

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.

# Debugging

## An introduction

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

# Outline

- 1 Introduction
- 2 Linux and Windows Debuggers
- 3 Symbols and debug information
- 4 Controlling the execution
- 5 Backtraces

Debuggers are software (or hardware) that permit to get an insight into what a program/system is doing, by

- pausing the execution at specific places
  - optionally, when some conditions are met
- showing the contents of registers and memory
- resuming the execution
- ...

# DEBUGGING

THE CLASSIC MYSTERY GAME

WHERE YOU ARE

THE DETECTIVE,

THE VICTIM,

AND THE MURDERER!




<https://twitter.com/lucacarettoni/status/1200538741946929153>

*"Debuggers don't remove bugs.  
They only show them in slow motion."  
(unknown)*

# Kind of debuggers

We can distinguish between:

- Source-level vs Assembly-level debuggers
  - Local vs Remote
  - User-mode vs Kernel debuggers
- 
- credo : vs / gdb

In both Linux and Windows

- you can start a debugging session by either
  - starting a new process from the debugger
  - attaching a debugger to a running process
- each process can have at most one debugger attached

# Outline

- 1 Introduction
- 2 Linux and Windows Debuggers
- 3 Symbols and debug information
- 4 Controlling the execution
- 5 Backtraces

- Documentation: “Debugging with GDB”  
<https://sourceware.org/gdb/download/onlinedocs/>
- GEF <https://hugsy.github.io/gef/>
  - and the extras: <https://hugsy.github.io/gef-extras/>
  - My cheatsheet for GDB+GEF: [https://github.com/zxgio/gdb\\_gef-cheatsheet](https://github.com/zxgio/gdb_gef-cheatsheet)
- generally solid; however, if you encounter strange behaviours, try to compile a newer version, e.g. x.y, from sources:
  - `wget https://ftp.gnu.org/gnu/gdb/gdb-x.y.tar.xz`
  - install Python development headers (in Ubuntu: `python3-dev`)
  - extract the archive, create a build directory, and then invoke configure from it:
    - `where-gdb-has-been-extracted/configure --prefix=$HOME/bin/gdbx.y --with-python=/usr/bin/python3`
    - `make && make install`
- ret-sync <https://github.com/bootleg/ret-sync>



## Windows: x64dbg

**x64dbg** <https://x64dbg.com/>, that can be enhanced with plugins; e.g.

- **ret-sync** <https://github.com/bootleg/ret-sync>
- **xAnalyzer** <https://github.com/ThunderCls/xAnalyzer>
- **ScyllaHide** <https://github.com/x64dbg/ScyllaHide>
- ...

WinDbg can be used for (local) kernel-debugging too

# Outline

- 1 Introduction
- 2 Linux and Windows Debuggers
- 3 Symbols and debug information**
- 4 Controlling the execution
- 5 Backtraces

# Symbols and debug information

Programs and libraries can include

- **Symbols**, mapping names to memory addresses. For instance, they allow you to find in which function the *instruction-pointer* is, ...
- Full **debug information**, that allow a debugger to match machine code to its corresponding source-level constructs

Released programs typically don't, but they rely on standard libraries. ..

# Symbols in Linux

Libc symbols are distributed separately

- in Ubuntu, `libc6-dbg` and `libc6-dbg:i386`
- in `gdb` (or `~/.gdbinit`) use:  
`set debug-file-directory /usr/lib/debug`  
then,
- `info address symbol` shows where data for symbol is stored
- `info symbol addr` prints the name of symbol stored at *addr*

You can also use `addr2line(1)` to read symbol information

Compiling with `-g/-ggdb` adds **debug information** using DWARF [Eag12]  
(<https://dwarfstd.org/>)

- to dump DWARF information, `dwarfdump`; e.g. `-1` prints the association between PCs and source lines  
→ `c-examples/buggy_factorial`

# Symbols in Windows

Symbol information can be downloaded from a **symbol server**

- official one: <https://msdl.microsoft.com/download/symbols>
- you can set the environment value `_NT_SYMBOL_PATH` to something like:  
`srv*your-cache-path*server-url`; e.g.  
`setx _NT_SYMBOL_PATH ^`  
`srv*c:\sym*https://msdl.microsoft.com/download/symbols`
- x64dbg set the server in: **Options → Preferences → Misc**
- `symchk.exe` from Windows SDK, and 3<sup>rd</sup>-party utilities, can download symbols and store them in the cache

With MS's compiler, `/DEBUG` add **debugging information**:

- 1 The linker puts these information into a **program database (PDB)** file
- 2 The **executable/DLL** contains the **path** of the corresponding PDB
- 3 A **debugger** reads the **embedded name** and uses the PDB

# Outline

- 1 Introduction
- 2 Linux and Windows Debuggers
- 3 Symbols and debug information
- 4 Controlling the execution**
- 5 Backtraces

# Common features for controlling the execution

Debugger typically allow you to

- set **breakpoints**; two kinds:
  - **software** breakpoints (the default)
    - **Unlimited number**
    - On **execution only**
  - **hardware** breakpoints
    - **Limited number**, four on x86/64. Set by writing into *debug registers*:  
[https://en.wikipedia.org/wiki/X86\\_debug\\_register](https://en.wikipedia.org/wiki/X86_debug_register)  
These registers cannot be *directly* accessed by user-code
    - Can be on **Read**, **Write** or **Execute**
- **single-step**: the debugger stops after every (machine) instruction
  - **step-into** or **step-over** CALL instructions
- **step-out/finish/execute-till-return** functions
- ...

→ c-examples/buggy\_factorial

# Software breakpoints

On x86 the one byte opcode `0xCC` corresponds to instruction `INT3`, intended for calling the debug exception handler

In general,  $\text{int } n \leftrightarrow 0xcd\ n$  however,  $\text{int3} \leftrightarrow 0xcc$

To implement a (software) breakpoint at address  $\alpha$ , a debugger:

- ① saves the content of address  $\alpha$ :  $t \leftarrow [\alpha]$
- ② replaces such a byte with `0xcc`:  $[\alpha] \leftarrow 0xcc$

When an exception/signal is triggered, the debugger:

- ③ restores the original content:  $[\alpha] \leftarrow t$
- ④ decrements the instruction pointer:  $IP \leftarrow IP - 1$
- ⑤ allows the user to inspect/change the debugged process
- ⑥ if the breakpoint should persist:
  - executes a single-step
  - if  $IP = \alpha$  then go to (5), else go to (1)



## Single-step (on x86/64)

You could implement single-stepping with software breakpoints; *however* you'd need to

- decode the current instruction to find out its length
- handle instructions that jump to themselves, even indirectly (e.g. `JMP RAX`, when `RAX==RIP`)

Better approach: setting the **trap flag** in the flag register

- issues an `INT 1` after a single instruction, and
- reset itself

See “17.3.1.4 Single-Step Exception Condition” in the *System Programming Guide* by Intel for more details

# Outline

- 1 Introduction
- 2 Linux and Windows Debuggers
- 3 Symbols and debug information
- 4 Controlling the execution
- 5 Backtraces**

# Backtraces

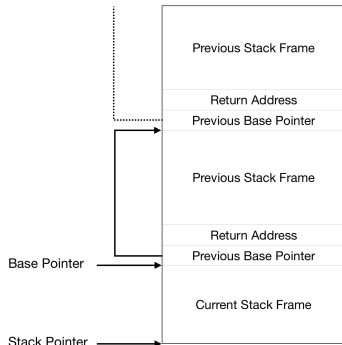
Backtraces/call-stacks help you understand how the execution got to a certain point, by listing all callers

Issues:

- who's my caller?
- where are the functions?

They can be tackled differently, depending whether we have:

- frame pointers
- debug information
- frame (exception handling) info



Taken from: [https://techno-coder.github.io/example\\_os/2018/06/04/A-stack-trace-for-your-OS.html](https://techno-coder.github.io/example_os/2018/06/04/A-stack-trace-for-your-OS.html)

Can be tricky; see, e.g., [Demystifying Thread Call Stack Spoofing](#) by Alessandro Magnosi  
<https://www.youtube.com/watch?v=d1-AuN2xsbg>

- 1 Set a breakpoint
  - at the beginning of function `fact`
  - with the condition that the argument value is 0

Then, start the program. . .

When the breakpoint is hit, check the backtrace/call-stack

- 2 Verify what happens when you set a breakpoint:

**Linux** You can read `/proc/pid/mem` with `dd/xxd`

**Windows** You can use *System Informer* (successor of Process Hacker)

<https://systeminformer.sourceforge.io/>

<https://github.com/winsidersss/systeminformer/>

- [Eag12] Michael J Eager.  
Introduction to the DWARF debugging format, 2012.