# Applied Software Security

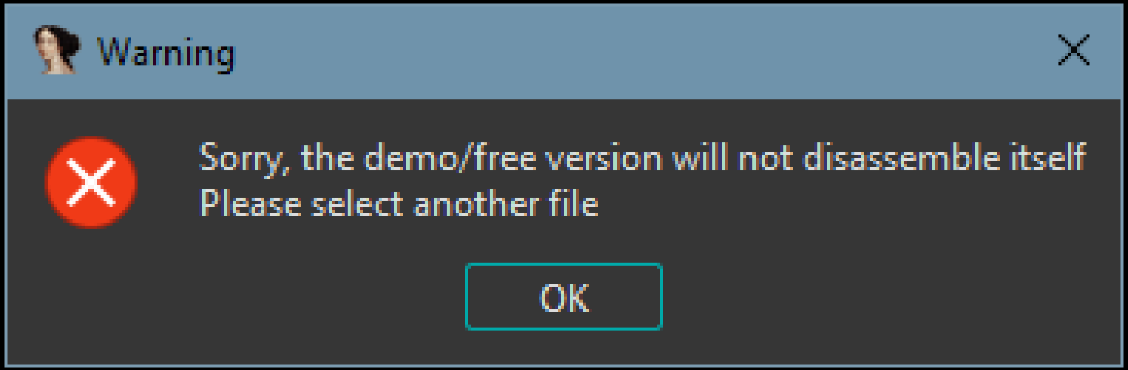
## Week 4 Tutorial
## Anti-Reversing and Anti-debugging

# Anti-Reversing: What?

Anything that makes the process of analyzing the compiled code more difficult

- Specifically when done on purpose

Anti-debugging specifically refers to making debugging less pleasant



# Anti-Reversing: Why?

To make your life harder.

In general, it benefits a software publisher to make their code harder to reverse-engineer. (Unless they are a pure free software developer)

- Mainly for keeping trade secrets / protecting intellectual property
  - Make copy-protection hard to crack (for both software and media)
  - Obfuscate APIs and protocols
- “Security by obscurity”
  - Generally not a good practice by itself
  - A hacker will find a way, you’ll just make them angrier in the process
- Malicious code / Malware
  - Also a type of “intellectual property” protection
  - Malicious code is bound to be analyzed at some point
  - The enemy of the malware developer is the reverse-engineer

# Anti-Reversing: Why?

- These techniques are costly
  - More programmer hours
  - Costly obfuscation software
  - Costly also in the sense of excess program run time and size
- Some code is not worth the effort
- If a processor can run it then (usually) a human will eventually be able to read it

# Anti-Reversing: How?

General approaches:

- Symbol elimination
- Obfuscation
- Encryption
- Confusion / unnecessary complication

Tool-specific approaches:

- Specifically targeting certain disassemblers and decompilers
- Tools exist specifically for VM based programs (Java/.NET) that are inherently easier to decompile
- Anti-debuggers
- May require reverse-engineering the tools themselves

# Anti-Reversing: How?

First and most obvious – removing symbolic data:

- Strip the debugging and symbol info from the executable
- Obfuscate strings that you have to use in the program
  - e.g. by xor with some number, or something more complicated
  - so they won't be detected as ascii strings and used as anchor points
- Obfuscate literals/"magic numbers"
  - e.g. instead of "mov eax, `089ABCDEFh`" use:

    ```
    mov eax, 080A0C0E0h
    or  eax, 0090B0D0Fh
    ```
  - Some decompilers will still detect the original value

# Anti-Reversing: How?

Code Obfuscation/Encryption:

- Pack / shell / obfuscate the code and have a small code snippet unpack it before the main code runs
- Encrypt the code
  - Remember that the code will have to run unencrypted from memory eventually, so this is not a perfect solution
  - You’ll also have to store the key somewhere...

# Anti-Reversing: How?

Confusion:

- Doing calculations in a roundabout way
- Sacrificing efficiency for un-clarity
- Strange program flow
- Long / inline functions
- Constructing own program logic
- Not using standard libraries for basic tasks

# Anti-Reversing: How?

## Confusion, an academic example:

Here is such an algorithm, that sorts an array $A$ of $n$ elements in non-decreasing order. For easier exposition in the proof later on, the array is 1-based, so the elements are $A[1], \ldots, A[n]$.

---

**Algorithm 1** ICan'tBelieveItCanSort($A[1..n]$)

---

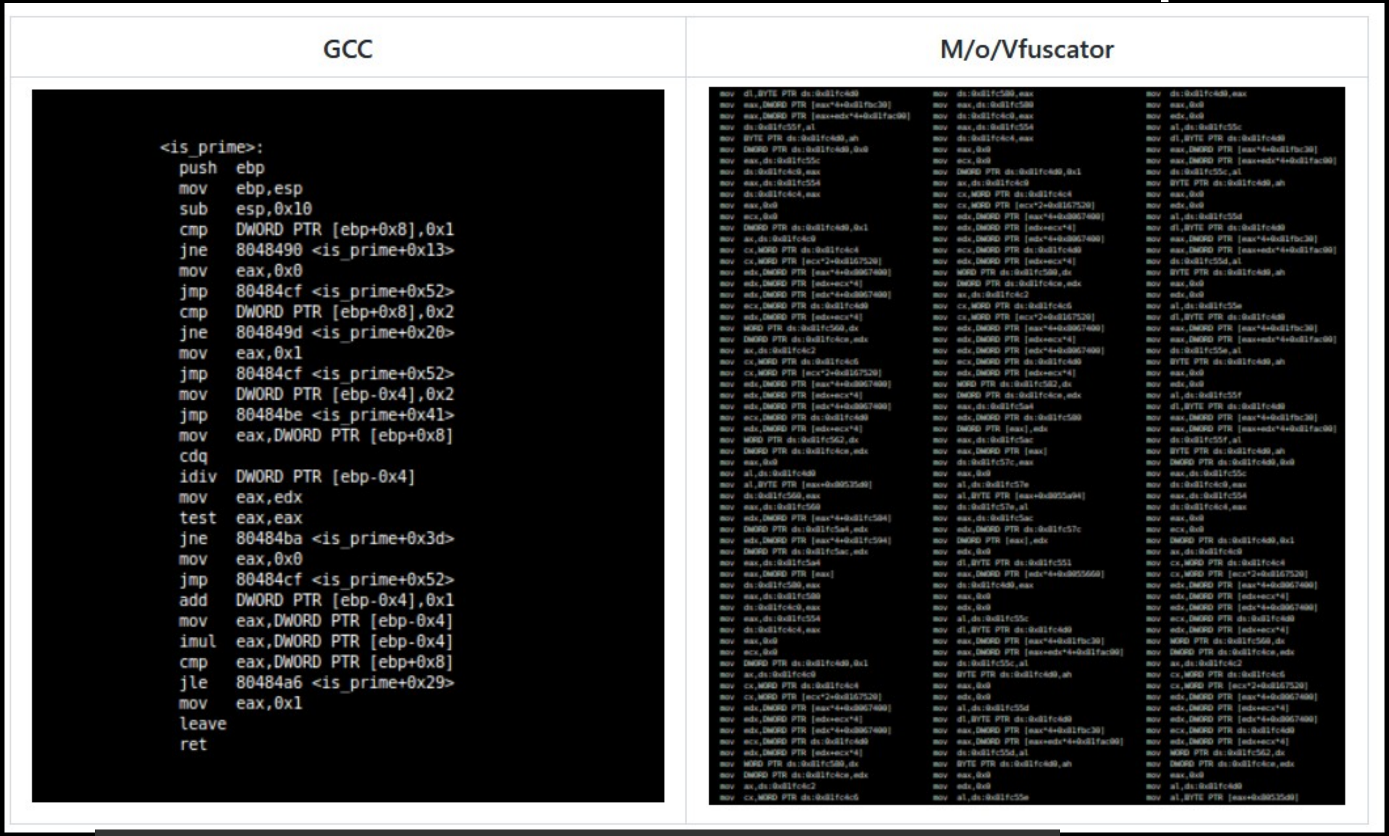
**for** $i = 1$ **to** $n$ **do**
    **for** $j = 1$ **to** $n$ **do**
        **if** $A[i] < A[j]$ **then**
            swap $A[i]$ and $A[j]$

---

https://arxiv.org/pdf/2110.01111.pdf

# Anti-Reversing: How?

## Confusion, a ludicrous example:




https://github.com/xoreaxeaxeax/movfuscator

- Movfuscator, a compiler for 32-bit x86 using only mov instructions.
- Apparently, mov is Turing-complete
  mov-is-turing-complete.pdf

# Tool-specific approaches

## Confusing the disassembler

Jumping to the middle of an instruction (from unreachable code)

Treating code as data and data as code

Inserting function prologues and epilogues where they shouldn’t be and calling non-functions

Heavily indirect calls and jumps

Breaking traditional function structures

# Tool-specific approaches

Decompilers in particular are an easy target. Take the following toy example:

```c
#include <stdio.h>

const int confusion = 0;

int main(int argc, char *argv[])
{
  if (*((volatile int *) &confusion) == 0)
  {
    printf("Everything is fine\n");
  }
  else
  {
    printf("I am confused\n");
  }
}
```

- Since “confusion” is declared a constant, it will be put in the .rodata section
- Casting to “volatile” tells the compiler not to actually consider “confusion” as equivalent to 0
- However, since a decompiler will treat it as read-only, its decompilation may assume it’s equal to 0

- If we look at Ghidra’s disassembly and decompilation side by side, we can see that the “unreachable” code is omitted

```
                     FUN_00101149                              XREF[3]:     entry:00101081(*), 0010
                                                                            00102100(*)
00101149 f3 0f 1e fa      ENDBR64
0010114d 55               PUSH         RBP
0010114e 48 89 e5         MOV          RBP,RSP
00101151 48 83 ec 10      SUB          RSP,0x10
00101155 89 7d fc         MOV          dword ptr [RBP + local_c],EDI
00101158 48 89 75 f0      MOV          qword ptr [RBP + local_18],RSI
0010115c 48 8d 05         LEA          RAX,[DAT_00102004]
         a1 0e 00 00
00101163 8b 00            MOV          EAX=>DAT_00102004,dword ptr [RAX]
00101165 85 c0            TEST         EAX,EAX
00101167 75 0e            JNZ          LAB_00101177
00101169 48 8d 3d         LEA          RDI,[s_Everything_is_fine_00102008]          = "Everything is fin
         98 0e 00 00
00101170 e8 db fe         CALL         <EXTERNAL>::puts                             int puts(char * __s)
         ff ff
00101175 eb 0c            JMP          LAB_00101183

                     LAB_00101177
00101177 48 8d 3d         LEA          RDI,[s_I_am_confused_0010201b]
         9d 0e 00 00
0010117e e8 cd fe         CALL         <EXTERNAL>::puts
         ff ff

                     LAB_00101183
00101183 b8 00 00         MOV          EAX,0x0
         00 00
00101188 c9               LEAVE
00101189 c3               RET
```

Decompile: FUN_00101149 - (confusion_)

```

/* WARNING: Removing unreachable block (ram,0x00101177) */

undefined8 FUN_00101149(void)

{
  puts("Everything is fine");
  return 0;
}

```

# Tool-specific approaches

If we cleverly craft code to change “confusion” we can outsmart the decompiler:

```c
#include <sys/mman.h>
#include <stdio.h>
#define PAGESIZE (4096)
```

```c
void some_func(void)
{
  // Round down to the nearest page
  void *rodata_page = (void*) (((long)(&confusion)) & (~(PAGESIZE - 1)));
  // Make .rodata section writable; this library function calls the mprotect system call
  mprotect(rodata_page, PAGESIZE, PROT_WRITE | PROT_READ);
  // Write to a read-only variable
  *((int *) &confusion) = 1;
}
```

- This code first uses “mprotect” to grant write permissions to the read-only .rodata section
- It then changes “confusion” to 1

# Tool-specific approaches

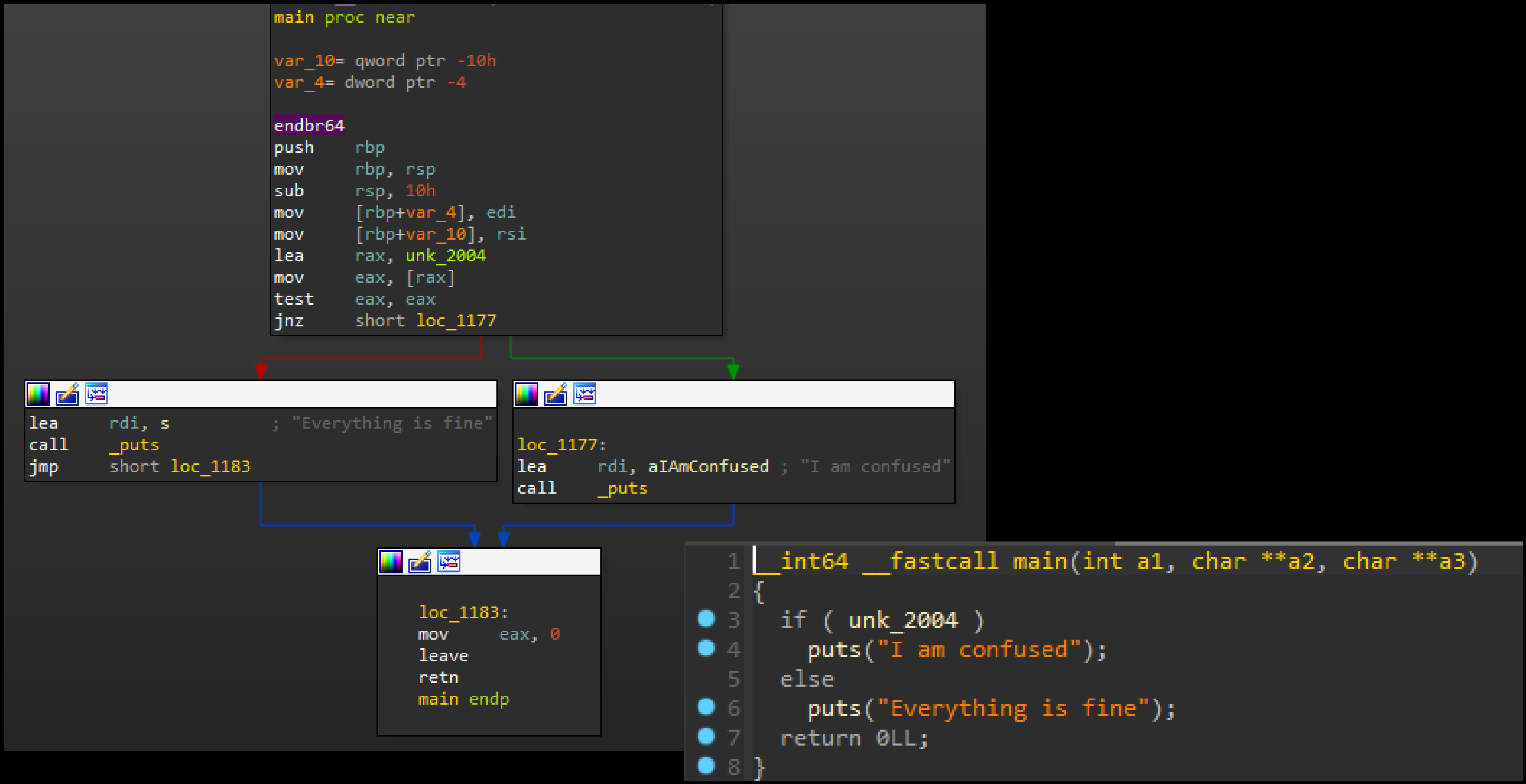
However, if Ghidra sees that the program writes to “confuson”, it’ll no longer consider it a constant.

Decompile: main - (confusion21)

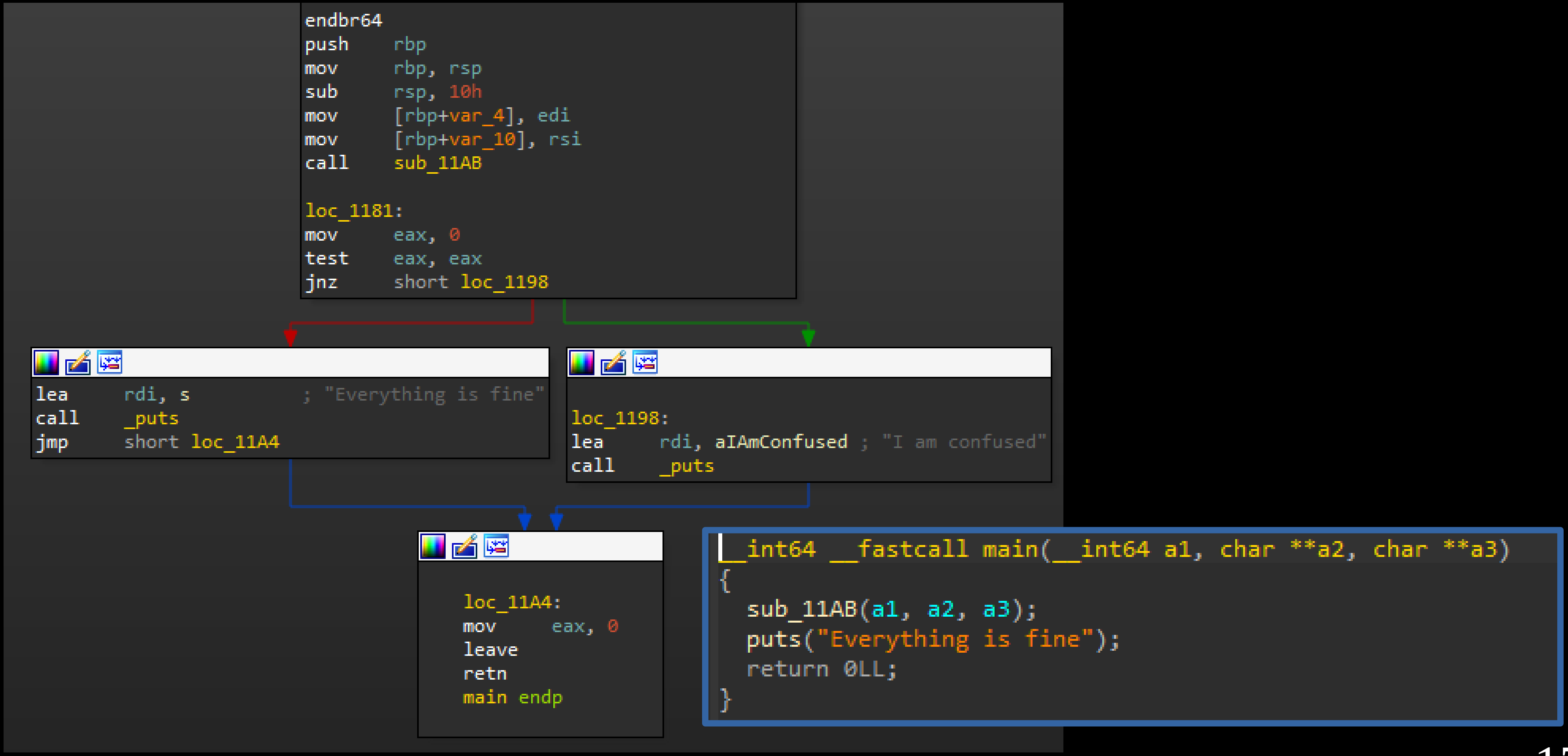
```
int main(int argc,char **argv)

{
  some_func();
  if (confusion == 0) {
    puts("Everything is fine");
  }
  else {
    puts("I am confused");
  }
  return 0;
}
```

- We can overcome this by invoking “`some_function`” in a roundabout way (for example, very indirectly), such that Ghidra will not recognize it as a function in its analysis

IDA, however, displays both paths even when “confusion” is truly read-only:



# But we can make IDA drop some code too, by making it truly unreachable:



We can still change the flow to get to the unreachable code by modifying the code itself from somewhere else in the program

- For example changing “`mov eax, 0`” to “`mov eax, 1`”

To do that we need to make the “.text” section writable the same way as before:

```c
void some_func(void)
{

  // Round down to the nearest page
  void *text_page = (void*) (((long)(&main)) & (~(PAGESIZE - 1)));
  // Make .text section writable; this library function calls the mprotect system call
  mprotect(text_page, PAGESIZE, PROT_WRITE | PROT_READ | PROT_EXEC);
  // Change zero byte to 1 in main such that "mov eax, 0" changes to "mov eax, 1"
  *((char *) main + 0x19) = 0x01;

}

```

```
$ ./confusion
I am confused
```

# (Anti-)Debugging

First we'll discuss (briefly) how debuggers actually work.

- In (usermode) Unix-like systems:

```
NAME
       ptrace - process trace

SYNOPSIS
       #include <sys/ptrace.h>

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

DESCRIPTION
       The ptrace() system call provides a means 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.

(from the manual page for ptrace)
```

# (Anti-)Debugging

```
A  process  can initiate a trace by calling fork(2) and having
the resulting child do a PTRACE_TRACEME, followed  (typically)
by  an  execve(2).   Alternatively,  one  process may commence
tracing another process using PTRACE_ATTACH or PTRACE_SEIZE.

While being traced, the tracee will stop each time a signal is
delivered, even if the signal is being ignored.  (An exception
is SIGKILL, which has its usual effect.)  The tracer  will  be
notified at its next call to waitpid(2) (or one of the related
"wait" system calls); that call will  return  a  status  value
containing information that indicates the cause of the stop in
the tracee.  While the tracee is stopped, the tracer  can  use
various ptrace requests to inspect and modify the tracee.  The
tracer then causes the tracee to continue, optionally ignoring
the  delivered  signal  (or even delivering a different signal
Instead).
```

(from the manual page for ptrace)

# (Anti-)Debugging

So how do breakpoints work? (Very briefly)

- Interrupts are programmed into the code. Of particular interest to us are Interrups 1 (Debug) and 3 (Breakpoint)
- The tracer process can react to the signals these interrupts generate

**Table 6-1. Exceptions and Interrupts**

| Vector | Mnemonic | Description | Source |
|---|---|---|---|
| 0 | #DE | Divide Error | DIV and IDIV instructions. |
| 1 | #DB | Debug | Any code or data reference. |
| 2 | | NMI Interrupt | Non-maskable external interrupt. |
| 3 | #BP | Breakpoint | INT 3 instruction. |
| 4 | #OF | Overflow | INTO instruction. |
| 5 | #BR | BOUND Range Exceeded | BOUND instruction. |
| 6 | #UD | Invalid Opcode (UnDefined Opcode) | UD2 instruction or reserved opcode.[1] |
| 7 | #NM | Device Not Available (No Math Coprocessor) | Floating-point or WAIT/FWAIT instruction. |

# (Anti-)Debugging

The tracer then waits for signals from the tracee using wait() and waitpid() syscalls

```
Real parent
       The ptrace API (ab)uses the standard UNIX parent/child signaling
       over waitpid(2).  This used to cause the real parent of the
       process to stop receiving several kinds of waitpid(2)
       notifications when the child process is traced by some other
       process.

       Many of these bugs have been fixed, but as of Linux 2.6.38
       several still exist; see BUGS below.

       As of Linux 2.6.38, the following is believed to work correctly:

       *  exit/death by signal is reported first to the tracer, then,
          when the tracer consumes the waitpid(2) result, to the real
          parent (to the real parent only when the whole multithreaded
          process exits).  If the tracer and the real parent are the
          same process, the report is sent only once.
```

(from the **official** manual page for ptrace)

# (Anti-)Debugging

When the TRAP flag is set in the EFLAGS register, Interrupt 1 is triggered on every instruction (step)

- There is no special instruction to set or clear the TRAP flag. One must adjust the flag register manually (thank you Intel):

```
pushf                        ; place EFLAGS register on the stack
or WORD PTR[rsp], 0x0100     ; Set TRAP flag to 1
popf                         ; write back the altered flags to the CPU
```

The debugging process (tracer) will receive a SIGTRAP signal on every instruction of the debugee, which the debugger will then handle

# (Anti-)Debugging

Software breakpoints are set by overwriting the code with the INT3 instruction (0xCC)

- One-byte instruction so that one-byte instructions can be overwritten without overwriting the next one
- For Longer instructions only the first byte is overwritten
- Other interrupts use a two-byte instruction (CD ??)

GDB and other debuggers will mask the interrupts they set in the code to the human using it. (If you examine the memory at a breakpoint in GDB, it will claim the instruction hasn’t changed.)

- Assuming the code remains static, it’s a nice solution to implement unlimited breakpoints that doesn’t require hardware support (aside from interrupt handling)

# (Anti-)Debugging

What about more hardware support?

- x86_64 provides hardware support for (code and data-access) breakpoints in the form of *debug registers* DR0-DR4 that hold breakpoint/watchpoint addresses.
- DR6 is the debug status register (for determining which debug conditions have occured)
- DR7 is the debug control register (for setting breakpoints/watchpoints)
- Privileged resource, can only be accessed by the kernel
- Support only four breakpoints per core, one per register including for kernel uses. For user programs they can be switched with each context switch and shared between different threads and processes
- GDB tries to register HW breakpoints via the ptrace syscall

  `hbreak` – set hardware breakpoint

  `awatch` – set a watchpoint (hardware access (r/w) watchpoint)

```
Could not insert hardware breakpoints:
You may have requested too many hardware breakpoints/watchpoints.

Command aborted.
```

# **Anti**-Debugging

Anti-debugging:

- Detection of debugger
- Fail/Exit if debugger present
  - Or different behavior when being debugged
- Making it difficult to set breakpoints and watchpoints
- Obfuscation of run-time memory (as well as program memory)

# Anti-Debugging

Some basic anti debugging methods:

- Detecting if a specific debugger is installed/running by looking at the running processes or the filesystem
- Timing the program execution to detect abnormally long times (to detect halted execution due to a breakpoint, or single stepping)
- Code integrity checks to detect software interrupts (injected int3 instructions) and patches
- Packed code will also not work if breakpoints are set before unpacking

# Anti-Debugging

More basic anti debugging methods:

- Checking whether the program is being debugged by using the OS API:
  - In Linux: Calling ptrace with a PTRACE_TRACEME request that should fail if the process is already being traced
  - Also in Linux: Looking for TracerPID in /proc/(PID)/status
  - In Windows: IsDebuggerPresent()
- Attaching yourself as the tracer
  - Cannot attach another debugger to a process that is being debugged
  - Moreover, one can implement some important functionality via ptrace between the tracer and the tracee
  - That way the researcher can’t forcibly attach a debugger without breaking the functionality
- Detecting SIGTRAP and SIGINT signals

# Anti-Debugging

How to deal with anti-debugging techniques:

- Disassemble statically (you be the CPU)
- If you can run it, you’re probably able to patch it
  - NOP it out: Statically Identify the anti-debugging mechanisms and disable them
- To deal with code integrity checks and packing:
  - Use hardware breakpoints as opposed to software (`hbreak` in gdb)
  - Set breakpoints in the standard library functions instead of the main executable
- Creativity
  - Logs, prints, hooks, etc.

The next exercises should be difficult but fun and fulfilling.