

This work is licensed under a Creative Commons license



Attribution-NonCommercial-NoDerivatives 4.0 International  
(CC BY-NC-ND 4.0)

You are free to:

**Share** copy and redistribute the material in any medium or format.

Under the following terms:

**Attribution** You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

**NonCommercial** You may not use the material for commercial purposes.

**NoDerivatives** If you remix, transform, or build upon the material, you may not distribute the modified material.

# The Life Cycle of Binaries

How binary programs are built and run

Giovanni Lagorio

`giovanni.lagorio@unige.it`

`https://csec.it/people/giovanni\_lagorio`

X & GitHub: `zxgio`

DIBRIS - Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi  
University of Genova, Italy



`www.zenhack.it`

# Outline

- 1 Introduction to Binaries, and their Analysis
- 2 Compilation and Linking
- 3 Computing Platforms

# Introduction

We'll

- study the life cycle of (binary) programs
  - how they are built, loaded and run
- understand how to deduce properties and behavior of programs
- review how modern operating systems work
  - examine some aspects of prominent ones

# Market share

According to various reports:

- 2013 — 2022, **Windows is the dominant *desktop* OS**, with **76%** share
  - Apple remains a minor player
  - Linux has a small but stable share
- in 2017, according to the Linux Foundation, **Linux**
  - runs **90% of the public cloud** workload
  - has **62% of the embedded market** share
  - **99% of the supercomputer market** share
  - runs **82% of smartphones**

# Microsoft ♡ Linux?

As declared by CEO Satya Nadella in 2015?

- 2016, joined The Linux Foundation (Platinum, \$500,000 annually)
- 2016, introduced **WSL** with Windows 10 Anniversary Update,
- 2019, announced **WSL 2**
- 2020, developed an internal Linux distribution, CBL-Mariner
- 2021, released **WSLg** and a Windows Store version of WSL

Scott Hanselman jokes (e.g. see [Developing for Linux on Windows](#)) that this is *the year of the Linux desktop*

# WSL magic

WSL2 transparently handles a VM, and you can access

- Linux files via `\\wsl$\distro\...`
- Windows disks via `/mnt/drive-letter/path`

Moreover,

- Listening sockets opened in WSL are reachable from Windows
- Your Windows PATH is appended to WSL path by default
- In WSL you can run Windows executables, they
  - retain the working directory (as UNC path; e.g. `\\wsl$\Ubuntu\home\...`; so `cmd.exe /c` doesn't work, but `powershell.exe /c` does)
  - run as the active Windows user, with permissions as the WSL process
- In Windows you can run Linux executables by prepending `wsl`
- I/O redirection works

We'll use:

- Windows 11
- Ubuntu/Ubuntu-derived Linux distros
  - Suggested: Ubuntu 24.04 (It's the one I'm currently using)

*however*, the same concepts apply to other versions



# Warm-up

What are binaries?

Let's start with some examples to warm up

→ `file0.exe`, ...

## File extensions are immaterial

- For the OS, the content of a file is simply a sequence of bytes
- Different *parsers* can interpret the same sequence differently
  - ZIP/JAR parsers look for the “End of Central Directory” from the end of the file
  - BMP/PE/ELF/... parsers expects an header at *the beginning*
  - PDF header *should* be at the beginning, but most viewers are happy if it is in the first 1024 bytes of the file
  - ...

# Polyglots

Files meeting the specifications of multiple file-formats are called **polyglots**

- e.g. [Janus.com](#)  
*[...] a 512-byte file that is simultaneously an x86 bootloader, COM executable, ELF, ZIP, RAR, GNU Multiboot2 Image and Commodore 64 PRG executable [...]*

<https://xcellerator.github.io/posts/bggp21/>

- *Ange Albertini* has done a lot of interesting work on file-formats and polyglots; e.g. *Funky File Formats*

<https://www.youtube.com/watch?v=hdCs6bPM4is>

- See also:

[Magika](#) <https://github.com/google/magika>

[Polyfile](#) <https://github.com/trailofbits/polyfile>

# How can you “take a look” inside a binary file?

You can't really cat it...



<https://twitter.com/Q3w3e3/status/1256293618991669249>

# Generic and structure-aware tools

## Generic

- the (in)famous `strings`
- any hex-editor

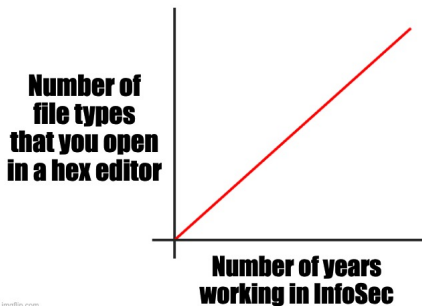
Linux `bvi`, `ghex`, ...

Windows `HxD` <https://mh-nexus.de/en/hxd/>, ...

## Structure-aware

- Kaitai, a parser+viewer for binary structures  
<https://ide.kaitai.io/>
- similar, but can also edit: *010 Editor*, non-free, 30 day trial  
<https://www.sweetscape.com/010editor/>
- the open-source alternative, `ImHex`  
<https://github.com/WerWolv/ImHex>

# The more you work in infosec. . .



<https://twitter.com/cyb3rops/status/1493885187729547265>

Joking aside, some tools offer specific features for executable formats. . .

# Tools for executables

Even more specific tools:

ELF+PE **hte** can also *edit* — *note*: Ubuntu package name is ht

<https://hte.sourceforge.net/>

<https://github.com/sebastianbiallyas/ht>

ELF • **readelf** and **objdump**

- **objdump** can also disassemble, and can parse PE too

- **XELFViewer**

<https://github.com/horsicq/XELFViewer>

- ...

PE • **dumpbin** more or less an **objdump** for PE files

- **PE Bear**

<https://github.com/hasherezade/pe-bear>

- **PE Studio** <https://www.winitor.com/>

- **XPEViewer**

<https://github.com/horsicq/XPEViewer>

- ...

# Hello world

At last we have the “obligatory” C program:

```
#include <stdio.h>

int main()
{
    printf("Hello world!\n");
}
```

...can we run it?

→ c\_examples/hello\_world/hello-world.c

## Long story short

- 1 CPUs do not “understand” C, they execute (their own) machine code
- 2 This is an ASCII **text file**, not a sequence of machine instructions
- 3 So, to run this “hello world” program, we first need to *compile* it

By doing so, we get an **executable**, that we can run



# Binary Analysis

Can you be sure that the compiled program has the same semantics as the corresponding source? The unnerving answer is that **you can't!**

## There might be surprises

As a matter of fact, it's extremely likely that running the compiled\* version of the previous “Hello world” does *not* call the function `printf` (!)

(\*) using gcc, unless `-fno-builtin` is specified

*Binary analysis is the science and art of analyzing the properties of binary computer programs, called **binaries**, and the machine code and data they contain. ...the goal [...] is to figure out (and possibly modify) the true properties of binary programs: **what they really** do, as opposed to what we think they should do. [And18]*

# Static vs Dynamic Analysis

Broadly speaking, the tools at our disposal are:

- **Dynamic analysis: we run the binary and analyze it**, or log its behavior, as it executes
  - **often simpler**, can observe runtime states
  - **can be harmful**; e.g., malware
  - not everything is necessarily apparent
  - for each run you observe *that* particular execution, and might miss *interesting* parts of the code; e.g.

```
if (random()==0xcafebabe) { /* interesting stuff */ }
```

- **Static analysis: we reason about the binary without running it**
  - you can analyze the whole binary in one go
  - you don't need a CPU/system that can run such a binary
  - (obviously, almost) no knowledge of runtime states
  - **can be difficult to pinpoint *interesting* parts**

# Challenges

- low-level languages
- often, no “symbols”; i.e. meaningful names
- (almost) no type information
- no class/module/namespace boundaries
- code and data can be mixed, and they usually do
- difficulty in adding/changing/removing instructions
- ...

Let's take a step back: how are (binary) programs built?

# Outline

- 1 Introduction to Binaries, and their Analysis
- 2 **Compilation and Linking**
- 3 Computing Platforms

# Compilation pipeline

gcc/g++, clang(++), cl.exe are *front-ends*, which run

- 1 C/C++ **pre-processor**, which handles macros/includes
- 2 C/C++ **compiler**, which translates into **assembler code**
- 3 **assembler**, which produces **relocatable/object files**:

$$*.c + *.h \xrightarrow{(\text{cpp})+cc1} *.s \xrightarrow{\text{as}} *.o$$
$$\text{sources in some language} \xrightarrow{\text{compiler}} *.s \xrightarrow{\text{as}} *.o$$

- 4 (static) **linker**, which links relocatable files and libraries to produce the **executable/dynamic-library**

With default dynamic-linking, the **loader + dynamic-linker** finish the job

# Assembler files

A step towards machine code, however they contain

- assembler **instructions**, that will be translated into machine code
  - e.g., `push rbp`  $\Rightarrow$  `55`
- **pseudo-instructions**, to emit arbitrary data/bytes
  - e.g., `.string "Hello world!"`  $\Rightarrow$  `48 65 6c 6c 6f...`
- **directives**, to give the assembler information/commands
  - e.g., `.text`  $\Rightarrow$  following items belongs to the `.text` section

Moreover, they use **symbols (=names)** instead of addresses

- defined; e.g., `main: push rbp ...`
- undefined; e.g., `call printf` gets translated into:
  - 1 `e8 ?? ?? ?? ??`; that is, machine code with “holes”
  - 2 and metadata: **relocations** that tell the linker how to “fill such holes”

$\rightarrow$  `c_examples/ex1/hello.c`, `hello.s` and `hello64.o`

# Sections

Different data types are kept in different **sections**; e.g.

Common `.text` → code

`.(r/ro)` data → initialized (read-only) data

`.bss` → uninitialized data

ELF `.symtab/.dynsym` → **symbol table**/dynamic symbol table

`.strtab/.dynstr` → symbol table's string tables

`.rel[a](.dyn)` *section* → (dynamic) **relocations** for *section*

`.shstrtab` → string table for section names

...

PE `.rsrc` → embedded resources

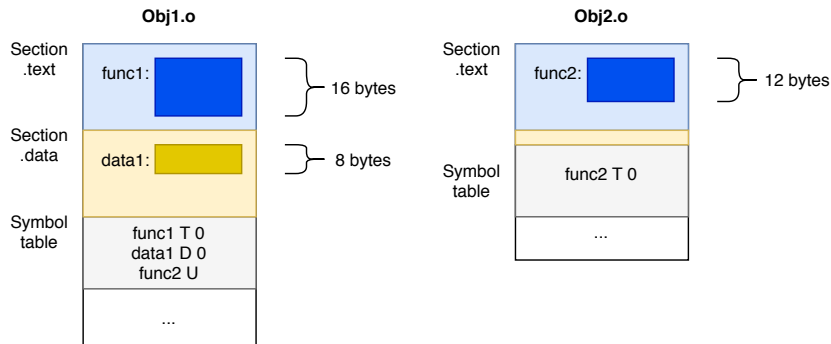
`.reloc` → relocations

...

# Section merging (1/2)

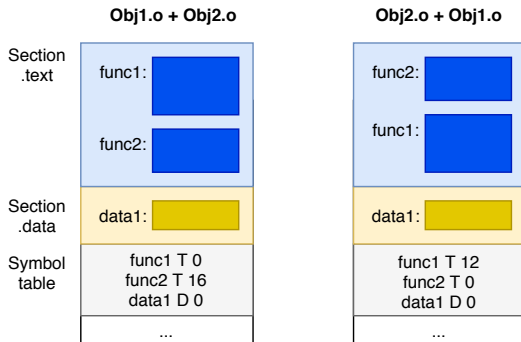
Object files are merged by the **link editor**, that

- **concatenates** all sections with the same name, **applying relocations**
- **merges symbol tables, resolving undefined symbols** and making sure that no symbols is multiple defined (there are exceptions)





# Section merging (2/2)



# Static vs Dynamic linking

The last phase is **linking**, where the linker combines object-files/libraries

Then, with

- **Static linking** executables are **self-contained**
- **Dynamic linking** executables/libraries contain
  - 1 **your code and data**
  - 2 **metadata** specifying what libraries/functions they need

so

- smaller programs, that *automatically* use the most recent libraries
- less RAM needed (we'll see why later)

# Dynamic linking

Dynamic linking can be:

- **implicit**, i.e. libraries are automatically loaded and linked when the program is run. Functions can be either
  - **eagerly bound**, at load-time
  - **lazily bound**, the first time they are called
- **explicit**; programs can
  - **load (and link)** dynamic libraries, by invoking special functions
  - **lookup symbols** in those

Who brings programs in memory and make them run?

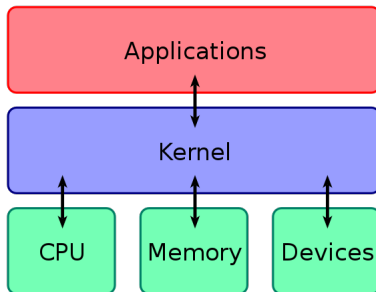
# Outline

- 1 Introduction to Binaries, and their Analysis
- 2 Compilation and Linking
- 3 Computing Platforms

# The operating system layer

Unless you work on bare metal, programs are executed (in **user-mode**) on top of an **Operating System** (in **supervisor/kernel-mode**), which

- handles and virtualises hardware devices
- manages resources
- offers handy abstractions, e.g. a file system view of a block device
- ...



Source: Wikipedia

# Processes

When you run a program, you create\* a process, identified by a PID

To see the process list:

**Linux** `proc(5)`, `ps(1)`, `(h)top(1)`, ...

**Windows** `pslist`, Task manager, Process Explorer, ...

Each process has its own

- threads, an abstraction of CPUs
- address space, an abstraction of RAM  
→ `c-examples/proc-examples/mem_example.c`
- security context
- handles/file-descriptors corresponding to “open” resources
- ...

(\*) technically, this is a lie; details later

# System calls

The OS offers its services by exposing **system calls**, often wrapped in *functions* that form the **Application Programming Interface**

- e.g. to open a file, `open` in POSIX or `CreateFile` in Win32
- For C/C++ programmers, “invoking a system call” means calling a *wrapper function*, which executes special instructions

`Linux libc.so` exposes the syscall wrappers

`Windows ntdll.dll` exposes the syscall wrappers

- Which are, in turn, wrapped by API functions in `kernel32.dll`, `user32.dll`, ...  
E.g. `kernel32.CreateFile` relies on `ntdll.NtCreateFile`
- Languages may offer their own (more abstract) APIs; e.g. `fopen`

Compiled programs must be compliant to an **Application Binary Interface**. . .

An **ABI** is a set of **specifications** allowing the interface of programs, libraries and the operating system, detailing:

- executable and object file formats
- fundamental types (e.g. size and alignment for int, long, ...)
- how data types are laid out in memory
- (sys)calling conventions
- how programs are started up (initialization, dynamic linking, ...)



# ABIs

Different architectures have different ABIs, and you can have even **multiple ABIs** on the same system. Examples of differences are:

- Machine instructions to invoke a system call:

32-bits, x86 INT and, on some CPUs, SYSENTER

64-bits, x64 SYSCALL

- System calls are identified by a number, which can change from version to version; e.g.

Linux open is 5 for x86, 2 for x64, and 0x900005 for ARM

- This is stable for *all* kernel versions (in “regular” PCs)

Windows NtCreateFile is

x86 0x163 on 8.0, 0x168 on 8.1, 0x016e on 10.1507, 0x170 on 10.1511, 0x172 on 10.1607, ...

x64 0x53 on 8.0, 0x54 on 8.1, 0x55 on 10.1507 to 10.20H2

See <https://syscalls.w3challs.com/>

## ISA

The running code consists of

- **user-mode machine instructions** of the Instruction Set Architecture, e.g. x86/IA-32 (implemented by Intel 80486, Intel Pentium, AMD Athlon, ...) or ARMv7 (ARM Cortex A12, Apple A6, ...)
  - what if a process executes an instruction that is not in such a subset?
- **system calls** to the OS
  - you can think of system calls like a sort of “macro-instructions”

ISA+ABI implements an **abstract machine** for running programs

ISA is typically implemented in HW, but *emulators* are (hardware or) software enabling one computer system to behave like another one

Some notable open-source projects are:

- **QEMU** [Bel05] a generic machine emulator and virtualizer  
<https://www.qemu.org/>
- **MAME** a multi-purpose emulation framework, with the purpose is to preserve decades of software history  
<https://www.mamedev.org/>
- **DOSBox** a DOS emulator to re-live the good old days  
<https://www.dosbox.com/>

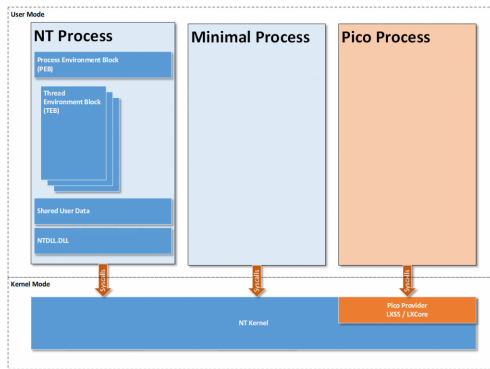
# Translator layers

An ABI can be implemented on top of another; e.g.:

- **Wine** is a compatibility layer capable of running (many) Windows applications on Linux  
<https://www.winehq.org/>
- **WSL 1** allows us to run (many) Linux programs on Windows  
<https://docs.microsoft.com/en-us/windows/wsl>
- **WoW64** "... allows 32-bit Windows-based applications to run seamlessly on 64-bit Windows"  
<https://docs.microsoft.com/en-us/windows/win32/winprog64/running-32-bit-applications>

# Windows-specific: Minimal and Pico processes

- Win8.1: **minimal processes**, empty address space, no handle-table
  - no official API, used for “secure system” and “memory compression”
- A **Pico Process** is a minimal process with a kernel **Pico Provider**
  - all system calls and exceptions forwarded to the provider to handle as it sees fit; e.g. **WSL 1** (`wsl --set-version Ubuntu 1`)



<https://docs.microsoft.com/en-us/archive/blogs/wsl/pico-process-overview>

# More resources

- **Back to Basics: Compiling and Linking**  
by Ben Saks @ CppCon2021  
<https://youtu.be/cpkDQaYttR4>
- **The Bits Between the Bits: How We Get to main()**  
by Matt Godbolt @ CppCon2018  
<https://www.youtube.com/watch?v=dOfucXtyEsU>
- **The C++ ABI From the Ground Up**  
by Louis Dionne @ CppCon2019  
<https://youtu.be/DZ931P1I7wU>
- **C and C++ Compiling**, by Milan Stevanovic [Ste14]
- Useful video to understand user/kernel interface: “Linux User/Kernel ABI: the realities of how C and C++ programs really talk to the OS”  
by Greg Law @ ACCU 2018  
<https://www.youtube.com/watch?v=4CdmGxc5BpU>

# References

- [And18] Dennis Andriesse.  
*Practical Binary Analysis*.  
No Starch Press, 2018.
- [Bel05] Fabrice Bellard.  
QEMU, a fast and portable dynamic translator.  
In *USENIX Annual Technical Conference, FREENIX Track*,  
volume 41, page 46, 2005.
- [Ste14] Milan Stevanovic.  
*Advanced C and C++ compiling*.  
Apress, 2014.