# Concurrency and security

Dr Robert N. M. Watson
University of Cambridge
Computer Laboratory

Part II Security
12 February 2014

UNIVERSITY OF CAMBRIDGE

# Outline

- What is concurrency?
- How does it relate to security?
- Case studies
- (Some) lessons learned

concurrent (adj):

Running together in space, as parallel lines; going on side by side, as proceedings; occurring together, as events or circumstances; existing or arising together; conjoint, associated.

*Oxford English Dictionary, Second Edition*

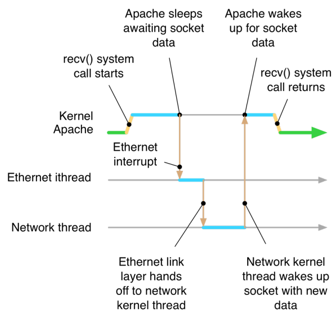


# Concurrency

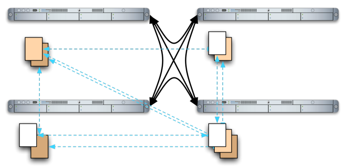
- Recall I.B *Concurrent and Distributed Systems*:
  - Multiple processes occur simultaneously and may interact with each other
  - Concurrency incurs the appearance (reality?) of non-determinism — e.g., variations in execution path and timing
- You were warned

# Origins of concurrency

- Interleaved or asynchronous computation
- Parallel computing
- Distributed systems



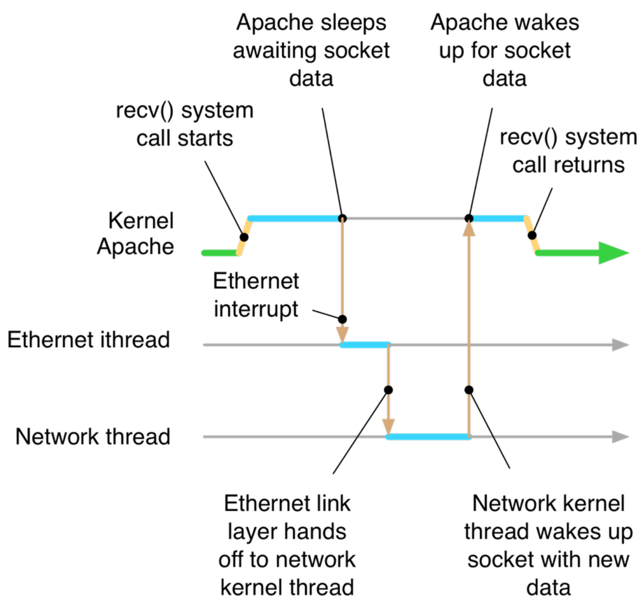






# Local concurrency

- Interleaved or asynchronous execution on a single processor
- “Better” scheduling, more efficient use of computation resources
- Mask I/O latency, multitasking, preemption



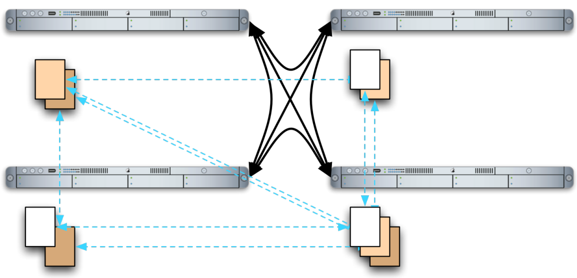
# Shared memory multiprocessing

- Multiple CPUs with shared memory
- Possibly asymmetric memory speed/topology
- Weaker memory model: writes order weakened, explicit synchronisation
- New programming models




# Message passing and distributed systems

- Protocol-centric approach with explicit communication
- Synchronous or asynchronous
- Explicit data consistency management
- Distributed file systems, databases, etc.



# Concurrency research

- Extract more concurrency and parallelism
- Maximise performance
- Represent concurrency to the programmer
- Identify necessary and sufficient orderings
- Detect and eliminate incorrectness
- Manage concurrency-originated failure modes



# Practical concerns with concurrency

- Performance
- Consistency of replicated data
- Liveliness of concurrency protocols
- Non-deterministic execution
- Distributed system failure modes

Most classes of bugs are interesting in security — but these two cconcurrency problems have proven particularly fruitful

# Concurrency and security

- Abbot, Bisbey/Hollingworth in 1970's
- Inadequate synchronisation or unexpected concurrency leads to violation of security policy
  - Most commonly: race conditions
  - Also a concern: timing side channels
- Distributed systems, multicore notebooks, ... this is an urgent and timely issue

E.g., OS access-control bugs

E.g., key leakage



# Consistency models and data races

- Semantics of accessing [possibly] replicated data concurrently from multiple processes
  - Strong models support traditional non-concurrent programming assumptions
  - Weak models exchange consistency for performance improvement
- Critical bugs can arise → *race conditions*

# ACID properties

- Database transaction properties
  - **A**tomicity - all or nothing
  - **C**onsistency - no inconsistent final states
  - **I**solation - no inconsistent intermediate states
  - **D**urability - results are durable



# Serialisability

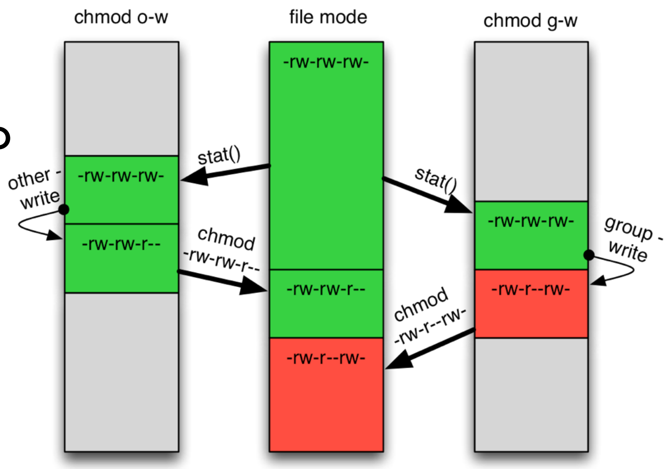
- Results of concurrent transactions must be equivalent to outcome of a possible serial execution of the transactions
  - Serialisable outcomes of {A, B, C}:
    - A B C    A C B    B A C<br>B C A    C A B    C B A
- Strong model that is easy to reason about

# Weaker consistency

- Strong models expose latency/contention/failure modes
- Desirable to allow access to stale data in distributed systems
- Timeouts: DNS caches, NFS attribute cache, x.509 certificates, Kerberos tickets
- Other weak semantics: AFS last close, UNIX passwd/group vs. in-kernel credentials
  - The difficulty of *revocation*
- More generally, *capability-system semantics*
  - E.g., UNIX file descriptors with respect to DAC



# UNIX API concurrency




- Simultaneously execute two instances of UNIX **chmod** with "update" syntax
  - chmod o-w file
  - chmod g-w file
- stat()/chmod() can't express atomicity
- Output of one system call lost: *read-modify-write* race
- *Passive vulnerability*: hard for attackers to exploit directly

# Concurrency vulnerabilities

- Incorrect concurrency management / synchronisation leads to vulnerability
  - Violation of specifications
  - Violation of user expectations
- Passive - information or privilege "leaked"
- Active - allow adversary to extract information, gain privilege, deny service...



# Reasoning about concurrency and security

- Both *security* and *concurrency* require reasoning about adversarial behaviour and bugs
  - "Weakest link" analysis
  - Malicious rather than probabilistic incidence
- Can't exercise bugs deterministically in testing Debuggers mask rather than reveal bugs
- Static and dynamic analysis tools limited

# From concurrency bug to security bug

- Concurrency bugs in security-critical interfaces
  - Races on arguments and interpretation
  - Atomic "check" and "access" not possible
- Data consistency vulnerabilities
  - Stale or inconsistent security metadata
  - Security metadata and data inconsistent
- Side channels from execution timing



# Learning by example

- Consider two vulnerability types briefly
  - /tmp race conditions
  - SMT covert channels
- Detailed study
  - System call wrapper races

# /tmp race conditions

- Bishop and Dilger, 1996
- UNIX file system APIs allow non-atomic sequences resulting in vulnerability
- Unprivileged processes manipulate shared /tmp
- Race against vulnerable privilege processes to replace targets of open(), etc.



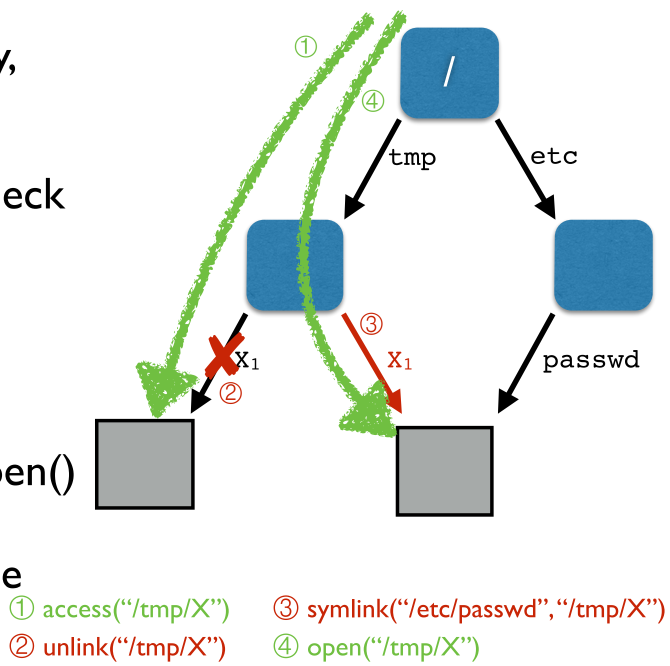
# xterm /tmp race

- xterm setuid root to allow pty, utmp operations

  1. access() used *real UID* to check permissions on /tmp/X

  2. open() uses *effective UID* to authorize file access

- Race between access() and open() lets attacker exploit xterm to overwrite system password file

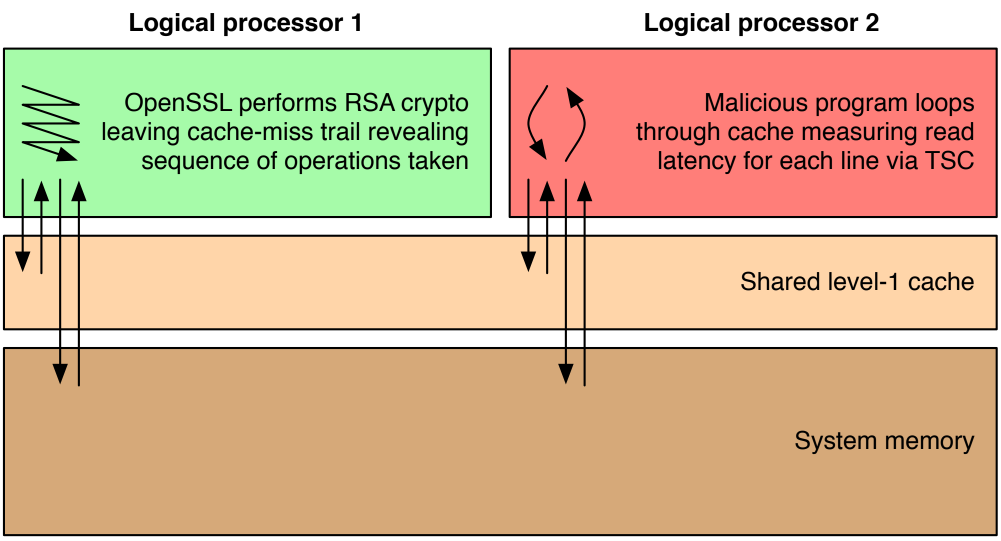


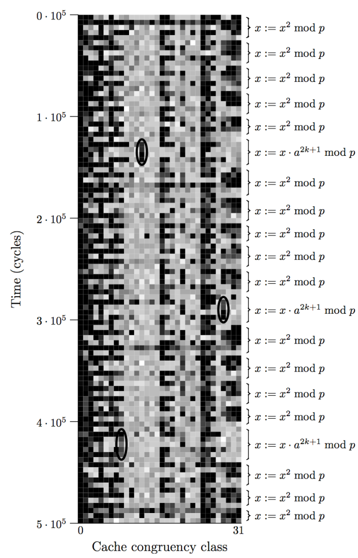
# SMT side channels

- Percival 2005, Bernstein 2005, Osvik 2005
- Covert/side channel channels historically considered an quite academic research topic
- Symmetric multithreading, hyper-threading, and multicore processors share caches
- Extract RSA, AES key material by analysing cache misses in "spy process"
- Many other side channels have been explored to extract keying material including, recently, *audio side channels* to extract RSA keys from other machines in the same room



# Percival SMT side-channel attack



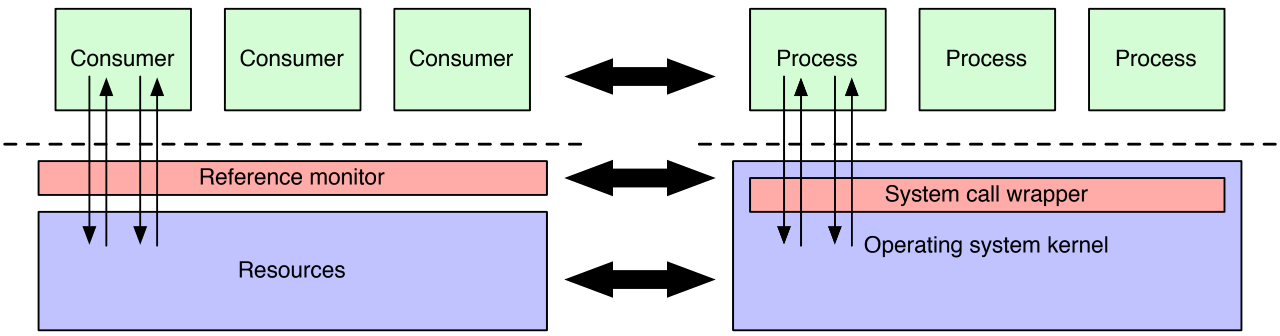



# System call wrapper vulnerabilities

- Our main case study: system call wrappers
- Popular extension technique in 1990s, 2000s
  - No OS kernel source code required
- Pre- and post-conditions on system calls
- Application sand-boxing and monitoring
  - Frameworks: GSWTK, Systrace, CerbNG
  - Almost all commercial anti-virus systems



# System call wrappers as a reference monitor



# Are wrappers a reference monitor?

- Reference monitors (Anderson 1972)
  - Tamper-proof: in kernel address space
  - Non-bypassable: can inspect all syscalls
  - Small enough to test and analyse: security code neatly encapsulated in one place
- Perhaps they count?



# … but not entirely

- No time axis in neat picture
  - System calls are non-atomic
  - Wrappers even more non-atomic with kernel
- Opportunity for race conditions on copying and interpretation of arguments and results

# Race conditions to consider

- **Syntactic races** - indirect arguments are copied on demand, so wrappers do their own copy and may see different values
- **Semantic races -** even if argument values are the same, interpretations may change between the wrapper and kernel



# Types of system call wrapper races

- TOCTTOU - *time-of-check-to-time-of-use*
- TOATTOU - *time-of-audit-to-time-of-use*
- TORTTOU* - *time-of-replacement-to-time-of-use*

* Peter Neumann has accurately described this acronym as "torturous"

# Goals of the attacker

- Bypass wrapper to perform controlled, audited, or modified system calls

```
open("/sensitive/file", O_RDWR)
write(fd, virusptr, viruslen)


connect(s, controlledaddr, addrlen)
```

- Can attack indirect arguments: paths, I/O data, socket addresses, group lists, ...



# Racing in user memory

- User process, using concurrency, will replace argument memory in address space between wrapper and kernel processing
- Uniprocessor - force page fault or blocking so kernel yields to attacking process/thread
- Multiprocessor - execute on second CPU or use uniprocessor techniques

# Practical attacks

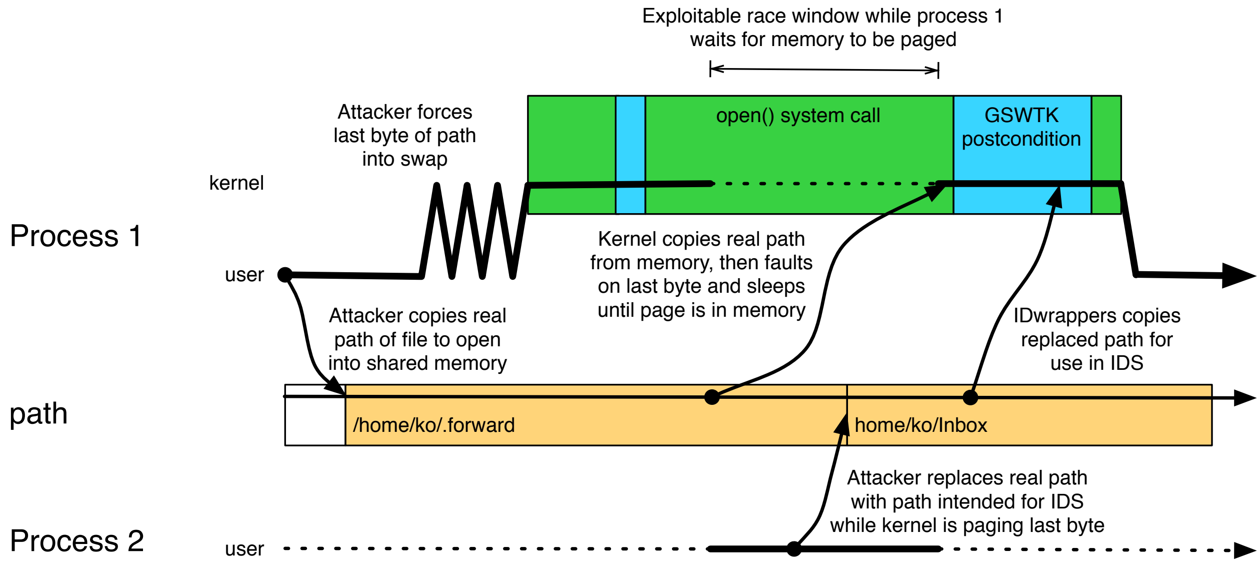
- Consider attacks on three wrapper frameworks implementing many policies
  - Systrace [sudo, sysjail, native policies]
  - GSWTK [demo policies and IDwrappers]
  - CerbNG [demo policies]
- Attacks are policy-specific rather than framework-specific



# Uniprocessor example

- Generic Software Wrappers Toolkit (GSWTK) with IDwrappers
  - Ko, Fraser, Badger, Kilpatrick 2000
  - Flexible enforcement + IDS framework
  - 16 of 23 demo wrappers vulnerable
- Employ page faults on indirect arguments

# UP GSWTK exploit

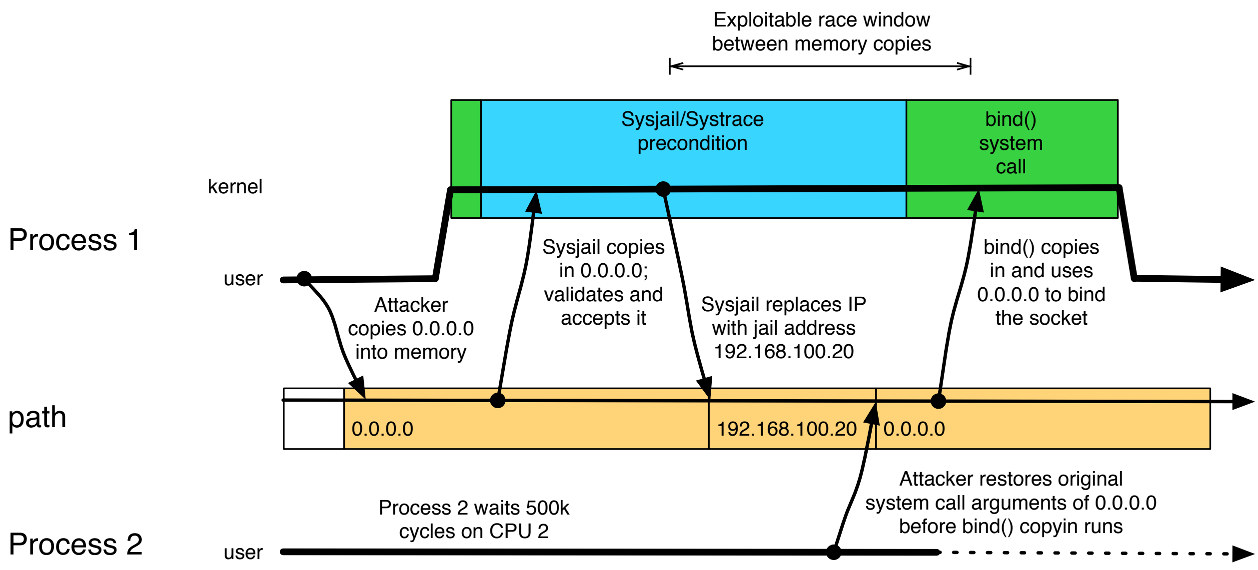





# Multiprocessor example

- Sysjail over Systrace
  - Provos, 2003; Dzonsons 2006
  - Systrace allows processes to instrument system calls of other processes
  - Sysjail implements FreeBSD's "jail" model on NetBSD and OpenBSD with Systrace
- Employ true parallelism to escape Sysjail

# SMP Systrace exploit






# Implementation notes

- OS paging systems vary significantly
- On SMP, race window sizes vary
  - TSC a good way to time attacks
  - Systrace experiences 500k cycyle+ windows due to many context switches; others faster
- Both techniques are extremely reliable

# Defence against wrapper races

- Serious vulnerabilities
  - Bypass of audit, control, replacement
- Easily bypassed mitigation techniques
- Interposition requires reliable access to syscall arguments, foiled by concurrency
- More synchronisation, message passing, or just not using system call wrappers...



# Lessons learned

- Concurrency bugs are a significant security threat to complex software systems
- Developing and testing concurrent programs is extremely difficult
- Static analysis and debugging tools are of limited utility, languages are still immature
- SMP and distributed systems proliferating

# Principles I

1. Concurrency is hard — avoid it
2. Strong consistency models are easier to understand and implement than weak ones
3. Where you must program concurrently, pick the easy path (E.g., multi-reader single-writer)
4. Prefer deterministic invalidation algorithms to time expiry of cached data



# Principles II

5. Take care not to rely on stronger atomicity than is afforded by the underlying substrate/API
6. Explicit message passing / state machines (vs. shared memory) support protocol-style analysis, formal definitions of correctness
7. Document locking or message protocols using assertions to ensure continuous testing

# Principles III

8. With side-channel-sensitive code (e.g., crypto) rely on existing carefully analysed implementations: don't roll your own
9. Remember that every narrow race window can be widened in a way you don't expect (e.g., system-call wrapper attacks)