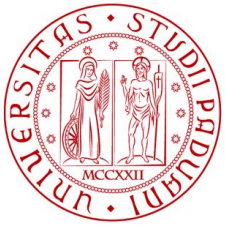
CyberSecurity: Principle and Practice

BSc Degree in Computer Science
2020-2021

Lesson 13: Debugging

Prof. Mauro Conti
Department of Mathematics
University of Padua
conti@math.unipd.it
http://www.math.unipd.it/~conti/

Teaching Assistants
Luca Pajola
pajola@math.unipd.it.
Pier Paolo Tricomi
pierpaolo.tricomi@phd.unipd.it



UNIVERSITÀ DEGLI STUDI DI PADOVA

SPRITZ SECURITY & PRIVACY RESEARCH GROUP

DIPARTIMENTO MATEMATICA

Disclaimer

All information presented here has the only purpose of teaching how reverse engineering works

Use your mad skillz only in CTFs or other situations in which you are legally allowed to do so

Do not hack the new Playstation. Or maybe do, but be prepared to get legal troubles 😊

Debugging under the hood

Fact:

- Debug is a strong tool that allows inspecting any process
  - Originally: For developers to solve problems
  - For attackers to exploit the vulnerabilities!

Consequence:

- Security Eng. developed techniques to automatically recognize whether a process is under debug
  - And possibly change behaviour based on this
    - (this is called “anti-debug”)

Debugging under the hood

Topics:

- How a debugger works in linux
- Some classic anti-debug techniques
- How to deactivate anti-debug protections

More theory here

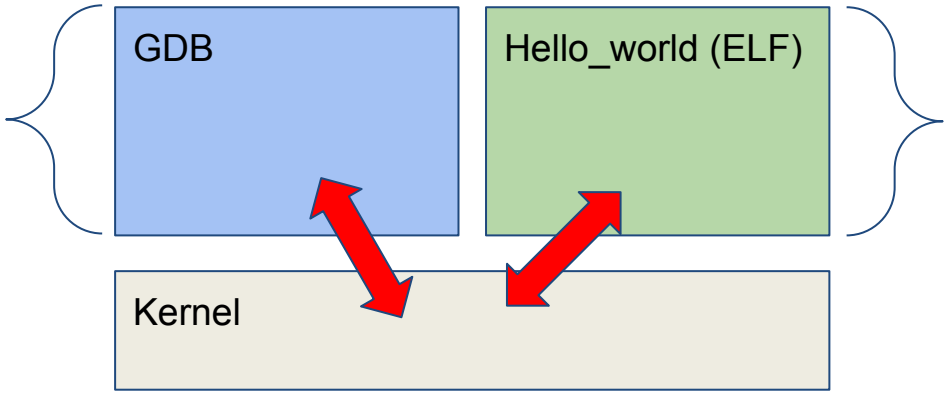
http://www.alexonlinux.com/how-debugger-works

Components and Interactions

Debugger:
(can be any process)

spawn a new process
or
attach existing process

can debug > 1 process



Debugee:
(can be any process)

can be debugged by
ONLY 1 process

Kernel:

Handles the interaction between debugger and debugee.

Provides tools to perform the debug: **ptrace()**

ptrace() aka our swiss-knife

ptrace() is a Linux syscall that allows a process (tracer/**debugger**) to inspect and control another process (tracee/**debugee**).
So, **debugger** uses **ptrace()** to control **debugee**, e.g., step-to-step, change variables, insert breakpoints.

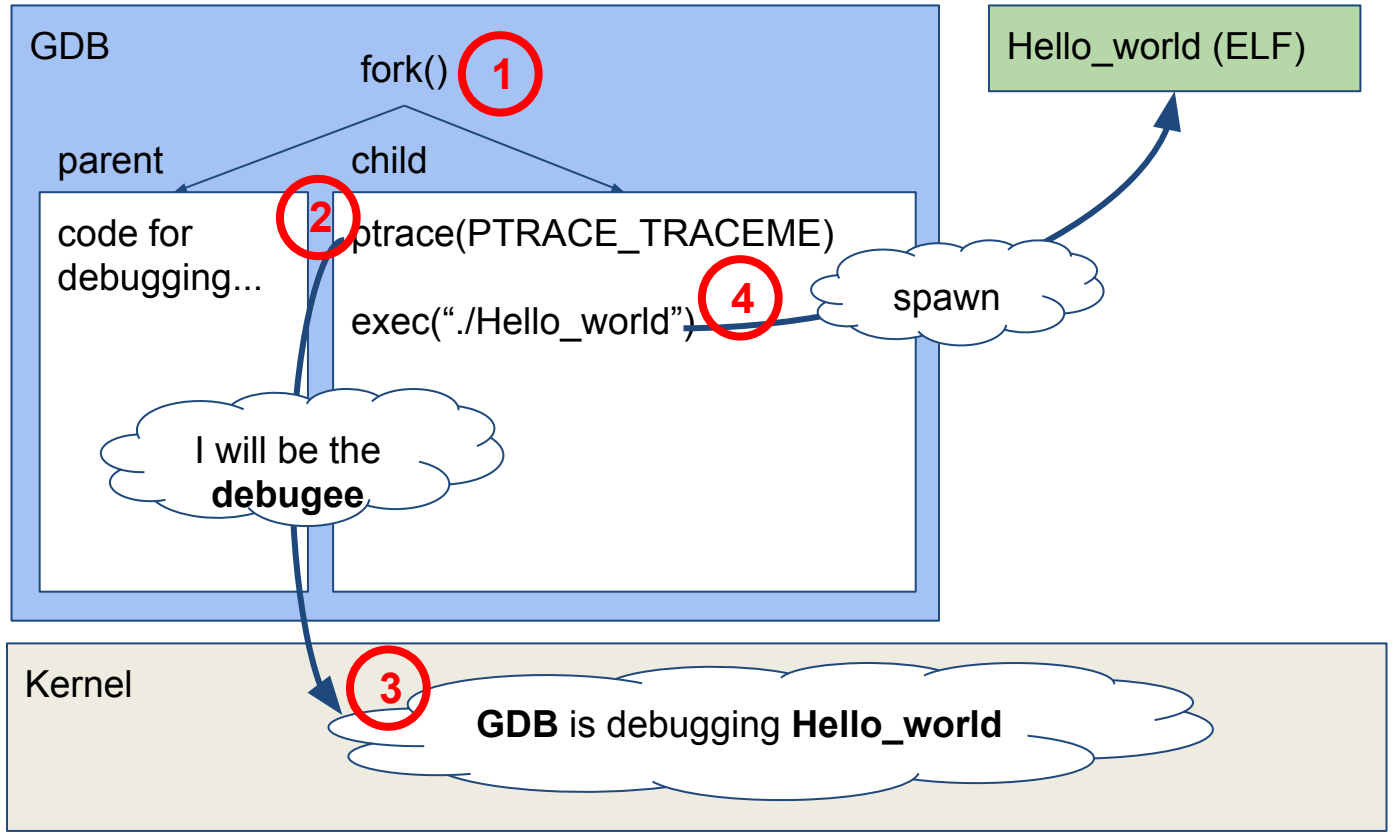
signature:
long **ptrace**(enum __ptrace_request request, pid_t pid, void *addr, void *data);

args:
- *request*: type of “action” that a **debugger** performs over **debugee**
  - (e.g., read from memory, write into memory, get/set registers value)
- *PID*: the process to attach to (can attach to itself)
- *addr* and *data* are used to transfer data from/to **debugee**

return:
- TL;TR; -1 if something wrong, otherwise depends by **request**

Source: http://man7.org/linux/man-pages/man2/ptrace.2.html

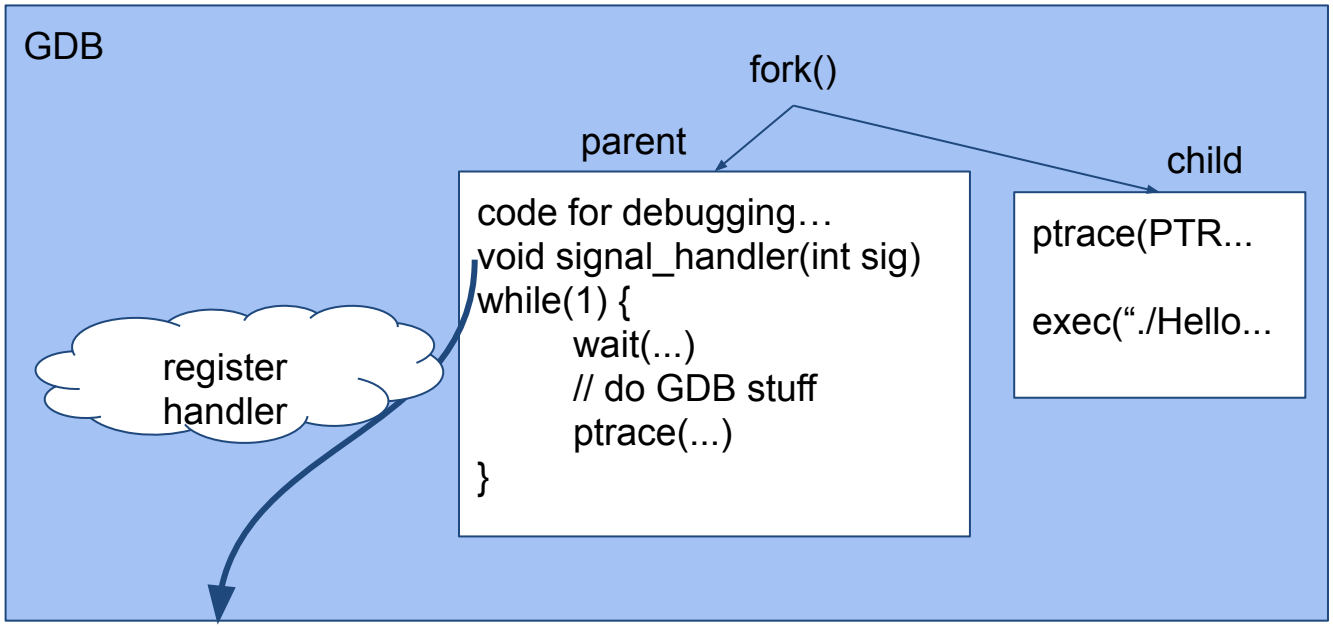
Spawning the debugee process



NOTE: ptrace() also allows to attach to a process already spawned

Registering signal handler

GDB registers to receive appropriate signals from the kernel… (triggered by the debuggee)

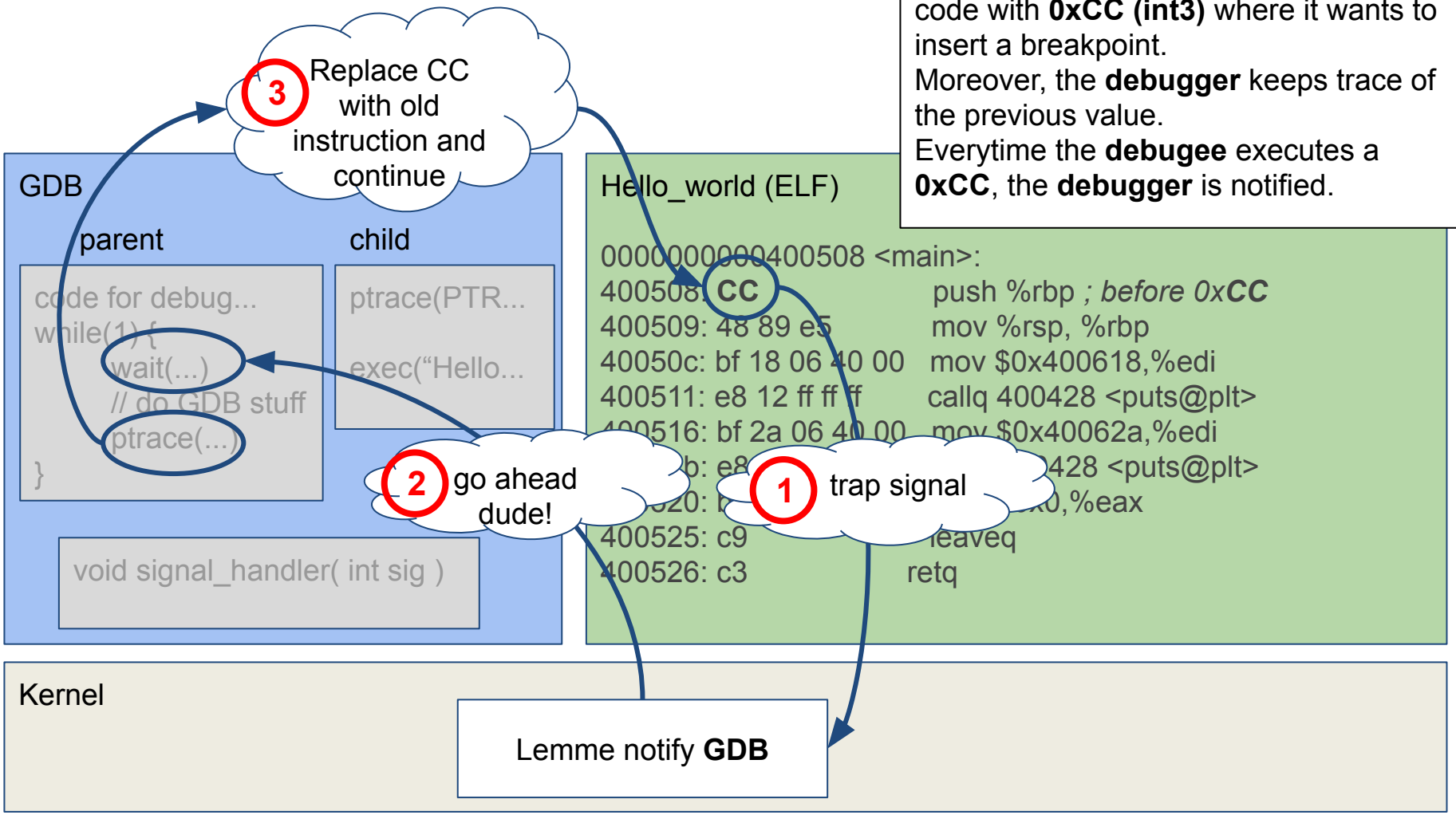


NOTE:
parent is notified by kernel whenever **Hello_world** encounters a breakpoint.

Gdb under the hood

Debugger - Debuggee Interactions

The **debugger** overwrites the assembly code with **0xCC (int3)** where it wants to insert a breakpoint.
Moreover, the **debugger** keeps trace of the previous value.
Everytime the **debugee** executes a **0xCC**, the **debugger** is notified.



Debugging - Quick recap

- the debugging process is mediated by the kernel
- **ptrace()** is the swiss-knife to debug processes
- a **debugger** can debug multiple processes
- a **debugee** can be debugged by only a single **debug**
- breakpoints are nothing but **0xCC** instructions injected in the **debuggee** process
- the **debugger** keeps track of the original values of **debugee**
- **strace(), GDB,** etc., relies on **ptrace()** to work

Anti-debugging

Again, I don't want somebody to touch my things!

Question: how the program "understand" if somebody is debugging it?

yeeeeeh, depends...

Anti-debugging: technique 1

check ptrace()

! Very Important and most common in challenges!

```
#include <stdio.h>
#include <sys/ptrace.h>
int main(int argc, char** argv) {
        if (ptrace(PTRACE_TRACEME, 0, NULL, NULL) == -1) {
                puts("there is already a debugger");
                return 1;
        }
        ptrace(PTRACE_DETACH, 0, NULL, NULL);
        puts("I am fine!");

        return 0;
}
```

If PTRACE_ME returns an error, someone is already debugging the program. Remember: only one debugger at time!

GDB env variable

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {

	if (getenv("LINES") || getenv("COLUMNS"))
		puts("there is already a debugger");
	else
		puts("I am fine!");

	return 0;
}
```

GDB creates detectable env variables.
If they exist, GDB is running!

GDB heap relocated

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
        //put var in bss (since var has no val)
        static unsigned char var_in_bss;
        //put var in heap and get addr
        unsigned char *probe = malloc(0x10);

        if (probe - &var_in_bss > 0x20000) {
                printf("I am fine\n");
        } else {
                printf("I got you GDB!\n");
        }
        return 0;
}
```

GDB relocates the heap to the end of the bss section (section containing variables declared but not assigned)

NOTE: 0x20000 is standard bss size, if less -> the heap has been relocated

GDB no-ASLR
(Address Space Layout Randomization)

GDB allocates libraries and text in specific addresses (disabling ASLR) that can be recognized (for example base address of ELF and shared libraries)

If these elements are found in those specific addresses, GDB is detected

Who is my parent?

```
pid_t parent = getppid();

link_target =  read("/proc/$parent/exe")
if (!strcmp(basename(link_target), "gdb"))
        res = RESULT_YES;
if (strstr(link_target, "lldb"))
        res = RESULT_YES;
if (!strcmp(basename(link_target), "strace"))
        res = RESULT_YES;
if (!strcmp(basename(link_target), "ltrace"))
        res = RESULT_YES;
```

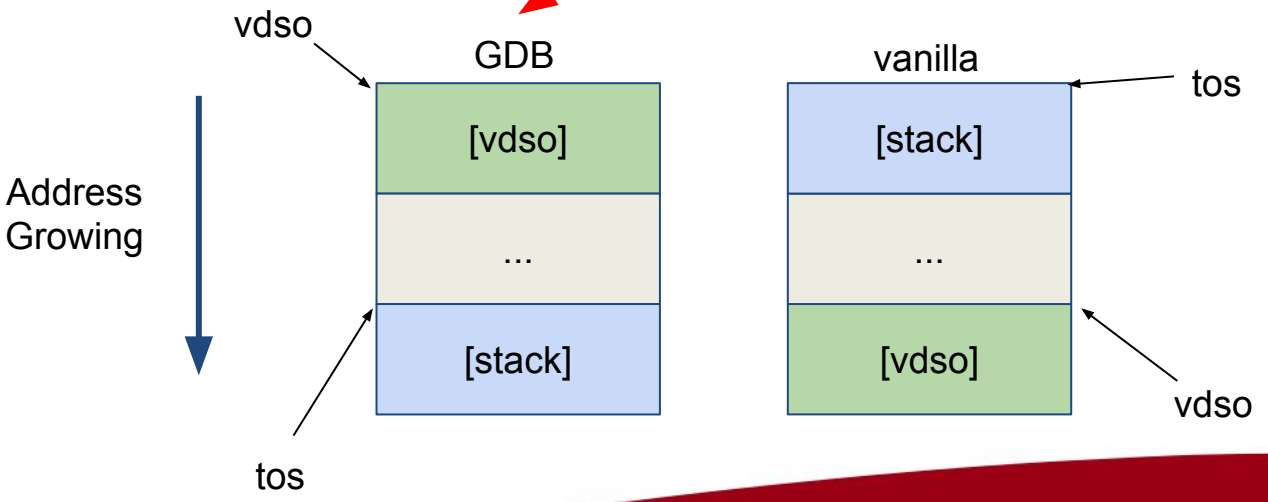
The parent process can be inspected
If my parent is GDB, lldb, strace, ltrace o any debugger, we can detect it!

VSDO (virtual dynamic shared object)

```
unsigned long tos; // top of stack
unsigned long vdso = getauxval(AT_SYSINFO_EHDR);

if ((unsigned long)&tos > vdso)
        return RESULT_YES;
else
        return RESULT_NO;
```

GDB moves the VDSO before the stack
Checking their addresses, we can detect GDB



How to bypass anti-debug stuffs?

Patch the binary!

Main steps:

1) look for implementation of anti-debug techniques
   - e.g., check for ptrace(), 0xCC, getenv(), …
   - check also non **main** thread:
     - **.init** and **.fini** (and other) sections might also contain implementations of anti-debug techniques (http://beefchunk.com/documentation/sys-programming/binary_formats/elf/elf_from_the_programmers_perspective/node3.html)

2) Use Hex Editor or radare to patch

Exercises

1) john galt is having some problems with his email again. But this time it's not his fault. Can you help him?

2) Using a debugging tool will be extremely useful on your missions. Can you run this program in gdb and find the flag?

3) The program detects if it is run into a debugger. Please remove the check!

4) The program runs several checks to detect a debugging environment. If running into gdb, every test should FAIL. Patch the program to obtain PASS in every check even when running into GDB

Questions? Feedback? Suggestions?