iSECpartners
part of nccgroup

Shattering Illusions in Lock-Free Worlds

Compiler and Hardware behaviors in OSes and VMs

Marc Blanchou

black hat USA 2013

Introduction

Developer:

"Compiler/hardware, that's not the code I wrote for my driver?!"

Compiler/hardware:

"But your code is **correctly synchronized** right? So you should not care – you do not want to actually execute this horror you just wrote – **trust us**"

Introduction

iSECpartners part of nccgroup

- What is it about?
  - Race conditions introduced by the compiler or the hardware in lock-free sections (in OSes and VMs among others)
- Why should you care?
  - You don't realize how messy lock-free code can be
  - You want to find these bugs more easily
  - You want to know more about the different layers involved in these types of race conditions

Agenda

iSECpartners part of nccgroup

- Definitions
  - Lock-free programming
  - Memory models
  - Optimizations
- Compiler, hardware and races
  - Reordering issues
  - Double-fetch (TOCTTOU) issues
  - Other issues
- How to find these bugs?
- Solutions?

Locks?

iSECpartners part of nccgroup

- Locks were initially created because of the difficulty of writing correct multi-threaded code
- They more or less allow developers (and researchers) to not care too much about memory models and various compiler and hardware optimizations

A multiprocessing system on a single computer involves problems similar to those of a distributed system because of the unpredictable order in which certain events can occur. … We have found that problems often arise because people are not fully aware of this fact and its implications.

— Leslie Lamport **1978**

Lock-free programming

iSECpartners part of nccgroup

- What is it?
  - Threads never waiting on each others
    - No more deadlocks
    - No livelocks or theorical scheduling issues
  - Usually cheaper and scale better than locks
  - Always completes operations in time (critical)
  - Usually only used for a few sections of applications
  - **Way harder** to get right than using locks
- What for?
  - Various OSes and VM operations
  - Multimedia and financial apps, some databases etc.

The (very) obvious

iSECpartners part of nccgroup

- What a compiler/hardware knows
  - Memory operations within the thread
- What a compiler/hardware does not know
  - Shared memory locations
- Solution
  - Let the compiler/hardware know
    - Appropriate memory barriers and atomic operations
    - BUT you can't make assumptions about memory models anymore
    - Be careful with compiler/hardware specific code
      - Can break on newer or different hardware/compilers

Cache and Sequential Consistency

iSECpartners part of nccgroup

- Cache coherence
  - No data lost or written before being transferred from the cache to the target memory
- Sequential Consistency (or illusion of)
  - Order of memory operations specified by a program
  - No memory reordering should be visible
  - People usually write code that needs SC
  - **This is language and hardware dependent**

Compiler Optimizations

iSECpartners part of nccgroup

- If not told otherwise, the compiler can do any optimization it wants to, as long the compiled code acts **as if** it would run on a **single threaded** machine.

- What about Profile Guided Optimization (PGO) or code obfuscation?

Memory models

iSECpartners part of nccgroup

Source code

```
int gl_value, gl_random_number;
int gl_is_updated;

void thread1() {
    gl_value = gl_random_number + 42

    gl_is_updated = 1;
}


int thread2()
{
  if (gl_is_updated) {

    return gl_value;
  }
}
```

Compiler

```
Peephole optimizations
    Constant folding
    Strength reduction
    Null sequences
    Combine Operations
    Algebraic Laws
    Special Case Instructions
    Address Mode Operations
Local optimizations
Global optimizations
Loop optimizations
    fission/distribution
    fusion/combining
    interchange/permutation
    inversion
    loop-invariant code motion
    parallelization
    reversal
    scheduling
    skewing
    software pipelining
    splitting/peeling
    tiling/blocking
    vectorization
    unrolling
    unswitching
    sectioning
Interprocedural, whole-program
    or link-time optimization
Machine code optimization
```

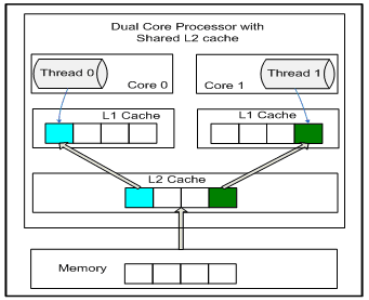
Machine code

```
thread1():
.LFB0:
    mov DWORD PTR gl_is_updated[rip], 1
    mov eax, DWORD PTR gl_random_number[rip]
    add eax, 42
    mov DWORD PTR gl_value[rip], eax
    ret
.LFE0:
thread2():
.LFB1:
    mov eax, DWORD PTR gl_is_updated[rip]
    test    eax, eax
    je  .L4
    mov eax, DWORD PTR gl_value[rip]
    ret
```

Processor / cache

Reordering
Prefetch
Branch prediction
private and share cache

Execution



Software memory model

Hardware memory model

Weak			Strong (closer to SC)
ARM	IA-64 (Itanium)	PowerPC	X86 - 64

What can go wrong?

Developer:

"Compiler/harware, but I swear to synchronize properly!"

Compiler/hardware:

"Fine… We'll do our best so that you can't tell we modified the program you wrote."

iSECpartners part of nccgroup

I'll synchronize, I promise

- Definitions
- **Compilers, hardware and races**
  - Reordering issues
  - Double-fetch (TOCTTOU) issues
  - Other issues
- How to find these bugs?
- Solutions

Lock-free and reordering

iSECpartners part of nccgroup

- Reordering can happen at compile time as well as at runtime (hardware).
- We did not need to care with locks before

- What does it mean for the developer?
  - Atomic operations
  - Appropriate memory barriers

Atomicity

iSECpartners part of nccgroup

- C/C++ operations are NOT presumed atomic
  - *But some native types can be if they are aligned*
- C++11: atomic<>
- RMW (read-modify-write) operations
- CAS (compare-and-swap)

Non-atomic	Atomic
g_value++; // g_value = g_value + 1;	InterlockedIncrement(&g_value);
int* rValue = (int*)([aligned_ptr] + 3); *rValue = 42; // not aligned	

Compiler reordering

iSECpartners part of nccgroup

- G++ 4.8 with no optimization flags

```
7  int gl_value, gl_random_number;
8  int gl_is_updated;
9
10 void thread1()
11 {
12     gl_value = gl_random_number + 42;
13
14
15     gl_is_updated = 1;
16 }
17
```

```
7      .zero   4
8  thread1():
9  .LFB0:
10     push    rbp
11     mov rbp, rsp
12     mov eax, DWORD PTR gl_random_number[rip]
13     add eax, 42
14     mov DWORD PTR gl_value[rip], eax
15     mov DWORD PTR gl_is_updated[rip], 1
16     pop rbp
17     ret
```

- G++ 4.8 with -O3

```
1  int gl_value, gl_random_number;
2  int gl_is_updated;
3
4  void thread1()
5  {
6      gl_value = gl_random_number + 42;
7      gl_is_updated = 1;
8  }
```

```
1  .Ltext0:
2  thread1():
3  .LFB0:
4      mov eax, DWORD PTR gl_random_number[rip]
5      mov DWORD PTR gl_is_updated[rip], 1
6      add eax, 42
7      mov DWORD PTR gl_value[rip], eax
8      ret
```

Compiler reordering

iSECpartners part of nccgroup

- G++ 4.8 and –O3

```
1  int gl_value, gl_random_number;
2  int gl_is_updated;
3
4  void thread1()
5  {
6      gl_value = gl_random_number + 42;
7      gl_is_updated = 1;
8  }
9
10 int thread2()
11 {
12   if (gl_is_updated)
13     return gl_value;
14 }
```

```
1  .Ltext0:
2  thread1():
3  .LFB0:
4      mov eax, DWORD PTR gl_random_number[rip]
5      mov DWORD PTR gl_is_updated[rip], 1
6      add eax, 42
7      mov DWORD PTR gl_value[rip], eax
8      ret
9  .LFE0:
10 thread2():
11 .LFB1:
12     mov eax, DWORD PTR gl_is_updated[rip]
13     test    eax, eax
14     je  .L4
15     mov eax, DWORD PTR gl_value[rip]
16     ret
17 .L4:
18     rep; ret
```

Compiler barriers

iSECpartners part of nccgroup

```
1  int gl_value, gl_random_number;
2  int gl_is_updated;
3
4  void thread1()
5  {
6      gl_value = gl_random_number + 42;
7      asm volatile("" ::: "memory");
8      gl_is_updated = 1;
9  }
```

```
1  .Ltext0:
2  thread1():
3  .LFB0:
4      mov eax, DWORD PTR gl_random_number[rip]
5      add eax, 42
6      mov DWORD PTR gl_value[rip], eax
7      mov DWORD PTR gl_is_updated[rip], 1
8      ret
9  .LFE0:
```

Prevent compiler reordering*

```
1  int gl_value, gl_random_number;
2  int gl_is_updated;
3
4  #define SIMPLE_BARRIER() asm volatile("" ::: "memory")
5
6  void thread1()
7  {
8      gl_value = gl_random_number + 42;
9      SIMPLE_BARRIER();
10     gl_is_updated = 1;
11 }
12
13 int thread2()
14 {
15   if (gl_is_updated) {
16     SIMPLE_BARRIER();
17     return gl_value;
18   }
19 }
```

```
1  .Ltext0:
2  thread1():
3  .LFB0:
4      mov eax, DWORD PTR gl_random_number[rip]
5      add eax, 42
6      mov DWORD PTR gl_value[rip], eax
7      mov DWORD PTR gl_is_updated[rip], 1
8      ret
9  .LFE0:
10 thread2():
11 .LFB1:
12     mov eax, DWORD PTR gl_is_updated[rip]
13     test    eax, eax
14     je  .L4
15     mov eax, DWORD PTR gl_value[rip]
16     ret
17 .L4:
18     rep; ret
19 .LFE1:
```

*note: this does not act as a hardware barrier

Preventing compile-time reordering

iSECpartners part of nccgroup

- Compiler barriers
  - VC++ specific
    - **Interlocked operations**
    - Volatile – not atomic, **VC++ specific** implementation (/volatile:ms)
    - ReadWriteBarrier() – but better use atomic<>
    - Use the /kernel flag!
  - GCC
    - *asm volatile ("" : : : "memory")*
    - Use specific **memory barriers defines** depending on the kernel
  - **C++11 atomic types**
    - Avoid relaxed atomic!
  - Volatile
    - Java: Full barrier (CPU+compiler) (!= C/C++ volatile)
    - Avoid volatile in C/C++ for synchronization
  - Implied
    - CPU fences
    - Some function calls (containing barriers or "unknown" functions) but they can be inlined
      - Use *__declspec(noinline)* for VC++ or *__attribute__((noinline))* for gcc

Real-time reordering

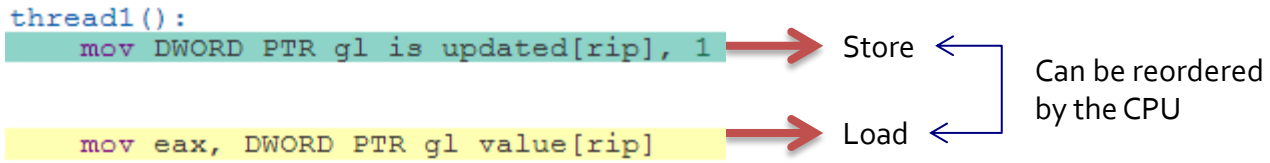
iSECpartners part of nccgroup

- Only visible on multicore or multiprocessor
- ONE CPU guarantees
  - Dependent memory accesses are in order
  - Overlapping load and store will appear ordered

Real-time reordering

iSECpartners part of nccgroup

- ONE CPU does NOT Guarantee
  - Overlapping memory accesses are not merged or discarded
  - Independent load and store are issued in the order given
    - Even on x86-64 (strong memory model)
      - An independent load (read) can be reordered with older stores



      - Non-SC load and store instruction: mov
      - SC load instruction: mov
      - SC store instruction: **xchg** (or mfence + mov)

Hardware barriers

iSECpartners part of nccgroup

- C++11 *atomic*<> types (apart from relaxed atomic)
- GCC: *volatile("[instruction]" ::: "memory")*
  - And the various defines (*mb()*, *rmb()*, *wmb()* etc.)
- VC++
  - *MemoryBarrier()* (full memory barrier, compiler+CPU)
  - Interlocked operations
- Volatile in Java (≠ C/C++ volatile)

Reordering at runtime

iSECpartners part of nccgroup

- Lots of different types of barriers depending on the CPU

C/C++11 Operation	x86 implementation
Load Relaxed:	MOV (from memory)
Load Consume:	MOV (from memory)
Load Acquire:	MOV (from memory)
Load Seq_Cst:	MOV (from memory)
Store Relaxed:	MOV (into memory)
Store Release:	MOV (into memory)
Store Seq Cst:	(LOCK) XCHG // alternative: MOV (into memory),MFENCE
Consume Fence:	<ignore>
Acquire Fence:	<ignore>
Release Fence:	<ignore>
Acq_Rel Fence:	<ignore>
Seq_Cst Fence:	MFENCE

C/C++11 Operation	ARM implementation
Load Relaxed:	ldr
Load Consume:	ldr + preserve dependencies until next `kill_dependency` *OR* ldr; teq; beq; isb *OR* ldr; dmb
Load Acquire:	ldr; teq; beq; isb *OR* ldr; dmb
Load Seq Cst:	ldr; dmb
Store Relaxed:	str
Store Release:	dmb; str
Store Seq Cst:	dmb; str; dmb
Cmpxchg Relaxed (32 bit):	_loop: ldrex roldval, [rptr]; mov rres, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq
Cmpxchg Acquire (32 bit):	_loop: ldrex roldval, [rptr]; mov rres, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq
Cmpxchg Release (32 bit):	dmb; _loop: ldrex roldval, [rptr]; mov rres, 0; teq roldval, rold; strexeq rres, rnewval, [rptr
Cmpxchg AcqRel (32 bit):	dmb; _loop: ldrex roldval, [rptr]; mov rres, 0; teq roldval, rold; strexeq rres, rnewval, [rptr
Cmpxchg SeqCst (32 bit):	dmb; _loop: ldrex roldval, [rptr]; mov rres, 0; teq roldval, rold; strexeq rres, rnewval, [rptr
Acquire Fence:	dmb
Release Fence:	dmb
AcqRel Fence:	dmb
SeqCst Fence:	dmb

http://www.cl.cam.ac.uk/~pes20/cpp/cppoxmappings.html

Other potential issues

iSECpartners part of nccgroup

- Speculative register promotion
- Write condition write
- Adjacent field overwrites
- Branch predictions
- Merging loops or inverting nested loops
- ABA problem?
- Conditional "locks"
- etc.

Example

iSECpartners part of nccgroup

Thread 1	Thread 2

g_value = ...;
gl_done = true;
	while (!gl_done) {
	[...]
	}
	local_data = g_value;

Example

iSECpartners part of nccgroup

- Register promotion and reordering

Thread 1	Thread 2

g_value = ...;
gl_done = true;
	while (!gl_done) {
	[...]
	}
	local_data = g_value;

Potential CPU and compiler reordering

Potential compiler optimization

```
register int tmp = gl_done;
while (!tmp) {
 [...]
}
local_data = g_value;
```

Classic double-fetch or TOCTTOU

iSECpartners part of nccgroup

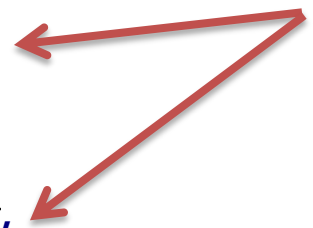
- Classic issue that can lead to privilege escalation
  - Kernel (local privilege escalation, userland->kernel)
  - Hypervisor (guest->host, VM breakout?)
- Example
  - Two memory reads in kernel space from a user-writable address
  - Kernel fetches the location once, verifies and validates the data
    - -> Attacker modifies the memory in user space
  - Kernel fetches the attacker-controlled value a second time and uses it

Classic double-fetch or TOCTTOU

iSECpartners part of nccgroup

```
void called_by_user(void* pUserSpaceMemory, [...]) { // kernel mode
        [...]
        try {
                [...]
                data_struct* p_data_struct =
                        (data_struct*) pUserSpaceMemory; // attacker controlled
                [...]

                ProbeForWrite(p_data_struct->buffer,
                              p_data_struct->len,
                              sizeof(UCHAR));
                [...]
                RtlCopyMemory(p_data_struct->buffer,
                              p_DATA,
                              p_data_struct->len);
                [...]
```



Captured twice

An attacker can change the address of the buffer after the check

More secure?

iSECpartners part of nccgroup

```
void called_by_user(void* pUserSpaceMemory, [...]) { // kernel mode
        [...]
        try {
                [...]
                captured_user_data = *(data_struct*) pUserSpaceMemory;
                [...]

                ProbeForWrite(captured_user_data.buffer,
                              captured_user_data.len,
                              sizeof(UCHAR));
                [...]
                RtlCopyMemory(captured_user_data.buffer,
                              p_DATA,
                              captured_user_data.len);
                [...]
```

Captured only once?

Potential compiler optimization?

iSECpartners part of nccgroup

```
void called_by_user(void* pUserSpaceMemory, [...]) { // kernel mode
        [...]
        try {
                [...]
                captured_user_data = *(data_struct*) pUserSpaceMemory;
                [...]

                ProbeForWrite(captured_user_data.buffer,
                              captured_user_data.len,
                              sizeof(UCHAR));
                ProbeForWrite(((data_struct*)pUserSpaceMemory)->buffer,
                              ((data_struct*)pUserSpaceMemory)->len,
                              sizeof(UCHAR));
                [...]
                RtlCopyMemory(captured_user_data.buffer,
                                                p_SOME_DATA,
                                                captured_user_data.len);
                RtlCopyMemory(((data_struct*)pUserSpaceMemory)->buffer),
                              p_DATA,
                              ((data_struct*)pUserSpaceMemory)->len));
```

Still captured twice?

The compiler may not see why you need the local storage (which just adds instructions) and could optimize away

Potential compiler bug?

iSECpartners part of nccgroup

```
void called_by_user(void* pUserSpaceMemory, [...]) { // kernel mode
        [...]
        try {
                [...]
                captured_user_data = *(volatile data_struct*) pUserSpaceMemory;
                [...]

                ProbeForWrite(captured_user_data.buffer,
                              captured_user_data.len,
                              sizeof(UCHAR));
                ProbeForWrite(((data_struct*)pUserSpaceMemory)->buffer,
                              ((data_struct*)pUserSpaceMemory)->len,
                              sizeof(UCHAR));
                [...]
                RtlCopyMemory(captured_user_data.buffer,
                                                p_SOME_DATA,
                                                captured_user_data.len);
                RtlCopyMemory(((data_struct*)pUserSpaceMemory)->buffer),
                              p_DATA,
                              ((data_struct*)pUserSpaceMemory)->len));
```

Force volatile semantic, force capture (legal) (/volatile:ms)

Use copy_from_user(...) on Linux

Still captured twice?

Compilers and CPUs can have issues

iSECpartners part of nccgroup

- Especially with lock-free code
- These bugs are more frequent than you may think, and can impact a lot of code – that may never be recompiled
- Compilers may not always follow the standard

How to find these bugs

iSECpartners part of nccgroup

- Blackbox
  - Determine which compiler was used
    - Any type of bug known to be introduced by the compiler?
  - Look for specific instructions (hardware barriers) and see what it is supposed to protect (in weak memory models: is the right instruction used?)
  - ThreadSanitizer (TSAN)
    - Linux/Mac based on Valgrind, based on PIN for Windows
  - Memory access pattern analysis?
    - See Bochspwn (M. Jurczyk and G. Coldwin)

Whitebox and solutions

iSECpartners part of nccgroup

- Thoroughly review code that:
  - Should not be optimized in any way
  - Where shared memory is accessed/written
- Test cases and fuzzing
  - You are not only testing your code but the compiler/CPU too
  - Using ThreadSanitizer (TSAN) or Helgrind
- Disabling optimizations has limits
- Compare an test against CPUs with weaker memory models
- Equivalence checking
  - Using different compilers (could be very difficult, though)
- *Temporary mitigation for the user: sandbox with one CPU*

Lock-free programming is hard

- It can create lots of issues that easily go unnoticed
- And even with valid code due to compiler/CPU issues

"The fences in the current [C++] standard may be the most experts-only construct we have in the language"
— Hans Boehm

"It's easy to write lock-free code that appears to work, but it's very difficult to write lock-free code that is correct and performs well. Even good magazines and refereed journals have published a substantial amount of lock-free code that was actually broken in subtle ways and needed correction."
— Herb Sutter

Thank You

iSECpartners part of nccgroup

- Marc Blanchou
  - Principal Security Consultant at iSEC Partners
  - marc@isecpartners.com

- References
  - Papers/articles from:
    - Hans Bohem, Leslie Lamport, Herb Sutter, Vance Morrison, Jeff Preshing, David Howells, Paul E McKenney, Intel Corporation, Andrei Alexandrescu, Linus Torvalds, Petru Marginean, Tian Tian

iSECpartners
part of nccgroup

UK Offices
Manchester - Head Office
Cheltenham
Edinburgh
Leatherhead
London
Thame

European Offices
Amsterdam - Netherlands
Munich – Germany
Zurich - Switzerland

North American Offices
San Francisco
Atlanta
New York
Seattle

Australian Offices
Sydney