Musings

RIK FARROW

Rik is the editor of ;login:.
rik@usenix.org

Imagine you are charged with defending the security of one or more systems, yet must also allow other people to run the code of their choice on your systems. I could be talking about your web browsers, those sources of malware infections, but my focus is actually public clouds.

Public clouds offer customers the ability to run any software that is not openly hostile through behaviors like port scanning or launching denial-of-service attacks. That leaves a lot of leeway for various mischief on the hosts they run on, especially if those hosts are running containers or cloud functions—lambdas in AWS-speak.

The initial way of strengthening security for containers was to run each customer's containers on top of a kernel running in a virtual machine (VM). VMs rely on hardware-based security, and while Sun, HP, IBM, and SGI had hardware support for VMs on or before the 1990s, Intel and AMD support appeared in 2005. Hardware support placed guest operating systems in a privilege ring above the virtual machine monitor (VMM), meaning that the VMM had control over the hardware and its treasures: CPUs, memory, storage, and networking.

But running containers inside of VM guests means that cloud vendors lose a lot of what they wanted to gain from container technology. They can't start up containers wherever they want to, as they are constrained by a customer's VM instances. And VMs are slow to start, taking seconds, and require at least an order of magnitude more memory and other resources than lightweight containers. Thus began a quest for more efficient ways of hosting containers.

One early example was NEMU, a stripped down version of QEMU, the open source system emulator. NEMU runs as a process, like QEMU, but instead of having all of the capabilities of QEMU, NEMU dispenses with support for things you can't use in the cloud, like USB, as well as most other devices and hardware emulation, making NEMU smaller and simpler than QEMU. Both QEMU and NEMU are type two hypervisors.

AWS and Google have created their own type two hypervisors, each with the goal of dispensing with VMs for isolating containers and cloud functions/lambdas. Like NEMU, each solution catches accesses to the underlying system, and each limits access using `seccomp()` to reduce the number of system calls that can be made to the host from the hypervisors. You can download the source code to both hypervisors from GitHub if you want to, as both are open source. But the way each has been designed is quite different.

You can read the Firecracker paper [1], presented at NSDI '20, for more on motivation and the deeper details. I found myself fascinated that the paper's authors talk about running more than 8,000 instances on a high-end server, so as to maximize the use of total physical memory, multiple CPUs, and NIC queues.

Firecracker is written in Rust, and like NEMU, provides a limited, virtual system on top of Linux KVM. Firecracker uses `seccomp` to limit the number of system calls to 24 and 30 `ioctl`s. Firecracker, like Docker, also relies on some of the same features for isolation, such as namespaces, cgroups, and dropping of privileges. Firecracker provides support for the container or lambdas being run by including a stripped-down Linux kernel. Instead of taking seconds to

boot, all this support structure can be running in less than 200 milliseconds. And Firecracker uses more than an order of magnitude less memory and storage than a VM-based approach.

Google programmers used Go, another language that provides strong typing and dynamically allocated and reclaimed memory, like Rust. The Google system, named gVisor, consists of two processes. The first, a hypervisor called Sentry, emulates a Linux system call interface sufficient to run most cloud functions or containers. Like Firecracker, Sentry needs to make system calls to the host, and uses a less severely restricted set at 68. Sentry has its own `netstack` to implement TCP/IP, unlike Firecracker, which uses its guest Linux kernel for the network stack. While Firecracker does away with access to the host file system, instead creating an image file like a VM, gVisor uses a helper application, called Gofer, to handle file system access.

The Firecracker paper doesn't include performance comparisons to gVisor, but a paper by Anjali et al. [2] examines both performance and a measure of security of LXC (native Linux containers), gVisor, and Firecracker. Anjali et al. use microbenchmarks to compare these three container solutions, along with Linux without containers. They report that the Firecracker has high network latency, while gVisor is slower at memory management and network streaming. gVisor is also much slower when it comes to opening and writing files. For security, the paper authors look at code coverage in the Linux kernel including KVM, with the assumption that an isolation solution that relies on more lines of kernel code, running at the highest privilege level, is less likely to be secure due to the potential for kernel bugs. Firecracker does rely less on the underlying Linux kernel, but not by much, using 9.59% of the kernel's 806,318 lines of code versus 11.31% for gVisor.

There are other approaches for isolating containers and cloud functions. Library OSes, also called unikernels, rely on building an application that includes the needed operating system support, and can run on bare metal or on top of a hypervisor like KVM. I ran across Nabla while reading [2] and discovered that Nabla is based upon MirageOS, a unikernel system written in OCaml. Using Nabla requires that the library OS be linked with the application, something I considered a roadblock back when I learned of unikernels [3, 4]. But Nabla was supposed to have a simple, three-step build process, and I tried the example for running "Hello, World!" The build failed at the second step, unable to find `seccomp.h`, even though there were copies of `seccomp.h` handy on my system, including one downloaded for the build.

AWS and Google know that many organizations prefer to build their apps using JavaScript and Python, and though that's possible using unikernel approaches, Firecracker and gVisor are designed to *just work*, as if you were running within a VM running Linux.

The Lineup

We start out this issue with an article based on the FAST '20 paper on Optane performance. Intel Optane, previously known as 3D Xpoint, can be used as main memory or in SSDs, and in the article, Yang et al. use microbenchmarks to tease out the performance characteristics of a system endowed with Optane DIMMs alongside ordinary DRAM, with the hardware support for making data flushed from CPU caches persistent even if power is interrupted.

Zahn et al. wrote "How to Not Copy Files" for FAST '20, and besides being curious about the paper title, I wondered just what was special about their approach—and what was wrong with how other file systems handle file copying. File copying is more important than ever in current systems, with copy-on-write (CoW) being used to speed up file cloning often used with containers. Zahn et al. demonstrate how BetrFS is better and faster at file cloning than any of the current Linux file system favorites while describing their modified B-epsilon trees.

I took advantage of my temporary access to Dick Sites, who wrote about his KUtrace tool in the Summer 2020 issue [5], to ask him some more questions. Honestly, there were a lot more I would have liked to have covered, as Sites has had an insider's view of developments in compilers and CPU architecture since 1966, but at least we dealt with several areas and provided pointers to where you can learn more.

Zhu et al. had an interesting paper about superpages in the Linux kernel. I had heard that superpage support should be disabled, and wondered just what the problem was with something that should increase the performance of memory-hungry applications. It turns out that the answer is complicated, but Zhu and his co-authors do a very good job of explaining the issues while presenting their own solution to improving superpage problems on Linux.

I had wanted to get a couple of the authors of Firecracker and gVisor to write for this issue, but that didn't work out. I did run across a fascinating technical report about cloud programming, and interviewed one of the authors, Ion Stoica, about issues raised in that report. While the cloud does abstract the details of operations, programming in the cloud mostly means microservices today, something very different than what most programmers have been taught how to do.

Georg Link offered to write about open source health: for example, how can you tell if an open source project is healthy enough to be around in five years? The answer to that isn't easy to figure out, but Link provides good suggestions about what he and the Linux Foundation's CHAOSS Project look for when determining health.

Uta et al. add to the understanding of how large clusters work by contributing time-series data of a datacenter in the Netherlands. They call this data MRI-like because it does allow analysis in multiple dimensions. Their contribution, and that of their organization, differs from other contributed traces of large clusters because of the types of applications being run in their DC.

Gómez-Iglesias et al. explain how Intel CPU bugs with names like Meltdown and Spectre are actually likely to affect systems. The authors explain what it takes to carry off a successful attack, the various ways that systems can be patched, and the trade-offs associated with the different mitigations, mainly loss of performance.

Anatoly Mikhaylov shares his experience in using tagging and OpenTrace to connect requests coming in to a service with database performance issues. Associating a particular request with an unusually slow SQL query isn't easy, because of the intermediate layers found in today's software architecture. Mikhaylov, who works at Zendesk, explains how he and coworkers have worked out how to do this cleanly.

Laura Nolan, reacting to Black Lives Matter, writes about how SREs and other technologists can contribute to changing how people of color are treated. Nolan suggests actions that include changing technical language, being aware, and making changes that are within your sphere of action.

Cory Lueninghoener presents the first installment in his column named "Systems Notebook." Lueninghoener describes how he and a group of coworkers avoided failure in the design of a new system management stack. Instead of plugging away in isolation and later presenting their new system, his group decided to involve others at his site to avoid problems down the road with missing features and lack of acceptance because they had excluded interested parties.

Dave Josephsen continues with his examination of eBPF. Josephsen begins with mythical lovers, forced to communicate through a crack in a wall. He compares this to communications between BPF within the kernel and its Python stub in userspace, and describes three ways that this communication can occur.

Terence Kelly, in his new column, "Programming Workbench," focuses on a locking technique that often gets mentioned but has been poorly documented. Kelly explains hand-over-hand locking, an easy-to-understand method for protecting data structures, such as linked lists, on systems with multiple threads. Kelly plans to continue on this theme, providing code examples in C for interesting algorithms that deserve more exploration.

Simson Garfinkel launches his own column, "SIGINFO," with some history involving his current place of work. Garfinkel begins with the story of how we wound up with 80-column terminals, covers UNIX's "everything is a file" concept, and winds up tying Multics segments to NVRAM. Garfinkel has a long history of writing, and he loves to get his research right as well.

Dan Geer, working solo this time, considers questions we should be asking about security in the time of the coronavirus. Geer, whose column focuses on metrics and measuring security, takes a deep look at how the pandemic has changed not just the way we work, but also the threats our computer systems and networks now face.

Robert Ferrell contrasts working from home with working remotely. He's done both and suggests that one is definitely more comfortable and sensible than the other.

Mark Lamourine has reviewed three books this time, one about algorithms, another concerning skepticism, and the third about re-engineering legacy software. I review a book of stories about interesting things, often failures, that happened to IT architects and the resulting build outs.

I once asked a professor why there weren't any papers about new operating systems at the SOSP we were attending. His answer was succinct: operating systems are hard. I think it is also hard to create ways to protect those operating systems from the software running above them, doing so in ways that are performant but also should remain secure. When I learned about Firecracker late in 2019, I started studying the current methods, from unikernels to system calls reimplemented in Go. Just as VMs and containers have their place in the clouds of today, so do cloud functions and lambdas, and for these to work efficiently they need to be secured with lightweight technology.

I don't think we have heard the last of developments in this area.

References

[1] A. Agache, M. Brooker, A. Florescu, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D. M. Popa, "Firecracker: Lightweight Virtualization for Serverless Applications," in *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*: https://www.usenix.org/conference/nsdi20/presentation/agache.

[2] Anjali, T. Caraza-Harter, and M. M. Swift, "Blending Containers and Virtual Machines: A Study of Firecracker and gVisor," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20)*: https://dl.acm.org/doi/pdf/10.1145/3381052.3381315.

[3] A. Kantee and J. Cormack, "Rump Kernels: No OS? No Problem!" *;login:*, vol. 39, no. 5 (October 2014): https://www.usenix.org/publications/login/october-2014-vol-39-no-5/rump-kernels-no-os-no-problem.

[4] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the Library OS from the Top Down," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*: https://dl.acm.org/doi/abs/10.1145/1950365.1950399.

[5] R. L. Sites, "Anomalies in Linux Processor Use," *;login:*, vol. 45, no. 2 (Summer 2020): https://www.usenix.org/publications/login/summer2020/sites.