Stellux: Exploring Fine-Grain Privilege Access in a Clean-Slate Kernel

Albert Slepak
Boston University (student)
Boston, Massachusetts, USA
aslepak@bu.edu

Thomas Unger
Boston University
Boston, Massachusetts, USA
tommyu@bu.edu

Jonathan Appavoo
Boston University
Boston, Massachusetts, USA
jappavoo@bu.edu

Abstract

Executing code with hardware privilege is inherently risky. The desire to mitigate this risk was a major motivation for exploring alternative kernel architectures, such as Microkernels [2][1] and Exokernels [3]. Everybody knows that operating system kernels execute with hardware privilege and applications do not – more formally, there exists a coupling between codebases (operating system / application) and modes of execution (supervisor / user). Because we have accepted this coupling as inviolable, our options for separating privileged and unprivileged execution have been limited to a binary, course-grained, design-time decision: whether specific functionality should reside within the application or the kernel codebase (and thus, the corresponding executable programs). This constraint has limited our approach to separating privileged and unprivileged execution.

We believe it is time to challenge the received wisdom of this coupling and to explore building systems that make privilege access an independent variable, i.e., making it possible to transfer privilege without transferring codebase. As applied to the challenge of minimizing the trusted compute base, we believe it offers an easy-to-use tool for removing large portions of code from privileged execution, turning the present approach from a cleaving into a dissection.

Our lab previously developed a mechanism that offers a fine-grained way for application threads to toggle in and out of hardware privilege execution at runtime. This mechanism is called Dynamic Privilege[4] (see Figure 1), and credentialed application threads can access it via a system call.

Let's consider a concrete example to illustrate the opportunity afforded by Dynamic Privilege to reduce privilege code execution. Page-table code traditionally lives in the kernel. While there is the occasional instruction that requires hardware privilege (such as a read or write of a control register, the invalidation of a page, and enabling/disabling interrupts), much of this code simply traverses trees in memory and performs bit-operations. With Dynamic Privilege, a credentialed user process (perhaps written in C, C++, Rust or others) can make a system call to access privilege, then execute a privileged instruction to read the register holding the base of the current page table (such as `MOV CR3, RAX` on x86_64). The thread can then `memcpy` the page table into its own address space where it transfers back to user privilege to operate on that datastructure. When ready, the thread transfers back into kernel privilege just long enough to `memcpy` the table into kernel memory, or perhaps update the page table base register. This is an example of how we propose more carefully dissect out code that actually needs to run with privilege.

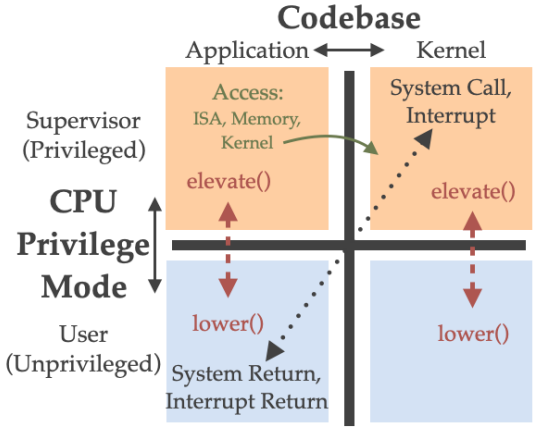

Figure 1: Black: Traditional OSs *couple* privilege level changes with codebase switches between application and kernel; atomic transfer instructions crystallize this coupling in hardware. Red: Dynamic Privilege *decouples* these variables, allowing independent mode transfers *at runtime*. Green: This enables low-level hardware access to the extended ISA and memory.

Equipping the Stellux kernel with Dynamic Privilege from the outset allows deferring decisions about the execution mode of a codepath to runtime. This alleviates the burden of making design-time decisions and enables incremental prototyping. Codepaths that would traditionally be executed in privileged mode can be gradually segmented and shifted to run more in user mode. Specifically, the ability to switch privilege at runtime leads to a further reduction of privileged code execution, as the kernel can run mostly in a "lowered," unprivileged state and "elevate" itself only to perform instructions that are privileged due to hardware requirements. While previous work demonstrated the feasibility of dynamic privilege switching by retrofitting it into existing operating systems like Linux, our current work differs by building a clean-slate operating system, StelluxOS, from the ground

up with dynamic privilege as a foundational principle. By integrating dynamic privilege switching from the inception of OS design, we can fundamentally rethink and reshape the architecture without being hindered by legacy constraints. This allows us to further reduce the use of privileged execution and explore new paradigms in OS construction, such as reorganizing kernel components to run predominantly in unprivileged mode and elevating privileges only when absolutely necessary due to hardware requirements.

Our case study focuses on building an operating system from scratch, aiming to extract as much of the kernel as possible into user space through the "lowering" mechanism. Only the bare minimum is performed in the privileged domain, such as setting up the Global Descriptor Table (GDT) and enabling syscalls. Afterwards, the kernel immediately lowers itself into an unprivileged state to continue full initialization and schedule further tasks. In our current prototype, we have demonstrated that we can perform the following in user mode, elevating privileges in a controlled manner only for specific privileged CPU instructions: setting up and installing the Interrupt Descriptor Table (IDT), mapping and initializing the Local APIC, configuring and starting the High-Precision Event Timer (HPET), initializing the kernel heap allocator and page frame allocator, setting up the scheduler, and bringing up secondary cores. Additionally, it has been extremely easy to move kernel components from the privileged domain into unprivileged mode, achievable by modifying only a couple lines of code. Even at this early stage of prototype development, this model has significantly reduced the use of the privileged execution domain compared to any existing operating system design. We anticipate that this approach will yield more significant benefits in the later stages of operating system development. By sandboxing the majority of kernel components, the system can achieve enhanced fault tolerance, as failures within non-privileged kernel components would not compromise the entire system. This approach could also simplify the development process by allowing kernel modules to be developed and tested in user mode, leveraging existing user-space debugging and testing tools.

Our work aims to re-examine long-held assumptions about operating system privilege models. By experimenting with dynamic privilege level switching within the kernel, we hope to uncover new insights and methodologies for building more robust, secure, and flexible operating systems. While challenges remain in departing from traditional models, we believe this line of inquiry is a promising avenue for future research and development in operating systems principles.

.

References

[1] Steven Hand, Andrew Warfield, Keir Fraser, Evangelos Kotsovinos, and Daniel J Magenheimer. 2005. Are virtual machine monitors microkernels done right?. In *HotOS*.

[2] Gernot Heiser and Kevin Elphinstone. 2016. L4 Microkernels: The Lessons from 20 Years of Research and Deployment. *ACM Trans. Comput. Syst.* 34, 1, Article 1 (apr 2016), 29 pages. https://doi.org/10.1145/2893177

[3] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. 1997. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (Saint Malo, France) *(SOSP '97)*. Association for Computing Machinery, New York, NY, USA, 52–65. https://doi.org/10.1145/268998.266644

[4] Thomas Unger. 2024. *Dynamic Privilege*. Ph. D. Dissertation. Boston University, Boston, MA.