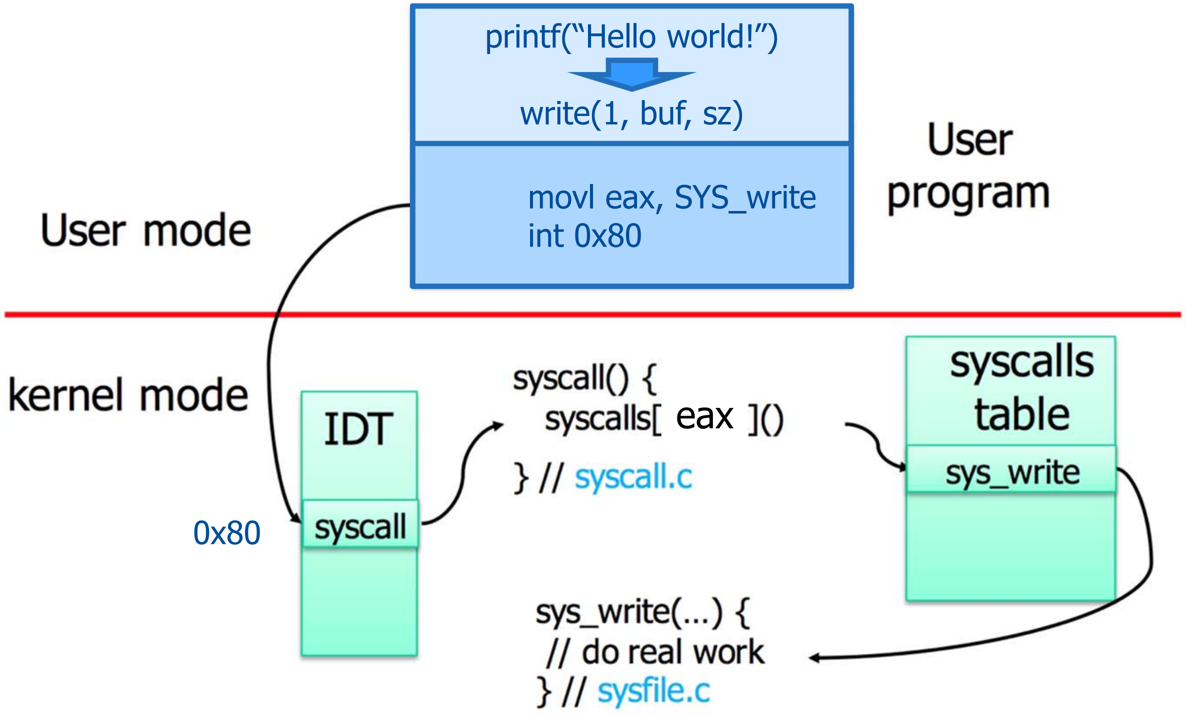
FITNESS

FAULT AND INTRUSION TOLERANT NETWORKED SYSTEMS

Anti-debugging techniques



SYSTEM CALL

An interface between application and OS kernel

□ Linux int 0x80 (syscall x86_64)

□ Win int 0x2e

eax = # syscall in syscalls table (e.g. 1 = write)

Gp registers host parameters (syscall dependents)

FITNESS

FAULT AND INTRUSION TOLERANT NETWORKED SYSTEMS

Understanding debuggers

SW BREAKPOINTS

□ Break @push rbp

```
push rbp
mov  ebp,esp
mov  rax,1
syscall (x86_64)
mov  rdi,1
mov  rsi,Hello
mov  rdx,len_Hello
syscall
mov  esp, ebp
pop   rbp
```

SW BREAKPOINTS

- □ Break @push rbp

1. Target <- int 3

INT 3 (0xCC) is a syscall which generates a **SIGTRAP**

Note that with some assemblers (like NASM), `int 3` is `CD 03`, and you need to write `int3` (no space) to get the `0xCC` single-byte opcode.

~~push rbp~~ int 3
mov ebp,esp
mov rax,1
syscall (x86_64)
mov rdi,1
mov rsi,Hello
mov rdx,len_Hello
syscall
mov esp, ebp
pop rbp

SW BREAKPOINTS

□ Break @push rbp

1. Target <- int 3
2. When target executed (EIP=target) a **SIGTRAP raised**

EIP -> ~~push rbp~~ int 3

mov ebp,esp

mov rax,1

syscall (x86_64)

mov rdi,1

mov rsi,Hello

mov rdx,len_Hello

syscall

mov esp, ebp

pop rbp

SW BREAKPOINTS

- □ Break @push rbp

1. Target <- int 3
2. When target executed (EIP=target) a **SIGTRAP raised**
3. [EIP] substituted with original

```
EIP ->   push rbp ~~int 3~~
         mov  ebp,esp
         mov  rax,1
         syscall (x86_64)
         mov  rdi,1
         mov  rsi,Hello
         mov  rdx,len_Hello
         syscall
         mov  esp, ebp
         pop  rbp
```

SW BREAKPOINTS

□ Break @push rbp

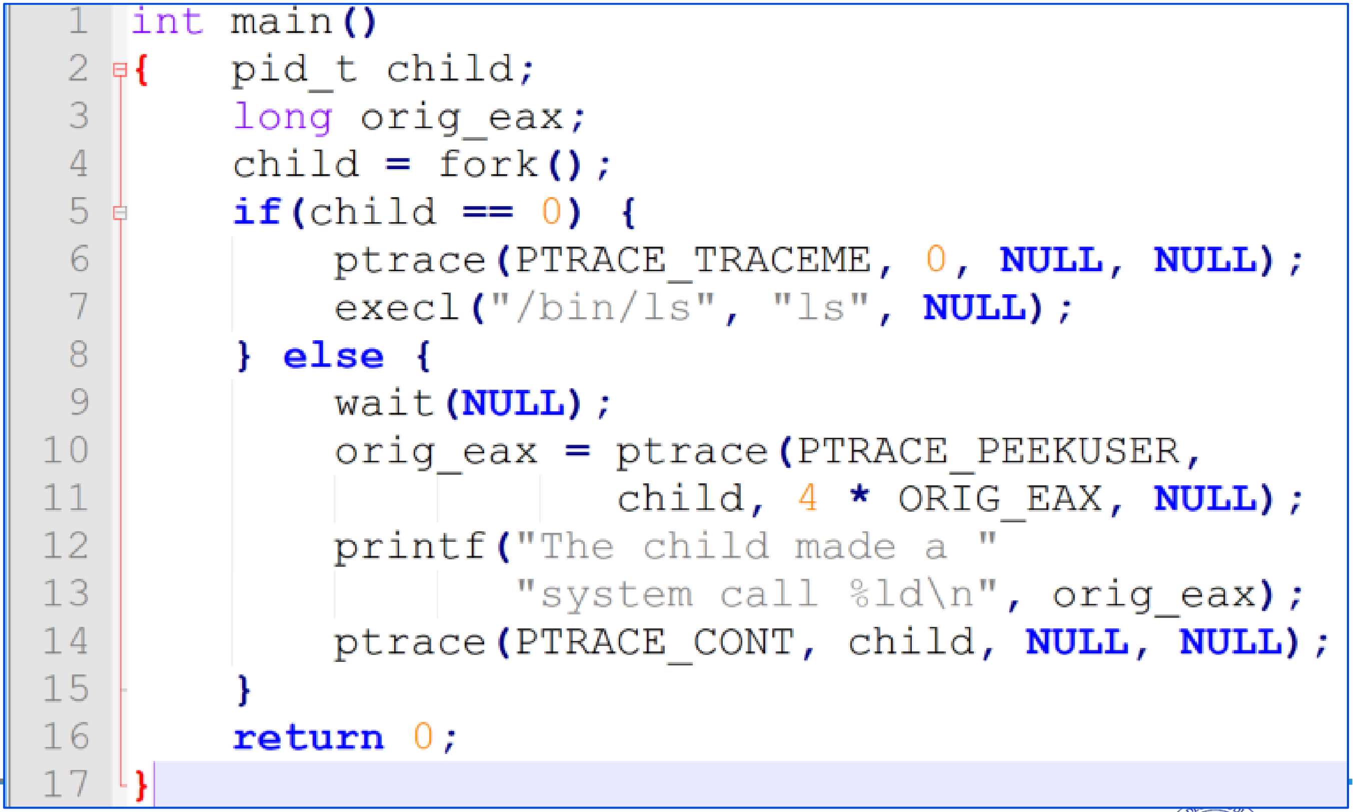
1. Target <- int 3
2. When target executed (EIP=target) a **SIGTRAP raised**
3. [EIP] substituted with original
4. Single step executed
5. and target newly prepared

~~push rbp~~ int 3
EIP -> mov ebp,esp
mov rax,1
syscall (x86_64)
mov rdi,1
mov rsi,Hello
mov rdx,len_Hello
syscall
mov esp, ebp
pop rbp

HOW THIS IS DONE: PTRACE (MAN PTRACE)

- □ The ptrace() system call provides a mean by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.

If a tracer is ptracing a tracee, when the tracee makes a syscall, it is stopped and a signal is sent to the tracer (keep it in mind: breakpoint are now int3)



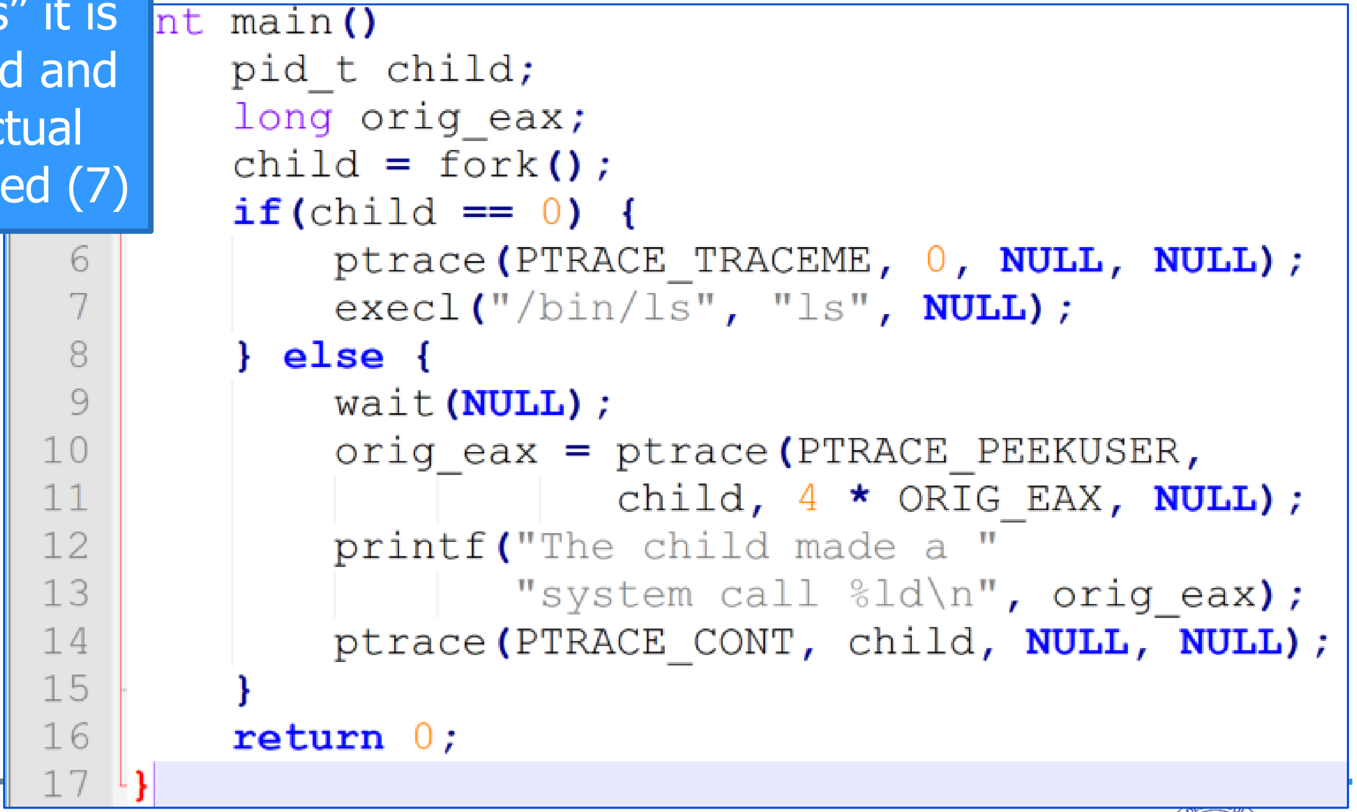
```c
int main()
{    pid_t child;
     long orig_eax;
     child = fork();
     if(child == 0) {
         ptrace(PTRACE_TRACEME, 0, NULL, NULL);
         execl("/bin/ls", "ls", NULL);
     } else {
         wait(NULL);
         orig_eax = ptrace(PTRACE_PEEKUSER,
                           child, 4 * ORIG_EAX, NULL);
         printf("The child made a "
                  "system call %ld\n", orig_eax);
         ptrace(PTRACE_CONT, child, NULL, NULL);
     }
     return 0;
}
```

How this is done: ptrace (man ptrace)

- The ptrace() system call provides a mean by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.

At (6) the child “states” it is willing being monitored and then executes the actual program to be monitored (7)

tracee makes a syscall, it is stopped and a signal is sent to the tracer (keep it in mind: breakpoint are now int3)



```
       nt main()
          pid_t child;
          long orig_eax;
          child = fork();
          if(child == 0) {
 6            ptrace(PTRACE_TRACEME, 0, NULL, NULL);
 7            execl("/bin/ls", "ls", NULL);
 8        } else {
 9            wait(NULL);
10            orig_eax = ptrace(PTRACE_PEEKUSER,
11                              child, 4 * ORIG_EAX, NULL);
12            printf("The child made a "
13                   "system call %ld\n", orig_eax);
14            ptrace(PTRACE_CONT, child, NULL, NULL);
15        }
16        return 0;
17   }
```

HOW THIS IS DONE: PTRACE (MAN PTRACE)

- □ The ptrace() system call provides a mean by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.

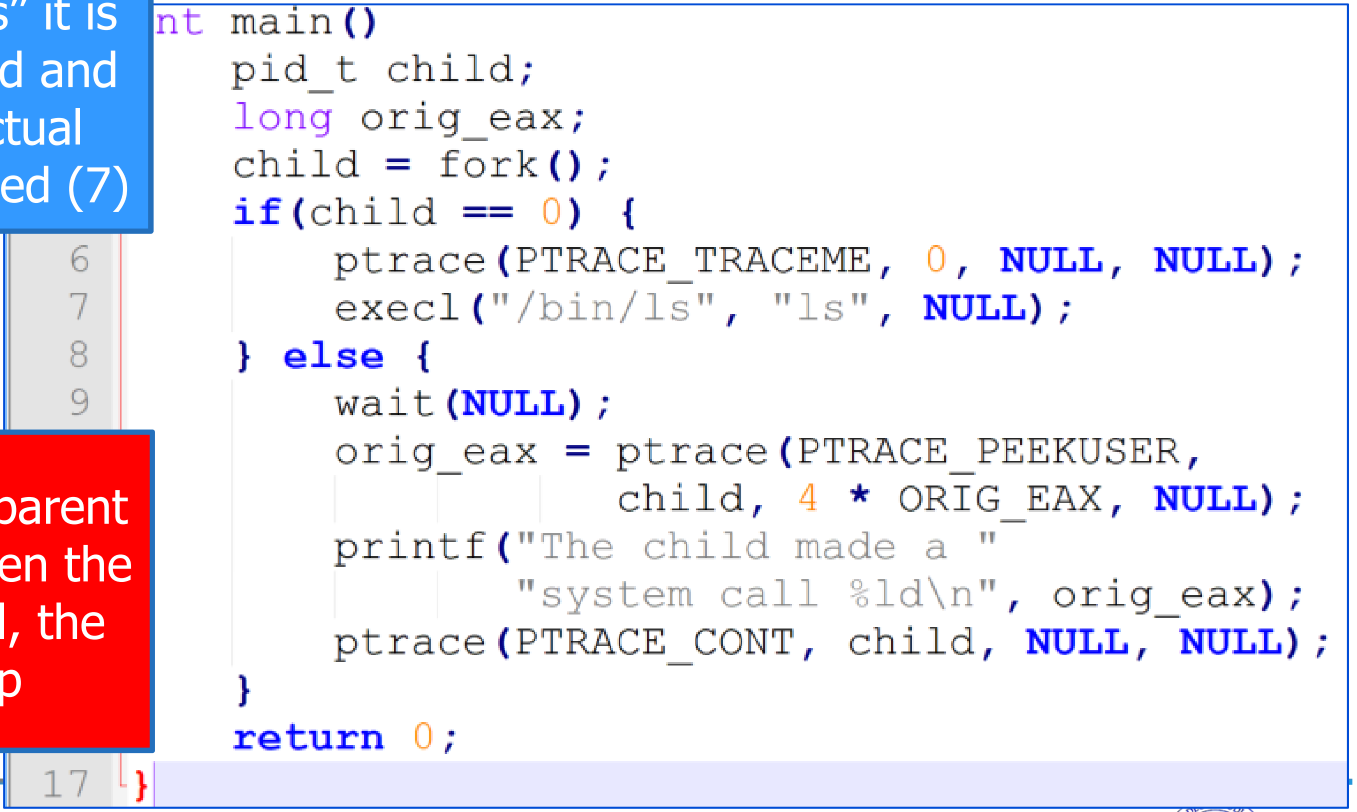
At (6) the child “states” it is willing being monitored and then executes the actual program to be monitored (7)

tracee makes a syscall,

it is stopped and a

signal is sent to the

In the meanwhile the parent invokes a wait (9). When the child invokes a syscall, the parent wakes it up



```
nt main()
    pid_t child;
    long orig_eax;
    child = fork();
    if(child == 0) {
 6      ptrace(PTRACE_TRACEME, 0, NULL, NULL);
 7      execl("/bin/ls", "ls", NULL);
 8  } else {
 9      wait(NULL);
        orig_eax = ptrace(PTRACE_PEEKUSER,
                          child, 4 * ORIG_EAX, NULL);
        printf("The child made a "
               "system call %ld\n", orig_eax);
        ptrace(PTRACE_CONT, child, NULL, NULL);
    }
    return 0;
17 }
```

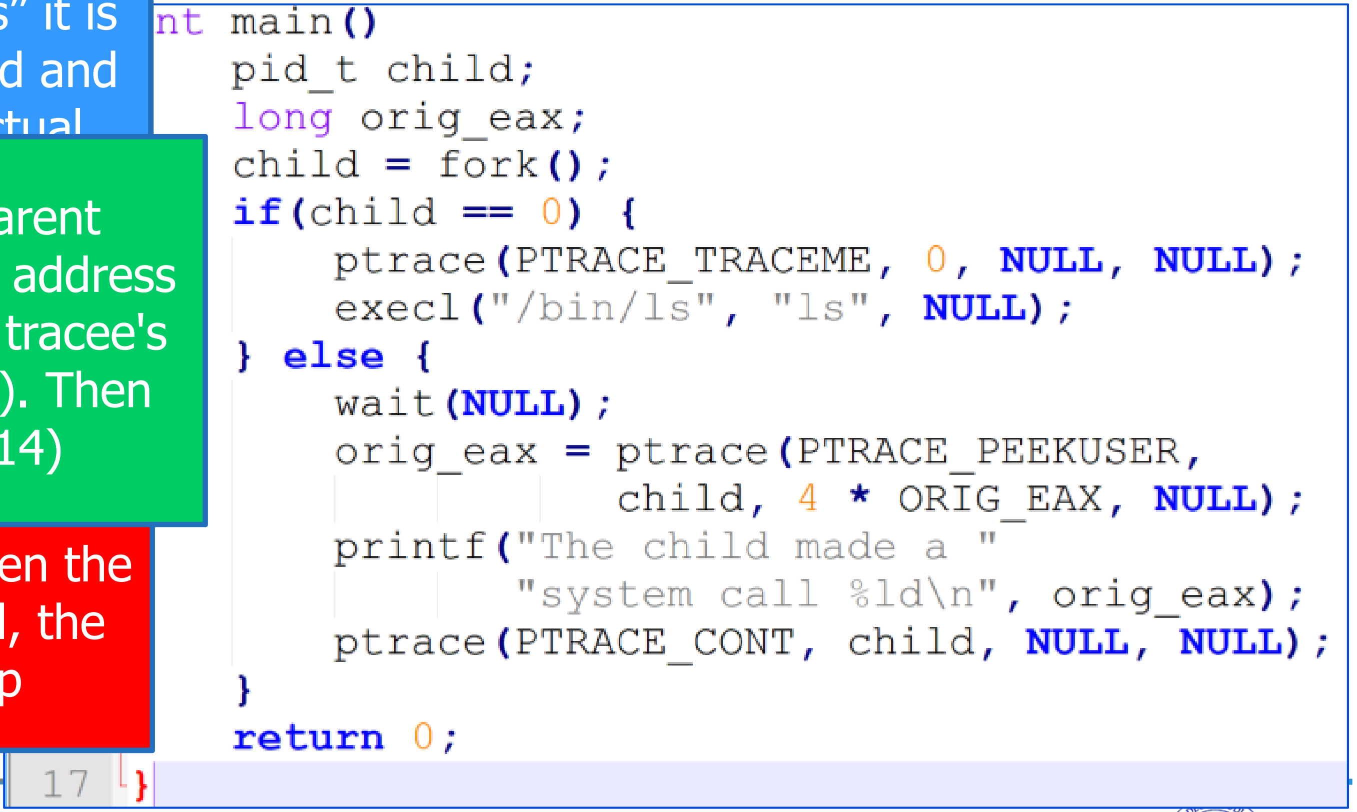
How this is done: ptrace (man ptrace)

□ The ptrace() system call provides a mean by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.

At (6) the child “states” it is willing being monitored and then executes the actual

Once “awake” the parent retrieves (10) a word at address $*ORIG_EAX, from the tracee's USER area (PEEKUSER). Then resume the child (14)

invokes a wait (9). When the child invokes a syscall, the parent wakes it up



```
nt main()
    pid_t child;
    long orig_eax;
    child = fork();
    if(child == 0) {
        ptrace(PTRACE_TRACEME, 0, NULL, NULL);
        execl("/bin/ls", "ls", NULL);
    } else {
        wait(NULL);
        orig_eax = ptrace(PTRACE_PEEKUSER,
                          child, 4 * ORIG_EAX, NULL);
        printf("The child made a "
                 "system call %ld\n", orig_eax);
        ptrace(PTRACE_CONT, child, NULL, NULL);
    }
    return 0;
17 }
```

FOPEN technique

FOPEN() TECHNIQUE…

```
FILE *fd = fopen("/tmp", "r");
if (fileno(fd) > 5)    {
   printf("I'm sorry GDB! You are not allowed!\n");
   exit(1);
}
fclose(fd);
```

□ FDs 0 (stdin), 1 (stdout) and 2 (stderr) are always opened...

□ gdb opens additional file descriptors (3,4,5) which are inherited ...

□ fineno(fd)>5 …but we never opened files before…gdb detected

0xCC technique

DETECTING BREAK POINTS: LOOK FOR INT3 OPCODE (0XCC)

```
void foo()
{
   printf("Hello\n");
}
int main()
{
  if ((*(volatile unsigned *)((unsigned)foo) & 0xff) == 0xcc)
  {
     printf("BREAKPOINT\n");
     exit(1);
  }
  foo();
}
```

DETECTING BREAK POINTS: LOOK FOR INT3 OPCODE (0XCC)

```
void foo()
{
   printf("Hello\n");
}
int main()
{
  if ((*(volatile unsigned *)((unsigned)foo) & 0xff) == 0xcc)
  {
     printf("BREAKPOINT\n");
     exit(1);
  }
  foo();
}
```

To escape this check, just use a near address as break point.

The real difficulty is “finding” the check that could silently stop the program being debugged:

1) Look for the breakpoint address in the assembly of the debugged program BUT it could be calculated

2) Checking for 0xCC in the code (…it could be a symptom)

DETECTING BREAK POINTS: LOOK FOR INT3 OPCODE (0XCC)

```
void foo()
{
   printf("Hello\n");
}
int main()
{
  if ((*(volatile unsigned*)((unsigned)foo) & 0xff) == 0xcc)
  {
     printf("BREAKPOINT\n");
     exit(1);
  }
  foo();
}
```

The program could be looking for a 0xCC in the whole assembly not only at a certain address

Manually insert an ICEBP (0xF1) – instead of 0xCC - which also stops gdb

PTRACE technique

DETECTING DEBUGGING

- □ Only one process at time can ptrace a program
- □ If the tracee invokes ptrace it will get an error i.e. return value = -1
- □ The tracee can know if it is being debugged by trying itself to invoke ptrace

```
// antidebug.c
int main()
{
  if (ptrace(PTRACE_TRACEME, 0, 1, 0) < 0)
  {
    printf(“Don’t waste your time!\n");
    return 1;
  }
  printf("Hello\n");
  return 0;
}
```

DETECTING DEBUGGING

- □ Only one process at time can ptrace a program
- □ If the tracee invokes ptrace it will get an error i.e. return value = -1
- □ The trac
  being de
  itself to

```
// antidebug.c
int main()
{
  if (ptrace(PTRACE_TRACEME, 0, 1, 0) < 0)
  {
    printf("Don't waste your time!\n");
```

To patch this and still perform the analysis:
1)NOP or invert the ptrace()
check before analyzing;
2)Before debugging overwrite
the ptrace function …

HIDING THE CALL TO PTRACE()

- □ We can hide the call to ptrace() by wripping it in a detection function such as:

```
void detect_gdb(void) __attribute__((constructor));
```

- □ `__attribute__((constructor[(priority)]))`
- □ ELF has two sections `.ctors` and `.dtors` that are used to store constructors and destructors
- □ `.ctors` functions are executed before `main()`
- □ Thus we can perform this test even before `main()`thus someway hiding the call and making it harder to intercept it
- □ Also check `.init` and `.fini`

OVERWRITING THE PTRACE FUNCTION

Create and load a shared library

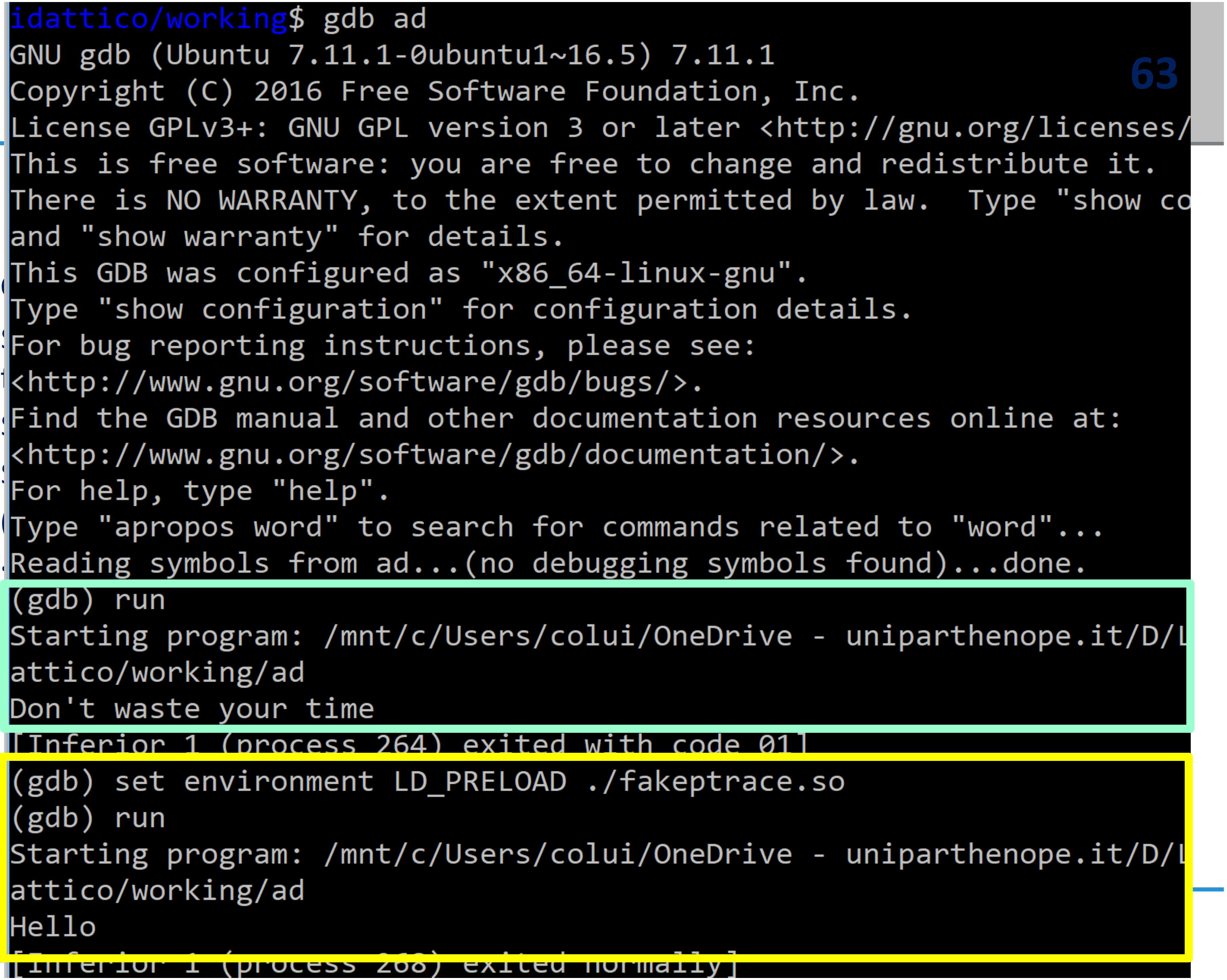
```
$ gcc -shared -o fakeptrace.so fakeptrace.c
$gcc –o ad antidebug.c
$gdb ad
(gdb) set environment LD_PRELOAD ./fakeptrace.so
(gdb) run
Hello
```

fakeptrace.c

```c
long ptrace(int request, int
  pid, int addr, int data)
{
    return 0;
}
```

In radare2 you can execute:

```
r2 –Ad rarun2 program=./ad preload=./fakeptrace.so
```



```
idattico/working$ gdb ad
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show co
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:
<http://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 ad...(no debugging symbols found)...done.
(gdb) run
Starting program: /mnt/c/Users/colui/OneDrive - uniparthenope.it/D/L
attico/working/ad
Don't waste your time
[Inferior 1 (process 264) exited with code 01]
(gdb) set environment LD_PRELOAD ./fakeptrace.so
(gdb) run
Starting program: /mnt/c/Users/colui/OneDrive - uniparthenope.it/D/L
attico/working/ad
Hello
[Inferior 1 (process 268) exited normally]
```

…BUT FAKEPTRACE ALWAYS RETURNS 0

```
int main()
{
    int offset = 0;
    if (ptrace(PTRACE_TRACEME, 0, 1, 0) == 0)  offset = 2;
    if (ptrace(PTRACE_TRACEME, 0, 1, 0) == -1) offset = offset * 3
    if (offset == 2 * 3){
        // normal execution
    } else {
        // don't trace me;
    }
}
```

…BUT FAKEPTRACE ALWAYS RETURNS 0

```
int main()
{
    int offset = 0;
    if (ptrace(PTRACE_TRACEME, 0, 1, 0) == 0)  offset = 2;
    if (ptrace(P                                          ffset * 3
    if (offset ==
        // norma
    } else {
        // don't trace me;
    }
}
```

The fakeptrace can have a state and reply 0 at first time and -1 for following calls

Additional techniques

WINDOWS ANTI-DEBUGGING: THE EASY WAY

```
int main()
{
    if (IsDebuggerPresent())
    {
        std::cout << "Stop debugging program!" << std::endl;
        exit(-1);
    }
    return 0;
}
```

IsDebuggerPresent()

- Determines whether the calling process is being debugged by a user-mode debugger

x32 implementation	**x64 implementation**
mov eax,dword ptr fs:[00000030h] movzx eax,byte ptr [eax+2] ret	mov rax,qword ptr gs:[60h] movzx eax,byte ptr [rax+2] ret

- It checks the second byte of the PEB (Process Environment Block) structure (fs:30h in x32, gs:60h in x64)

IsDebuggerPresent()

Windows PEB structure:

```
typedef struct _PEB {
    BYTE Reserved1[2];
    BYTE BeingDebugged;
    BYTE Reserved2[21];
    PPEB_LDR_DATA LoaderData;
    PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
    BYTE Reserved3[520];
    PPS_POST_PROCESS_INIT_ROUTINE
                                            PostProcessInitRoutine;
    BYTE Reserved4[136];
    ULONG SessionId;
} PEB;
```

1= being debugged
0 = not debugged

BYPASSING ISDEBUGGERPRESENT CHECK

- □ NOP the call to isDebuggerPresent()
  - To make it difficult the antidebugger programmer will not invoke the function in the main program (easy to discover and NOP) but in a TLS Callback (that are called when a thread starts or exits – cleanly - in the current process)
- □ Modify the PEB.BeingDebugged value
  - E.g. x32

```
mov eax, dword ptr fs:[0x30]
mov byte ptr ds:[eax+2], 0
```

- □ Update the value of EAX to 0 after the call
- □ …

CheckRemoteDebuggerPresent()

```
BOOL WINAPI CheckRemoteDebuggerPresent(
  _In_    HANDLE hProcess,
  _Inout_ PBOOL  pbDebuggerPresent // after the call TRUE if debugged
);
```

- □ Patch the comparison of the return value of CheckRemoteDebuggerPresent() in the malware code
- □ Patch the malware to jump over the CheckRemoteDebuggerPresent() check
- □ Patch the malware to NOP the CheckRemoteDebuggerPresent() check
- □ Set a breakpoint after the NtQueryInformationProcess() call and update its return value for ProcessDebugPort to 0
- □ Pre-load/hook a DLL that overrides NtQueryInformationProcess() and always returns 0 for ProcessDebugPort

CheckRemoteDebuggerPresent()

```
int main(int argc, char *argv[])
{
    BOOL isDebuggerPresent = FALSE;
    if (CheckRemoteDebuggerPresent(GetCurrentProcess(), &isDebuggerPresent ))
    {
        if (isDebuggerPresent )
        {
            std::cout << "Stop debugging program!" << std::endl;
            exit(-1);
        }
    }
    return 0;
}
```

Hardware Breakpoints

- □ Software Breakpoints are easy to detect and slow
- □ Hardware breakpoints
  - 8 dedicated registers: DR0-7
  - DR0-DR3 – breakpoint registers
    - contain linear addresses of breakpoints
  - DR4 & DR5 – reserved
  - DR6 – debug status
    - Indicates, which breakpoint is activated
  - DR7 – debug control
    - defines the breakpoint activation mode by the access mode: read, write, or execute

```
CONTEXT ctx = {};
ctx.ContextFlags = CONTEXT_DEBUG_REGISTERS;
if (GetThreadContext(GetCurrentThread(), &ctx))
{
   if (ctx.Dr0 != 0 || ctx.Dr1 != 0 || ctx.Dr2 != 0
     || ctx.Dr3 != 0)
   {
      cout << "Stop debugging program!" <<endl;
      exit(-1);
   }
}
```