Endokernel

A Thread Safe Monitor for Lightweight Subprocess Isolation

Yang, Im, Huang, Kaoudis, Vahldiek-Oberwagner, Tsai, Dautenhahn -- USENIX Security '24

Intro and Background

The problem

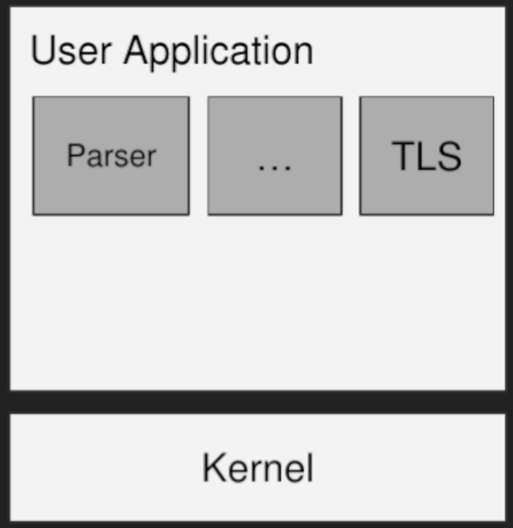
What is a Monitor

- Modern in-process isolation (ERIM, Hodor, Donky, Jenny, Cerberus) puts a **monitor inside the process** to mediate access between subprocesses, using Intel MPK for cheap memory isolation.
- Process isolation alternative gives the **whole address space** to one bug.

Privilege separation, in process

Privilege Separation with In-process Secure Monitor

- Kernel **unaware** of isolation policy and violate the policy
  - Filtering syscalls to ensure the kernel doesn't break the isolation policy in user space
- Monitor determines whether the system call is legitimate
- BUT, making the right policy decisions in **multithreading** is harder than you think
- **=>** "Secure" Monitor is **NOT** actually secure

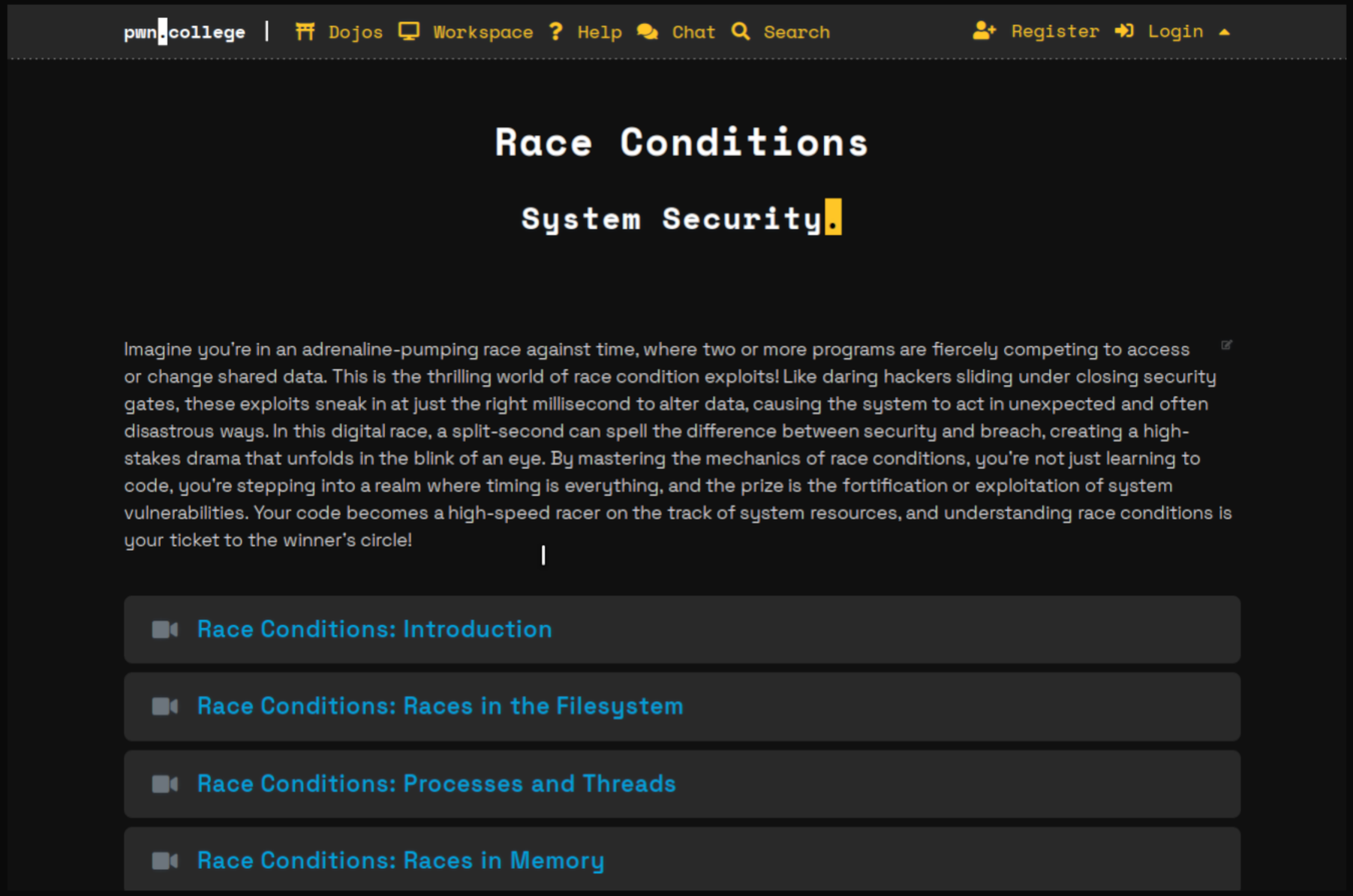


Secure Monitor **itself** becomes the problem!

3

The monitor sits in the same address space it polices, with threads the monitor itself becomes the problem.

Race conditions (in a monitor)



- Kernel locks its own state. The monitor's shadow state is **never** locked

Prior monitors don't survive threads

Existing Works Fails to Secure Multi-threaded Monitor

- Monitor needs **truth** about the system to make right decision
  - Which memory address belongs to whom? Is this file descriptor valid? …
- System states **changed** via syscalls and signals: easy if only **one** thread
- Gap: changes in state and updates in the monitor are **never synchronized**
  - The kernel maintains its internal consistency but not for the in-process monitor
  - Outdated or incorrect state will be used

Monitor makes decisions based on **incorrect** information!

5

Monitor check and kernel update are never atomic. Another thread races in the gap; the monitor sees stale state.

Memory Protection Keys (MPK)

Intel hardware feature for **page-granularity isolation inside a process**.

- 16 protection keys (pkeys), 4-bit tag per page in the PTE
- 32-bit PKRU register: 2 bits per pkey -- Access Disable / Write Disable
- WRPKRU switches PKRU in **~10 ns** -- no syscall, fully userspace
- vs. ~1 us for a process boundary -- 100x cheaper

The catch:

WRPKRU is **unprivileged**.

Anyone with code execution can flip PKRU. So a monitor must:

- Mediate every WRPKRU (binary scan, no rogue copies allowed)
- Mediate kernel paths that bypass MPK -- /proc/PID/mem, kmap_*, ...

Threat Model

- The kernel is hostile to the monitor; design as if every kernel interaction is adversarial.
- Consider the monitor is correct.
- User space programs are considered compromised and attacker has full capabilities.

Endokernel

The Solution

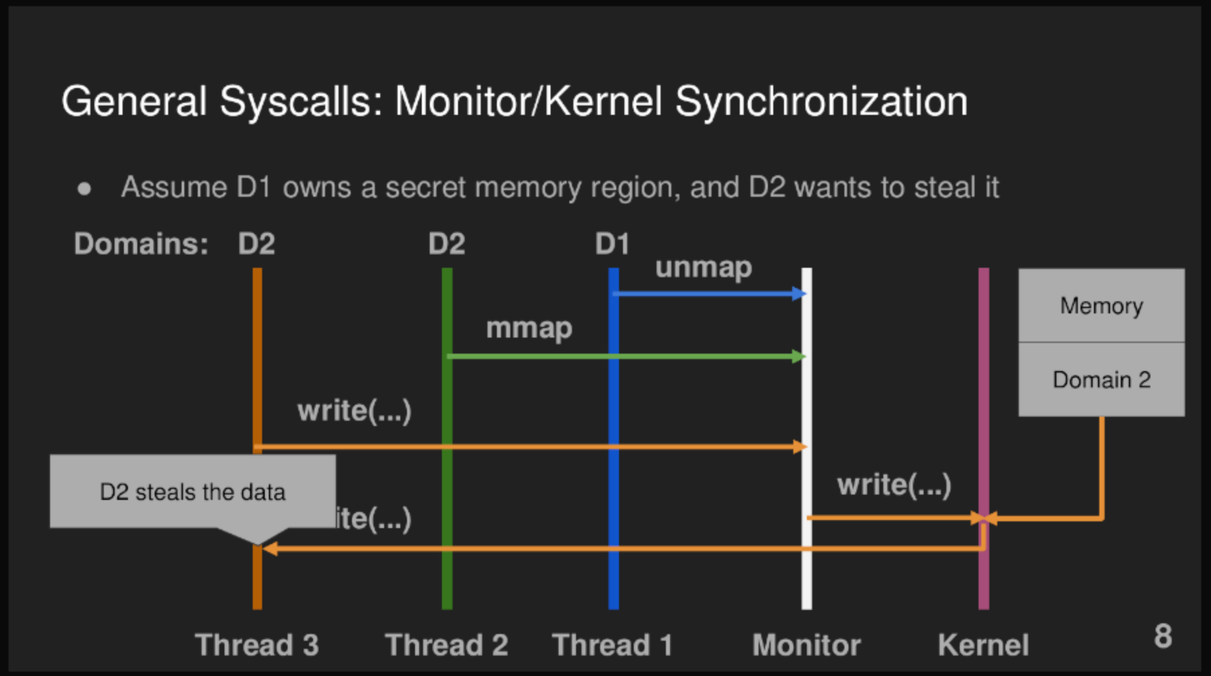
Endokernel fixes all three with one mental model:

- The monitor is a **second kernel stuck inside the process**.
- Three primitives:
  1. **Weak metadata synchronization** -- monitor perms <= kernel perms, always.
  2. **Full signal virtualization** -- Endokernel is the only kernel signal handler.
  3. **Inside-out syscall analysis** -- trace `kmap_*` kernel APIs back to the syscalls that reach them.

Endokernel architecture

- Left: Userspace monitor
- Right: Subprocesses (Sandbox, Main, Safebox)
- Red X's mark Endokernel-enforced boundaries (MPK + syscall mediation)

Challenge 1 - Syscall TOCTOU



1. D1 owns secret memory.
2. T1 (D1) unmaps; T2 (D2) mmaps same address
3. T3 (D2) writes.
4. T1's earlier in-flight write lands after the unmap -> D1's secret hits the file -> D2 reads it.

Solution 1 - Weak Metadata Sync

Solution: Weak Metadata Synchronization

- Tolerate inconsistencies before and after system calls; ensure they **only lead to inspection failures**
- Mark pages involved in system calls; block other calls that would change their properties while the memory is in use
- Allow concurrent invocation of system calls if they don't alter page properties
- Ensure **correct decisions** are made, even with Kernel-Endokernel inconsistencies, without violating policy.

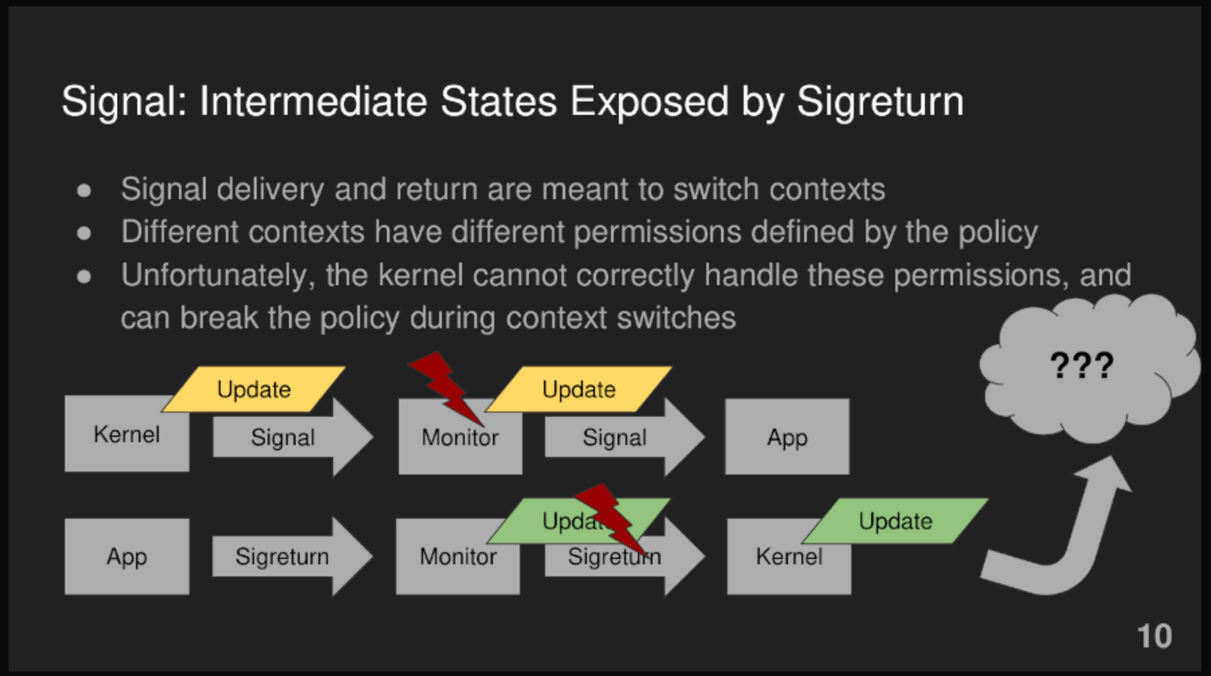
Desynchronization **never** violates security policies

9

Invariant 1: monitor's recorded perms <= kernel's actual perms.

- `mprotect` pattern - lock record, drop privilege, syscall, restore.
- Other threads only ever see a more restrictive view: a race causes a denied access, never a leak.

Challenge 2 - sigreturn bypass

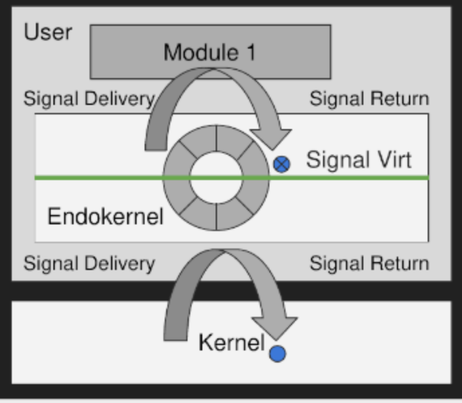


- Yellow Updates are PKRU changes. Red flashes are racy steps.
- The sigframe lives on a stack the attacker may control - forge it and sigreturn restores an arbitrary PKRU.

Solution 2 - Virtualize the signal path

Solution: Fully Virtualized Signal

- Endokernel acts as a **middleware**
- Endokernel receives signals from the kernel
  - Stores signals in a pending queue
  - Returns control to the kernel with sigreturn
- Endokernel delivers signals to the user
  - Creates a new sigcontext and sigframe.
  - Simulates the user's sigreturn syscall



Virtualized secure and compatible signals

11

- Endokernel is the only kernel signal handler.
- Kernel signals are queued and immediately sigreturned back to the monitor
- it synthesizes the user-facing signal on a sigaltstack page only it can write.

Challenge 3 - Highmem bypass

Highmem: Bypass Pattern and Delayed Memory Access

- Various triggering mechanisms
  - /sys/kernel/tracing/user_events_data
  - Process_vm_readv, Sendmsg with MSG_ZEROCOPY
- Access physical pages with high memory and bypass permission check
  - Some code paths checked
    - __get_user_pages -> check_vma_flags -> arch_vma_access_permitted
  - Nonetheless, sendmsg delayed the memory access
  - The MMU may change after the check

12

- /proc/PID/mem, sendmsg(MSG_ZEROCOPY, etc. all reach kmap_* and bypass MPK.
- sendmsg's NIC reads pages **after** the syscall returns - the window between mmap and mprotect_pkey is exploitable.

Solution 3 - Inside-out syscall analysis

1. Start **inside the kernel** - every kmap_* call site.
2. Walk **back out** to the syscalls that reach it.
3. Restrict / virtualize the offenders at the boundary.

Introduce five new attack patterns:

- Incorrect signal-return handling
- Fork-and-retry attack
- High memory access abuse
- TSX instruction probing
- Monitor / kernel inconsistency via syscall

Evaluation

Setup

Hardware / OS

- Intel **i7-1165G7** (11th gen), 4 cores @ 2.8 GHz
- Boost off, hyper-threading off, 16 GB RAM
- Ubuntu 20.10, kernel 5.9.8 with CET + SUD backports
- Kernel patch: **+82 / -45 LoC** (sigaltstack panic fix)

Configurations measured

(ablation)

1. nex-sec = nexpoline + seccomp
2. nex-sud = nexpoline + syscall user dispatch
3. cet-sec = Intel CET + seccomp
4. cet-sud = Intel CET + syscall user dispatch
5. strace1 = baseline

Per-application overhead

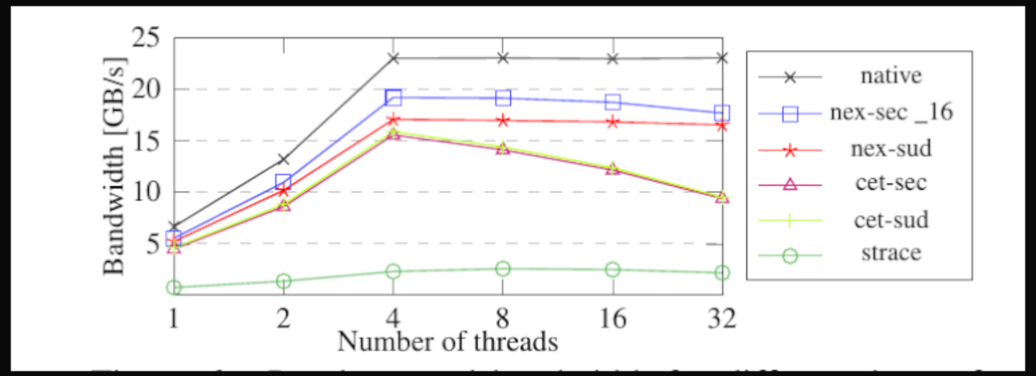
Network-bound (lighttpd, nginx) within ~5%; **nex-sud ~1%**. File / syscall-heavy: zip 4-17%, sqlite3 13-36%. **curl** worst -- 130k writes + 30k rt_sigaction => 26-56%. strace 1-2 orders worse (501% on sqlite3).

Why curl pays so much: every syscall crosses the monitor boundary. curl issues ~130k writes + ~30k rt_sigactions per 1 GB transfer; nginx amortizes via its event loop and so barely notices.

DUBIOUS

- Only 5 programs part of the ablation study

Thread scalability



Optimal at 4 threads; beyond, per-object monitor locks (file table, VMA list, signal state) start to bottleneck on futex contention.

1. **nex-sec_16**: 16% off native at 1T -> 23% at 32T
2. **cet-*** configs degrade up to ~60% at 32T

DUBIOUS

- Works great with a few threads

The Linux Test Project (LTP)

The kernel's own regression test suite, ~2000 tests covering:

- Syscalls (open, read, write, mmap, mprotect, ...)
- Signals (delivery, masking, sigaltstack, sigreturn)
- Threads, processes, fork, exec
- File system, networking, memory management

Why test Endokernel against it?

- Endokernel is wedged between user and kernel.
- Any monitor bug surfaces as a deviation from kernel-expected behavior.
- LTP is the most adversarial test of does the kernel still work as expected.

LTP results

- 1926 tests, 78 failed, ~95% pass

Lack of kernel functionality (PKEY of shared mem)	33
Security implication (Endokernel intentionally blocks)	19
Inconsistency of kernel functionality (edge cases)	11
Secondary loader / O_EXEC	5
Issues caused by Endokernel	5
Unsupported syscalls	2
Racing causes by the kernel (hugemmap06)	1

- **Zero** failures are signal- or threading-correctness bugs.
- The contributions Endokernel claims (thread-safe signals and metadata), pass cleanly.

Takeaways

Pros

- First in-process monitor that **survives multi-threading**
- **~5%** overhead on web workloads (nex-sud); **~23%** at 32 threads

Cons

- In-place policies can still be bypassed (TSX, future kmap callers, ...)
- Overengineering is a disease; TCB grows with every virtualized subsystem
- Recreating the kernel in userspace is hardly optimizable or extensible
- Hypervisors exist (e.g. Xen para-virtualization) - isolation has cheaper homes

References

- **Paper** (www.usenix.org/conference/usenixsecurity24/presentation/yang-fangfei)
- **Source code** (github.com/endokernel/endokernel-paper-ver)
- **Test Suite** (https://github.com/endokernel/test/)

Related Work

- ERIM
- Hodor
- Donky
- Jenny
- Cerberus
- uSWITCH

Questions?

Questions?