Strace

Code can be found here: https://github.com/c2003-tamu/413

Environment Setup

- Clone git repository
- Ensure strace and gdb and installed

1.bin

When we run strace on 1.bin, we get the following:

```
cade@cade-ThinkPad-T480s:~/Desktop/spring2025/csce413/413/strace$ strace ./1.bin
execve("./1.bin", ["./1.bin"], 0x7ffd13b4d380 /* 51 vars */) = 0
brk(NULL)                               = 0x27471000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x70b14050a000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=68179, ...}) = 0
mmap(NULL, 68179, PROT_READ, MAP_PRIVATE, 3, 0) = 0x70b1404f9000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\220\243\2\0\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
fstat(3, {st_mode=S_IFREG|0755, st_size=2125328, ...}) = 0
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
mmap(NULL, 2170256, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x70b140200000
mmap(0x70b140228000, 1605632, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x70b140228000
mmap(0x70b1403b0000, 323584, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b0000) = 0x70b1403b0000
mmap(0x70b1403ff000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1fe000) = 0x70b1403ff000
mmap(0x70b140405000, 52624, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x70b140405000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x70b1404f6000
arch_prctl(ARCH_SET_FS, 0x70b1404f6740) = 0
set_tid_address(0x70b1404f6a10)         = 42185
set_robust_list(0x70b1404f6a20, 24)     = 0
rseq(0x70b1404f7060, 0x20, 0, 0x53053053) = 0
mprotect(0x70b1403ff000, 16384, PROT_READ) = 0
mprotect(0x600000, 4096, PROT_READ)     = 0
mprotect(0x70b140548000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x70b1404f9000, 68179)           = 0
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x70b1404f6a10) = 42186
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x1), ...}) = 0
getrandom(World
"\x74\x9f\xdd\xcd\xe2\xbd\xc5\x77", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x27471000
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED, si_pid=42186, si_uid=1000, si_status=0, si_utime=0, si_stime=0} --
brk(0x27492000)                         = 0x27492000
write(1, "Hello\n", 6Hello
)                 = 6
exit_group(0)                           = ?
+++ exited with 0 +++
```

From this output, we can see that after all setup syscalls, we clone the current process, then “Hello\n” is written to stdout. If we look a bit closer at the line that starts with getrandom, we can see a rogue “World\n” that is clearly out of place in this dump. In order to see more in depth information about the program, let’s use gdb.

```
cade@cade-ThinkPad-T480s:~/Desktop/spring2025/csce413/413/strace$ gdb 1.bin
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from 1.bin...
(No debugging symbols found in 1.bin)
(gdb) b main
Breakpoint 1 at 0x40056a
(gdb) r
Starting program: /home/cade/Desktop/spring2025/csce413/413/strace/1.bin
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x000000000040056a in main ()
(gdb) disassemble
Dump of assembler code for function main:
   0x0000000000400566 <+0>:     push   %rbp
   0x0000000000400567 <+1>:     mov    %rsp,%rbp
=> 0x000000000040056a <+4>:     sub    $0x10,%rsp
   0x000000000040056e <+8>:     call   0x400450 <fork@plt>
   0x0000000000400573 <+13>:    mov    %eax,-0x4(%rbp)
   0x0000000000400576 <+16>:    cmpl   $0x0,-0x4(%rbp)
   0x000000000040057a <+20>:    jne    0x400588 <main+34>
   0x000000000040057c <+22>:    mov    $0x400624,%edi
   0x0000000000400581 <+27>:    call   0x400430 <puts@plt>
   0x0000000000400586 <+32>:    jmp    0x400592 <main+44>
   0x0000000000400588 <+34>:    mov    $0x40062a,%edi
   0x000000000040058d <+39>:    call   0x400430 <puts@plt>
   0x0000000000400592 <+44>:    mov    $0x0,%eax
   0x0000000000400597 <+49>:    leave
   0x0000000000400598 <+50>:    ret
End of assembler dump.
(gdb)
```

As seen in the assembly above, we are forking the process and then conditionally putting either “Hello\n” or “World\n” to stdout based on if we are in the parent or not. In order to tell which the parent is responsible for, let’s put a few breakpoints and see what is put to stdout from the child process.

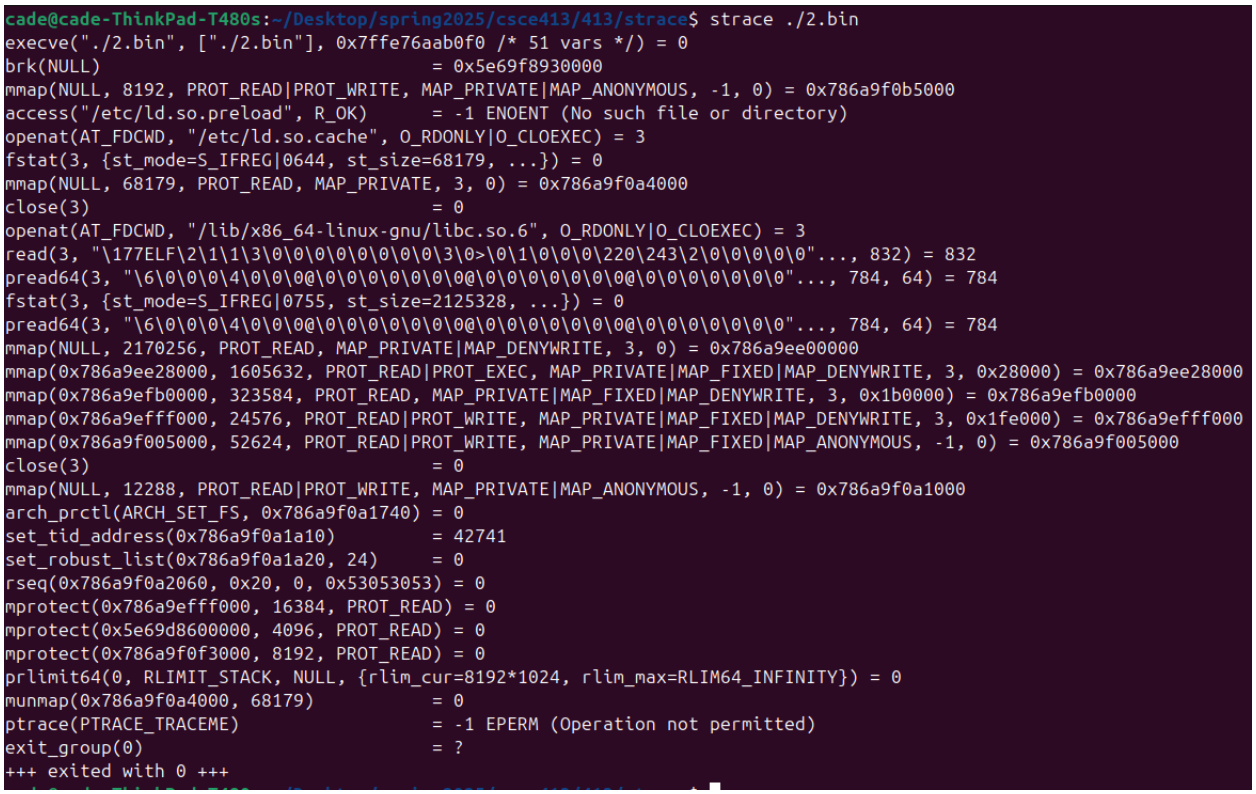
```
Breakpoint 1, 0x000000000040056a in main ()
(gdb) disassemble
Dump of assembler code for function main:
   0x0000000000400566 <+0>:     push   %rbp
   0x0000000000400567 <+1>:     mov    %rsp,%rbp
=> 0x000000000040056a <+4>:     sub    $0x10,%rsp
   0x000000000040056e <+8>:     call   0x400450 <fork@plt>
   0x0000000000400573 <+13>:    mov    %eax,-0x4(%rbp)
   0x0000000000400576 <+16>:    cmpl   $0x0,-0x4(%rbp)
   0x000000000040057a <+20>:    jne    0x400588 <main+34>
   0x000000000040057c <+22>:    mov    $0x400624,%edi
   0x0000000000400581 <+27>:    call   0x400430 <puts@plt>
   0x0000000000400586 <+32>:    jmp    0x400592 <main+44>
   0x0000000000400588 <+34>:    mov    $0x40062a,%edi
   0x000000000040058d <+39>:    call   0x400430 <puts@plt>
   0x0000000000400592 <+44>:    mov    $0x0,%eax
   0x0000000000400597 <+49>:    leave
   0x0000000000400598 <+50>:    ret
End of assembler dump.
(gdb) b *0x000000000040057a
Breakpoint 2 at 0x40057a
(gdb) b *0x000000000040057c
Breakpoint 3 at 0x40057c
(gdb) b *0x0000000000400588
Breakpoint 4 at 0x400588
(gdb) c
Continuing.
[Detaching after fork from child process 42596]
World

Breakpoint 2, 0x000000000040057a in main ()
(gdb)
```

Based on the output above, since the default behavior of gdb is to follow the parent process, we can see that the child is responsible for putting “World\n” to stdout, so the parent must be in charge of putting “Hello\n” to stdout. This is reflected when we run the binary outside of gdb, as “Hello\n” comes before “World\n”, as the time taken to spin up the new thread for the child takes longer than the execution of puts() in the parent.

```
cade@cade-ThinkPad-T480s:~/Desktop/spring2025/csce413/413/strace$ ./1.bin
Hello
World
cade@cade-ThinkPad-T480s:~/Desktop/spring2025/csce413/413/strace$
```

2.bin

Running strace on the file 2.bin, we get the following output:



```
cade@cade-ThinkPad-T480s:~/Desktop/spring2025/csce413/413/strace$ strace ./2.bin
execve("./2.bin", ["./2.bin"], 0x7ffe76aab0f0 /* 51 vars */) = 0
brk(NULL)                               = 0x5e69f8930000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x786a9f0b5000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=68179, ...}) = 0
mmap(NULL, 68179, PROT_READ, MAP_PRIVATE, 3, 0) = 0x786a9f0a4000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\220\243\2\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
fstat(3, {st_mode=S_IFREG|0755, st_size=2125328, ...}) = 0
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
mmap(NULL, 2170256, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x786a9ee00000
mmap(0x786a9ee28000, 1605632, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x786a9ee28000
mmap(0x786a9efb0000, 323584, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b0000) = 0x786a9efb0000
mmap(0x786a9efff000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1fe000) = 0x786a9efff000
mmap(0x786a9f005000, 52624, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x786a9f005000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x786a9f0a1000
arch_prctl(ARCH_SET_FS, 0x786a9f0a1740) = 0
set_tid_address(0x786a9f0a1a10)         = 42741
set_robust_list(0x786a9f0a1a20, 24)     = 0
rseq(0x786a9f0a2060, 0x20, 0, 0x53053053) = 0
mprotect(0x786a9efff000, 16384, PROT_READ) = 0
mprotect(0x5e69d8600000, 4096, PROT_READ) = 0
mprotect(0x786a9f0f3000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x786a9f0a4000, 68179)           = 0
ptrace(PTRACE_TRACEME)                  = -1 EPERM (Operation not permitted)
exit_group(0)                           = ?
+++ exited with 0 +++
```

This output is odd, as the only call outside of initialization is ptrace(PTRACE_TRACEME). After researching this strategy (huge thanks to https://hkopp.github.io/2023/08/the-ptrace-anti-re-trick), I found that this is a strategy commonly used to prevent debugging. To get around this, we can do a couple things in gdb:

1. To see the assembly of the main function, we will put a breakpoint in main and then run the program, doing this we get the address of the instruction where we are checking if ptrace succeeds or not.

```
Breakpoint 1, 0x00005555554007ce in main ()
(gdb) disassemble
Dump of assembler code for function main:
   0x00005555554007ca <+0>:     push   %rbp
   0x00005555554007cb <+1>:     mov    %rsp,%rbp
=> 0x00005555554007ce <+4>:     sub    $0x10,%rsp
   0x00005555554007d2 <+8>:     mov    %fs:0x28,%rax
   0x00005555554007db <+17>:    mov    %rax,-0x8(%rbp)
   0x00005555554007df <+21>:    xor    %eax,%eax
   0x00005555554007e1 <+23>:    mov    $0x0,%edi
   0x00005555554007e6 <+28>:    mov    $0x0,%eax
   0x00005555554007eb <+33>:    call   0x555555400670 <ptrace@plt>
   0x00005555554007f0 <+38>:    cmp    $0xffffffffffffffff,%rax
   0x00005555554007f4 <+42>:    jne    0x555555400800 <main+54>
   0x00005555554007f6 <+44>:    mov    $0x0,%edi
   0x00005555554007fb <+49>:    call   0x555555400680 <exit@plt>
   0x0000555555400800 <+54>:    call   0x5555554006a0 <fork@plt>
   0x0000555555400805 <+59>:    mov    %eax,-0xc(%rbp)
   0x0000555555400808 <+62>:    cmpl   $0x0,-0xc(%rbp)
   0x000055555540080c <+66>:    jne    0x55555540081c <main+82>
   0x000055555540080e <+68>:    lea    0xbf(%rip),%rdi        # 0x5555554008d4
   0x0000555555400815 <+75>:    call   0x555555400650 <puts@plt>
   0x000055555540081a <+80>:    jmp    0x555555400828 <main+94>
   0x000055555540081c <+82>:    lea    -0x10(%rbp),%rax
   0x0000555555400820 <+86>:    mov    %rax,%rdi
   0x0000555555400823 <+89>:    call   0x555555400690 <wait@plt>
   0x0000555555400828 <+94>:    mov    $0x0,%eax
   0x000055555540082d <+99>:    mov    -0x8(%rbp),%rdx
   0x0000555555400831 <+103>:   xor    %fs:0x28,%rdx
   0x000055555540083a <+112>:   je     0x555555400841 <main+119>
   0x000055555540083c <+114>:   call   0x555555400660 <__stack_chk_fail@plt>
   0x0000555555400841 <+119>:   leave
   0x0000555555400842 <+120>:   ret
End of assembler dump.
(gdb) b *0x00005555554007f0
Breakpoint 2 at 0x5555554007f0
(gdb) c
Continuing.

Breakpoint 2, 0x00005555554007f0 in main ()
```

2. If we put a breakpoint at the address that contains the success check for ptrace (0x00005555554007f0), we can control whether the system thinks the call succeeds or not by changing the rax register to 0.

```
Breakpoint 2, 0x000055555554007f0 in main ()
(gdb) info registers
rax            0xffffffffffffffff  -1
rbx            0x7fffffffddd8      140737488346584
rcx            0x7ffff7d260fd      140737351147773
rdx            0x0                 0
rsi            0xffffddd8          4294958552
rdi            0x0                 0
rbp            0x7fffffffdcb0      0x7fffffffdcb0
rsp            0x7fffffffdca0      0x7fffffffdca0
r8             0xffffffff          4294967295
r9             0x7ffff7fca380      140737353917312
r10            0x555555400850      93824990840912
r11            0x286               646
r12            0x1                 1
r13            0x0                 0
r14            0x0                 0
r15            0x7ffff7ffd000      140737354125312
rip            0x5555554007f0      0x5555554007f0 <main+38>
eflags         0x246               [ PF ZF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
fs_base        0x7ffff7fa9740      140737353783104
gs_base        0x0                 0
(gdb) set $rax = 0
```

3. Once we successfully trick the system into thinking that our ptrace() syscall succeeded, we can enter the area of the code that we previously couldn’t.

```
(gdb) c
Continuing.

Breakpoint 3, 0x0000555555400800 in main ()
(gdb) disassemble
Dump of assembler code for function main:
   0x00005555554007ca <+0>:     push   %rbp
   0x00005555554007cb <+1>:     mov    %rsp,%rbp
   0x00005555554007ce <+4>:     sub    $0x10,%rsp
   0x00005555554007d2 <+8>:     mov    %fs:0x28,%rax
   0x00005555554007db <+17>:    mov    %rax,-0x8(%rbp)
   0x00005555554007df <+21>:    xor    %eax,%eax
   0x00005555554007e1 <+23>:    mov    $0x0,%edi
   0x00005555554007e6 <+28>:    mov    $0x0,%eax
   0x00005555554007eb <+33>:    call   0x555555400670 <ptrace@plt>
   0x00005555554007f0 <+38>:    cmp    $0xffffffffffffffff,%rax
   0x00005555554007f4 <+42>:    jne    0x555555400800 <main+54>
   0x00005555554007f6 <+44>:    mov    $0x0,%edi
   0x00005555554007fb <+49>:    call   0x555555400680 <exit@plt>
=> 0x0000555555400800 <+54>:    call   0x5555554006a0 <fork@plt>
   0x0000555555400805 <+59>:    mov    %eax,-0xc(%rbp)
   0x0000555555400808 <+62>:    cmpl   $0x0,-0xc(%rbp)
   0x000055555540080c <+66>:    jne    0x55555540081c <main+82>
   0x000055555540080e <+68>:    lea    0xbf(%rip),%rdi        # 0x5555554008d4
   0x0000555555400815 <+75>:    call   0x555555400650 <puts@plt>
   0x000055555540081a <+80>:    jmp    0x555555400828 <main+94>
   0x000055555540081c <+82>:    lea    -0x10(%rbp),%rax
   0x0000555555400820 <+86>:    mov    %rax,%rdi
   0x0000555555400823 <+89>:    call   0x555555400690 <wait@plt>
   0x0000555555400828 <+94>:    mov    $0x0,%eax
   0x000055555540082d <+99>:    mov    -0x8(%rbp),%rdx
   0x0000555555400831 <+103>:   xor    %fs:0x28,%rdx
   0x000055555540083a <+112>:   je     0x555555400841 <main+119>
   0x000055555540083c <+114>:   call   0x555555400660 <__stack_chk_fail@plt>
   0x0000555555400841 <+119>:   leave
   0x0000555555400842 <+120>:   ret
End of assembler dump.
(gdb)
```

Inspecting the rest of this assembly looks a bit odd, as it calls fork() then conditionally calls puts() based on if the program is in its child or parent thread, then calls wait() presumably for the parent to wait for the child to exit. This all seems in order when we step through it in gdb, as we can verify that the both the child and parent threads get properly reaped upon the completion of the program:

```
(gdb) b *0x000055555540080c
Breakpoint 4 at 0x55555540080c
(gdb) b *0x000055555540080e
Breakpoint 5 at 0x55555540080e
(gdb) b *0x000055555540081c
Breakpoint 6 at 0x55555540081c
(gdb) c
Continuing.
[Detaching after fork from child process 44394]
I'm a malware

Breakpoint 4, 0x000055555540080c in main ()
(gdb) info inferiors
  Num  Description       Connection           Executable
* 1    process 44275     1 (native)           /home/cade/Desktop/spring2025/csce413/413/strace/2.bin
```

```
(gdb) disassemble
Dump of assembler code for function main:
   0x00005555554007ca <+0>:     push   %rbp
   0x00005555554007cb <+1>:     mov    %rsp,%rbp
   0x00005555554007ce <+4>:     sub    $0x10,%rsp
   0x00005555554007d2 <+8>:     mov    %fs:0x28,%rax
   0x00005555554007db <+17>:    mov    %rax,-0x8(%rbp)
   0x00005555554007df <+21>:    xor    %eax,%eax
   0x00005555554007e1 <+23>:    mov    $0x0,%edi
   0x00005555554007e6 <+28>:    mov    $0x0,%eax
   0x00005555554007eb <+33>:    call   0x555555400670 <ptrace@plt>
   0x00005555554007f0 <+38>:    cmp    $0xffffffffffffffff,%rax
   0x00005555554007f4 <+42>:    jne    0x555555400800 <main+54>
   0x00005555554007f6 <+44>:    mov    $0x0,%edi
   0x00005555554007fb <+49>:    call   0x555555400680 <exit@plt>
   0x0000555555400800 <+54>:    call   0x5555554006a0 <fork@plt>
   0x0000555555400805 <+59>:    mov    %eax,-0xc(%rbp)
   0x0000555555400808 <+62>:    cmpl   $0x0,-0xc(%rbp)
=> 0x000055555540080c <+66>:    jne    0x55555540081c <main+82>
   0x000055555540080e <+68>:    lea    0xbf(%rip),%rdi        # 0x5555554008d4
   0x0000555555400815 <+75>:    call   0x555555400650 <puts@plt>
   0x000055555540081a <+80>:    jmp    0x555555400828 <main+94>
   0x000055555540081c <+82>:    lea    -0x10(%rbp),%rax
   0x0000555555400820 <+86>:    mov    %rax,%rdi
   0x0000555555400823 <+89>:    call   0x555555400690 <wait@plt>
   0x0000555555400828 <+94>:    mov    $0x0,%eax
   0x000055555540082d <+99>:    mov    -0x8(%rbp),%rdx
   0x0000555555400831 <+103>:   xor    %fs:0x28,%rdx
   0x000055555540083a <+112>:   je     0x555555400841 <main+119>
   0x000055555540083c <+114>:   call   0x555555400660 <__stack_chk_fail@plt>
   0x0000555555400841 <+119>:   leave
   0x0000555555400842 <+120>:   ret
End of assembler dump.
(gdb) c
Continuing.

Breakpoint 6, 0x000055555540081c in main ()
(gdb) c
Continuing.
[Inferior 1 (process 44275) exited normally]
(gdb) info inferiors
  Num  Description       Connection           Executable
* 1    <null>                                 /home/cade/Desktop/spring2025/csce413/413/strace/2.bin
(gdb)
```

But this behavior is not what is observed in reality, as when we run the binary directly, we get the following behavior:

```
cade@cade-ThinkPad-T480s:~/Desktop/spring2025/csce413/413/strace$ ./2.bin
I'm a malware

[1]+  Stopped                 ./2.bin
cade@cade-ThinkPad-T480s:~/Desktop/spring2025/csce413/413/strace$ ps -a
    PID TTY          TIME CMD
   3289 tty2     00:00:00 gnome-session-b
  44916 pts/1    00:00:00 2.bin
  44919 pts/1    00:00:00 ps
cade@cade-ThinkPad-T480s:~/Desktop/spring2025/csce413/413/strace$ bg
[1]+ ./2.bin &
cade@cade-ThinkPad-T480s:~/Desktop/spring2025/csce413/413/strace$
```

Based on this output, the child process is clearly NOT cleaned up or properly merged back with the parent thread. Seeing as the only difference between my execution of the binary outside of gdb is the fact that the ptrace() call was not faked, it has to be an exploit in how the ptrace(PTRACE_TRACEME) syscall is handled by the kernel. Admittedly, I wasn't quite able to figure this out, but my best guess is that ptrace(PTRACE_TRACEME) hooks the binary and prevents the status of the thread from being available to the parent. In order for this to work, the parent must use the WNOHANG option when it calls wait() (source: https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/sys_wait.h.html)