What are your OS isolation mechanisms not telling you?

Sidhartha Agrawal, University of British Columbia, Canada

1 Motivation

New isolation mechanisms are constantly emerging and are often motivated by one of: the emergence of a new use case, improving the performance of an existing use case, or defending against a security vulnerability. However, these new mechanisms always use isolation as a tool. For example, they reserve resources (memory, storage, CPU time) [4], restrict access to unneeded state (kernel) [8], or share underlying state (drivers) [1, 2] to improve performance. Similarly, they increase the isolation of resources or underlying state to build defenses. Given the importance of varying isolation, it is imperative to have a clear understanding of which resources and state are shared when using any particular isolation mechanism. However, we lack a precise vocabulary to describe this sharing and isolation, which leads to security vulnerabilities.

Even when applications appear isolated, such as in the case of containers, they still share kernel state. For instance, Linux namespaces [6], which are a building block for containers, do not isolate all the kernel's visible state; some state can leak across container boundaries (e.g., the open file table) leading to denial-of-service attacks on other applications on the same host [9]. Additionally, since the container infrastructure and the kernel run in the same address space, a simple buffer overflow in one part of the kernel can bring down the shared kernel and both containers. Lightweight VMs such as FireCracker [1] and KataContainers [7] are more secure alternatives to Linux containers, providing the security of VMs with the overhead of containers. However, they achieve this performance by having the host OS (rather than the guest OS) provide functionality (e.g., drivers) for all VMs. Unfortunately, this leads to more shared state in the host kernel. Just as a shared kernel exposed issues with state leakage (for containers), shared drivers in the host kernel can do the same (for VMs). [1]

2 Solution

We present OSmosis, which consists of three parts. First, a mathematical *model* that enables us to precisely describe the resources in the system and how they are shared. Second, a framework that identifies the features needed by an operating system to build isolation mechanisms based on the model. Third, an implementation of the framework on seL4 [3].

[1]Keywords: microVM, containers

Model: In *OSmosis*, the running system consists of a set of *protection domains* and *resources*. Protection domains correspond to active entities such as processes, threads, and virtual machines. A protection domain has a set of resources and a *resource directory*. Resources can be either physical or virtual entities (e.g., virtual memory region, file, socket) that can be partitioned into smaller resources. The resource directory is a dictionary that identifies the protection domain responsible for satisfying a request for a resource that the current protection domain does not possess. For example, a user-level process wanting to allocate some memory will call `mmap()` to request more virtual memory resources. This corresponds to a lookup in the PD's resource directory for virtual memory resources and then requesting more virtual memory from the PD to which that resource maps. The resource relation describes dependencies between two resources (e.g., the page table keeps track of which virtual memory page depends on which physical page).

Framework: The *OSmosis* framework consists of three building blocks an OS needs to realize the model. First, we need a protection domain abstraction, which is the explicit owner of a set of resources. Second, we need a way to extract the resource relation from the system to create the system's dependency graph. This information is often already present; for example, page-tables describe how virtual addresses depend on physical addresses. And finally, we need an API that instantiates a PD's resource directory and resources as per the model. This is inspired by the `clone` system call in Linux [5] and is primarily syntactic sugar for the developer to create PDs with desired isolation properties easily.

Implementation: We have implemented a small portion of *OSmosis*– dealing with memory resources – using the capabilities-based microkernel seL4 [3]. We chose this microkernel as it has no existing abstractions for processes, containers, or virtual machines. This lack of existing abstractions allows us to define the building blocks as we see fit. We extract the model state at runtime and import it into Neo4j, which is where the queries are eventually run.

3 What does this enable?

Viewing the systems as a collection of resources and relations enables us to define queries on the model state, which is a DAG (with PD and resources as nodes and resource relations as edges). These queries can be used to precisely *compare* the level of isolation between two PDs. For instance, if we take the transitive closure (i.e. BFS) of the resource relations starting at a PD, we get a set of all the resources on which a

PD depends. Alternatively, an operator can convert threat models into graph queries to check which resources are vulnerable to comprise by which other PDs. This can help determine the minimum trusted computing base of a given deployment.

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*. https://www.usenix.org/conference/nsdi20/presentation/agache

[2] Simon Kuenzer, Vlad-Andrei B*u*adoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi R*u*aducanu, Cristian Banu, Laurent Mathy, R*u*azvan Deaconescu, Costin Raiciu, and Felipe Huici. 2021. Unikraft: Fast, Specialized Unikernels the Easy Way. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)*. Association for Computing Machinery. https://doi.org/10.1145/3447786.3456248

[3] Online. 2022. The seL4 Microkernel. (2022). https://sel4.systems/

[4] Online. 2023. Linux: Cgroups. (2023). https://man7.org/linux/man-pages/man7/cgroups.7.html

[5] Online. 2023. Linux: Clone. (2023). https://man7.org/linux/man-pages/man2/clone.2.html

[6] Online. 2023. Linux: Namespaces. (2023). https://man7.org/linux/man-pages/man7/namespaces.7.html

[7] Alessandro Randazzo and Ilenia Tinnirello. 2019. Kata Containers: An Emerging Architecture for Enabling MEC Services in Fast and Secure Way. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. https://doi.org/10.1109/IOTSMS48152.2019.8939164

[8] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. https://www.usenix.org/conference/atc20/presentation/shillaker

[9] Nanzi Yang, Wenbo Shen, Jinku Li, Yutian Yang, Kangjie Lu, Jietao Xiao, Tianyu Zhou, Chenggang Qin, Wang Yu, Jianfeng Ma, and Kui Ren. 2021. Demons in the Shared Kernel: Abstract Resource Attacks Against OS-Level Virtualization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery. https://doi.org/10.1145/3460120.3484744