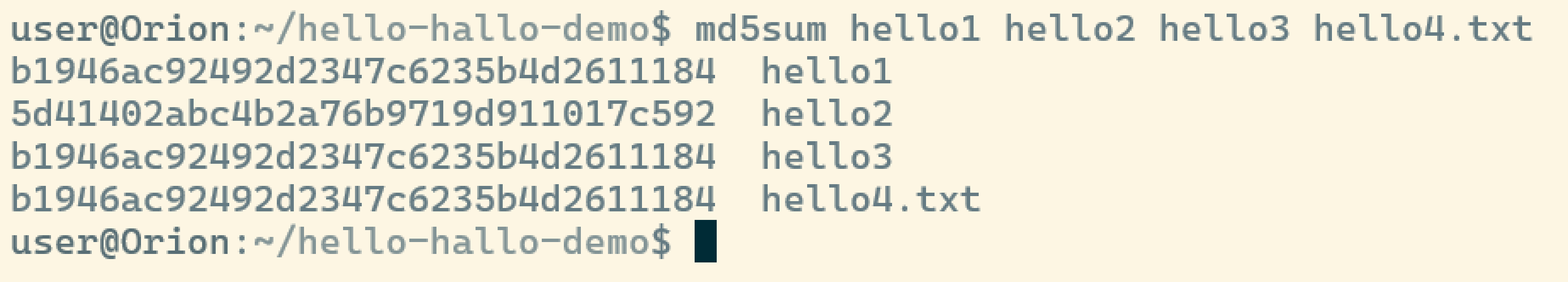
If you want a very good explanation about ASCII, Unicode, UTF-8, give the following video a watch: https://www.youtube.com/watch?v=GMF2Z1EZHXk



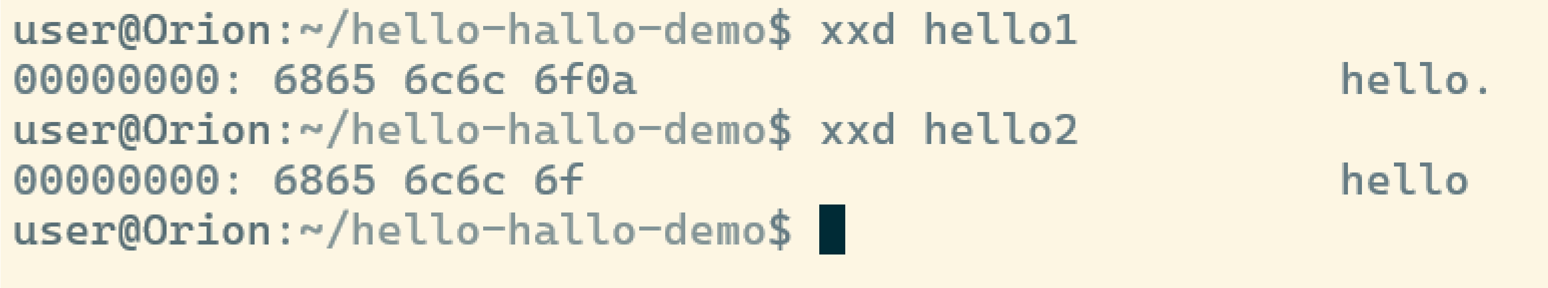
Hashes / Checksums / ASCII



```
user@Orion:~/hello-hallo-demo$ echo "hello" > hello1
user@Orion:~/hello-hallo-demo$ echo -n "hello" > hello2
user@Orion:~/hello-hallo-demo$ echo "hello" > hello3
user@Orion:~/hello-hallo-demo$ echo "hello" > hello4.txt
```



```
user@Orion:~/hello-hallo-demo$ md5sum hello1 hello2 hello3 hello4.txt
b1946ac92492d2347c6235b4d2611184  hello1
5d41402abc4b2a76b9719d911017c592  hello2
b1946ac92492d2347c6235b4d2611184  hello3
b1946ac92492d2347c6235b4d2611184  hello4.txt
user@Orion:~/hello-hallo-demo$ 
```



```
user@Orion:~/hello-hallo-demo$ xxd hello1
00000000: 6865 6c6c 6f0a                           hello.
user@Orion:~/hello-hallo-demo$ xxd hello2
00000000: 6865 6c6c 6f                             hello
user@Orion:~/hello-hallo-demo$ 
```

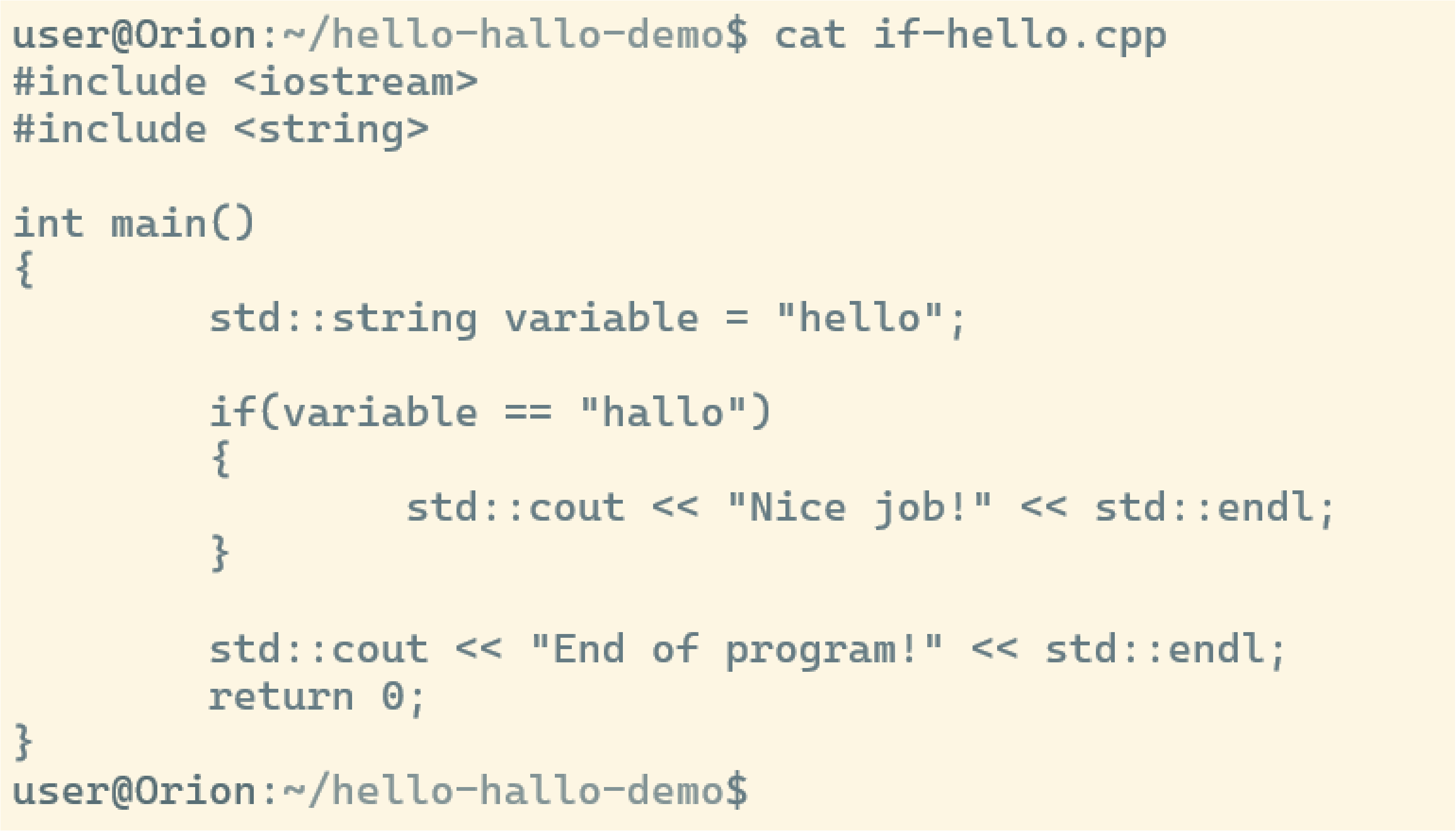
An executable contains *all* information it needs

An executable is a binary file containing instructions for the CPU. These instructions are not human readable. In a future lecture we will dive into these specifics.

However, since a lot of executables also contain “text”, using a specific encoding (ASCII, Unicode, etc), this text should also be present in between these instructions.

Let’s test this out with a demo!

Basic Static hacking ☺



```
user@Orion:~/hello-hallo-demo$ cat if-hello.cpp
#include <iostream>
#include <string>

int main()
{
        std::string variable = "hello";

        if(variable == "hallo")
        {
                std::cout << "Nice job!" << std::endl;
        }

        std::cout << "End of program!" << std::endl;
        return 0;
}
user@Orion:~/hello-hallo-demo$
```

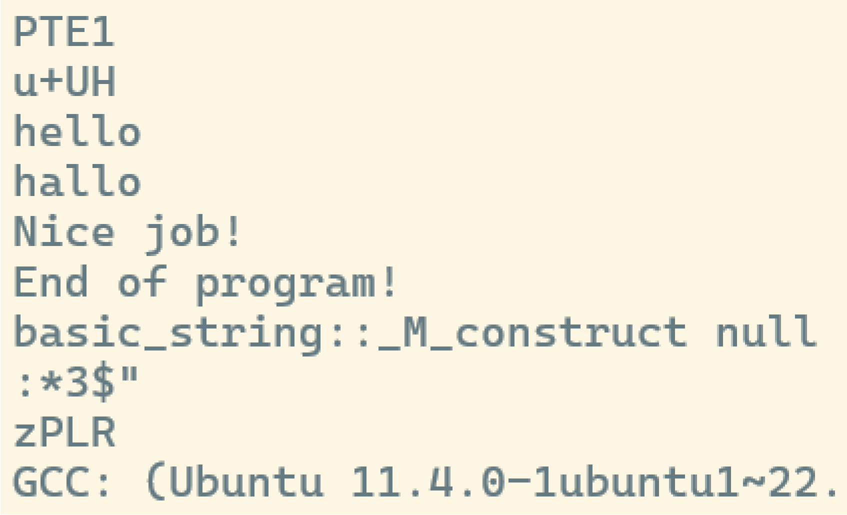
This was the original source code of the binary.

How can we manipulate the program to print the "Nice job!"?

←

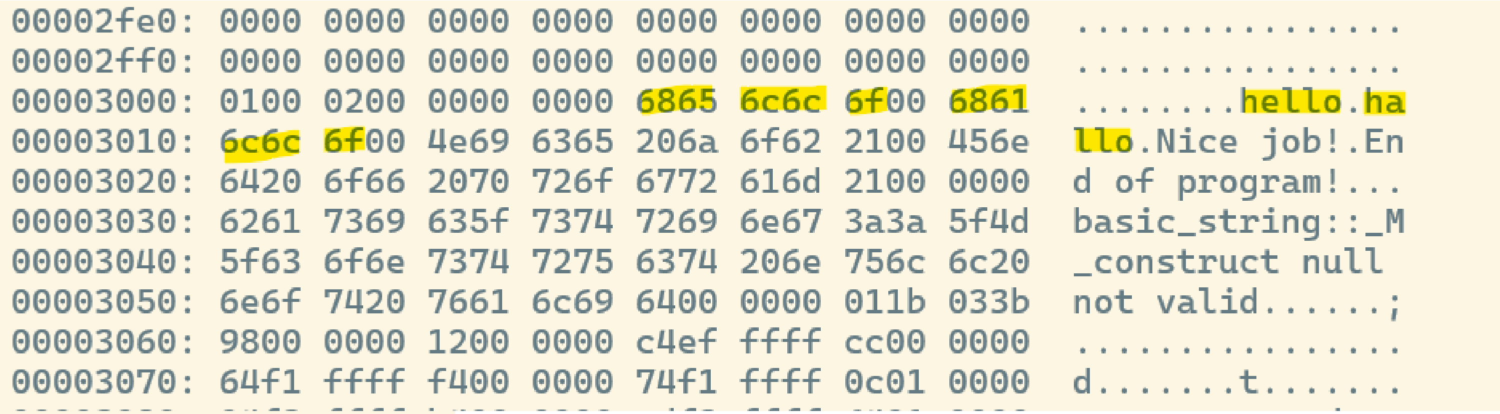
Basic Static hacking ☺

Strings ?



```
PTE1
u+UH
hello
hallo
Nice job!
End of program!
basic_string::_M_construct null not valid
:*3$"
zPLR
GCC: (Ubuntu 11.4.0-1ubuntu1~22.04.2) 11.4.0
```

Hex editor?

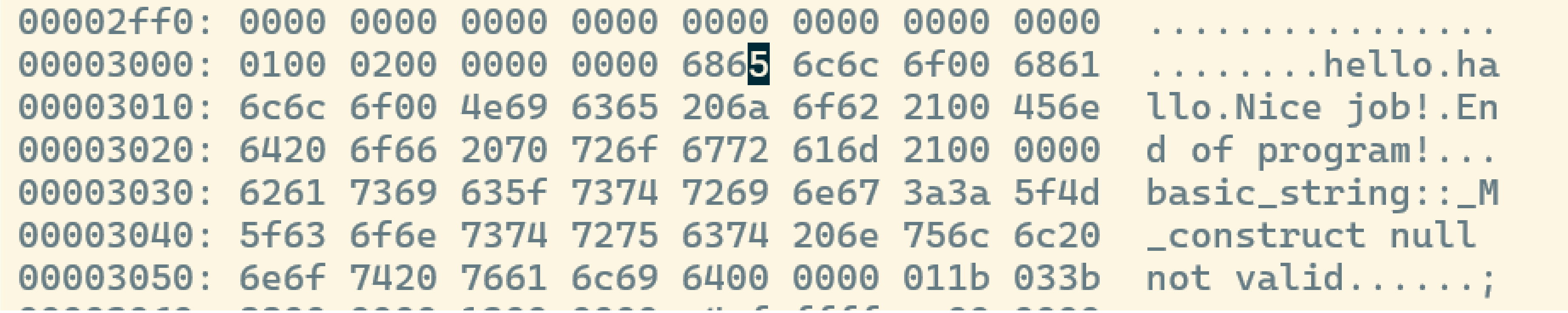


```
00002fe0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00002ff0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00003000: 0100 0200 0000 0000 6865 6c6c 6f00 6861  ........hello.ha
00003010: 6c6c 6f00 4e69 6365 206a 6f62 2100 456e  llo.Nice job!.En
00003020: 6420 6f66 2070 726f 6772 616d 2100 0000  d of program!...
00003030: 6261 7369 635f 7374 7269 6e67 3a3a 5f4d  basic_string::_M
00003040: 5f63 6f6e 7374 7275 6374 206e 756c 6c20  _construct null
00003050: 6e6f 7420 7661 6c69 6400 0000 011b 033b  not valid......;
00003060: 9800 0000 1200 0000 c4ef ffff cc00 0000  ................
00003070: 64f1 ffff f400 0000 74f1 ffff 0c01 0000  d.......t.......
```

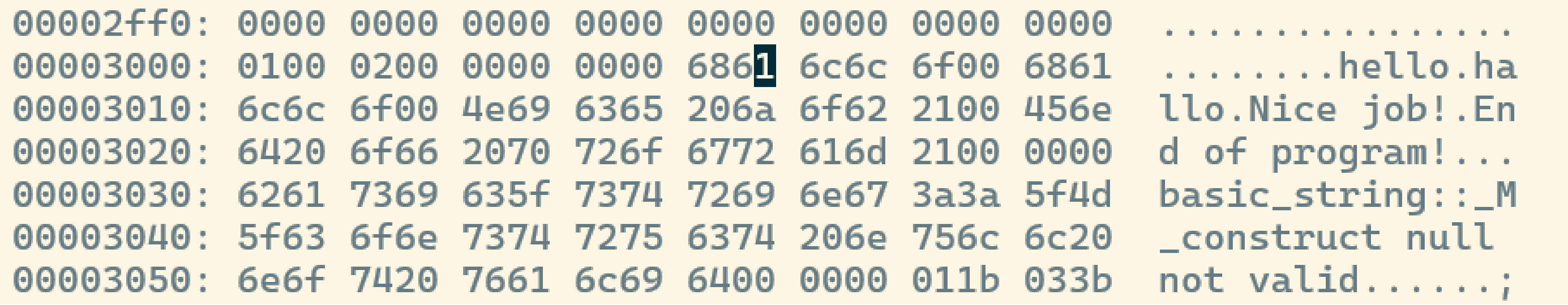
Basic Static hacking ☺

Let’s dump the hex data in a new file: *xxd if-hello > if-hello.hex*

Open it using a text editor (vim/nano/…) and change the “e” to “a” in the ASCII bytes of hello we found previously



```
00002ff0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00003000: 0100 0200 0000 0000 6865 6c6c 6f00 6861  ........hello.ha
00003010: 6c6c 6f00 4e69 6365 206a 6f62 2100 456e  llo.Nice job!.En
00003020: 6420 6f66 2070 726f 6772 616d 2100 0000  d of program!...
00003030: 6261 7369 635f 7374 7269 6e67 3a3a 5f4d  basic_string::_M
00003040: 5f63 6f6e 7374 7275 6374 206e 756c 6c20  _construct null
00003050: 6e6f 7420 7661 6c69 6400 0000 011b 033b  not valid......;
```



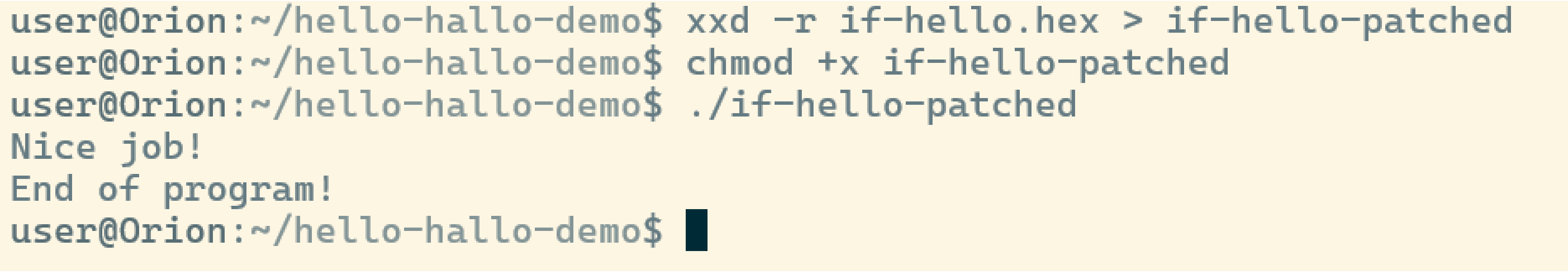
```
00002ff0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00003000: 0100 0200 0000 0000 6861 6c6c 6f00 6861  ........hello.ha
00003010: 6c6c 6f00 4e69 6365 206a 6f62 2100 456e  llo.Nice job!.En
00003020: 6420 6f66 2070 726f 6772 616d 2100 0000  d of program!...
00003030: 6261 7369 635f 7374 7269 6e67 3a3a 5f4d  basic_string::_M
00003040: 5f63 6f6e 7374 7275 6374 206e 756c 6c20  _construct null
00003050: 6e6f 7420 7661 6c69 6400 0000 011b 033b  not valid......;
```

Basic Static hacking ☺

Let’s restore from hex back to a binary: *xxd -r if-hello.hex > if-hello-patched*

Make the new program we created executable: *chmod +x if-hello-patched*

And run:



```
user@Orion:~/hello-hallo-demo$ xxd -r if-hello.hex > if-hello-patched
user@Orion:~/hello-hallo-demo$ chmod +x if-hello-patched
user@Orion:~/hello-hallo-demo$ ./if-hello-patched
Nice job!
End of program!
user@Orion:~/hello-hallo-demo$ 
```

howest
university of applied sciences

Let’s get practical

“PREMA” – Virtual Machine(s)

- See – leho page for details, we will update during the semester

- We will need a **Windows client**
  - For analysing the malware

- We will need a **Linux** machine
  - Most of all for learning some “operating system things”
  - https://remnux.org/ is a Linux distro
  - Kali is fine
  - A debian without GUI is fine as well ☺

Lab1.zip

- No malware (→ can be done while installing/downloading your virtual machine)

Let's play a CTF = Capture The Flag

A "FLAG" is something like "FLAG-000000".