Security for the Modern Age

JESSIE FRAZELLE

SECURELY RUNNING PROCESSES THAT REQUIRE THE ENTIRE SYSCALL INTERFACE

Research for Practice combines the resources of the ACM Digital Library, the largest collection of computer science research in the world, with the expertise of the ACM membership. In every RfP column experts share a short curated selection of papers on a concentrated, practically oriented topic.

When deploying applications in the cloud, practitioners seek to use the most operable set of tools for the job; determining the "right" tool is, of course, nontrivial. Back in 2013, Docker won the hearts of developers by being easy to use, but Linux containers themselves have been around since 2007, when cgroups (control groups) were added to the kernel. Today, containers have spawned a large ecosystem of new tools and practices that many professionals are using on a daily basis. The foundational technologies making up containers are not new, however. Unlike Solaris Zones or FreeBSD Jails, Linux containers are not discrete kernel components built with isolation in mind. Rather, Linux containers are a combination of technologies in the kernel: namespaces, cgroups, AppArmor, and SELinux (Security-Enhanced Linux), to name a few.

Containers are not the abstraction an application developer typically encounters today. The trend is toward functions and "serverless," allowing the user to run a single function in the cloud. Because of the way applications and functions are run in the cloud, there will likely be a new generation of isolation techniques built around running a

single process securely in an easy and minimal way.

While evidence has shown that "a container with a well-crafted seccomp (secure computing mode) profile (which blocks unexpected system calls) provides roughly equivalent security to a hypervisor" (https://blog.hansenpartnership.com/measuring-the-horizontal-attack-profile-of-nabla-containers/), methods are still needed for securely running those processes that require the entire syscall interface. Solving this problem has led to some interesting research.

Let's take a look at some of the research being done in these areas.

VIRTUAL MACHINES VERSUS CONTAINERS

My VM is Lighter (and Safer) Than Your Container; Filipe Manco et al.
https://dl.acm.org/citation.cfm?id=3132763

Containers became popular as an alternative to VMs (virtual machines) because they are better in the areas of fast boot, small memory overhead, and allowing high density on a single machine. This paper explores creating VMs that meet those same requirements, along with the container features of pause and unpause.

Taking into consideration that the required functionality for most containers is a single application, the authors explored unikernels (minimal VMs where the operating system is linked directly to the application) and TinyX (a tool to create minimal Linux distributions for an application). The

smaller the VM image is, the smaller the memory footprint will be and the faster the image will boot.

For containers, just like a typical process running on a host, the number of processes or containers you start does not affect the time to start them, given the usual caveats about resources not being infinite, even in the cloud. This is not true for VMs. The overhead to start a VM increases as more of them are run. The authors found, in the case of Xen, that this is a result of both device-creation time and interactions with the XenStore. The authors implemented their own LightVM to solve a lot of the algorithmic and design problems they found with Xen.

The result of their efforts are minimal VMs that can be booted in as little as 2.3 ms. A standard Linux process starts in about 1 ms, and a docker container starts in about 40 ms, depending on the size of the image. The boot time remains constant the more VMs are launched, which is in stark contrast to typical VMs. Unikernels, however, are not as easy to create as containers and require individual development time to be made functional for each application.

ISOLATION OF APPLICATIONS IN A MINIMAL WAY

Unikernel Monitors: Extending Minimalism Outside of the Box; Dan Williams and Ricardo Koller
https://dl.acm.org/citation.cfm?id=3027053

Minimal software has the benefits of reducing attack surface and making software more understandable with less overhead. Unikernels are frequently discussed in the context of minimal and secure ways to run programs in

the cloud. In the traditional approach a unikernel is a VM and, as such, is run in a VM monitor, which is a program that watches and controls the lifecycle of VMs, such as VMWare, QEMU, or VirtualBox. Unikernel monitors are bundled into the unikernel. This creates a minimal way to boot unikernels without the added complexity of using a stand-alone VM monitor.

Most VM managers/monitors are heavyweight, with features for devices that are not used in modern or cloud environments. Take QEMU, for example: it comes with the emulation for devices such as keyboards and floppy drives. If there is an exploit in the floppy-drive emulator, it is game over for the whole system, even though a floppy drive obviously has no usefulness in the cloud.

If a monitor is purpose-built for booting unikernels, its computing base is much more minimal than the VM monitors in use today (about five percent of the size). The authors of this paper created a monitor that has only two jobs: creating the isolation to run the unikernel and performing actions when the unikernel exits. The monitor is also baked into the executable for the unikernel, creating a simplistic and minimal approach for distributing and executing unikernels.

The boot time for their prototype was 10 ms, which is eight times faster than a traditional monitor. This paper has a positive vision of the future, running applications in a minimal and secure way in the cloud. IBM recently released a container runtime called Nabla (https://nabla-containers.github.io/) around the topics and implementations of this paper.

VIRTUALIZE AT THE RUNTIME LAYER

Alto: Lightweight VMs using Virtualization-Aware Managed Runtimes; James Larisch, James Mickens, and Eddie Kohler
https://mickens.seas.harvard.edu/files/mickens/files/alto.pdf

Traditional virtual machines, like Xen, virtualize at the hardware layer. Docker, on the other hand, virtualizes at the POSIX layer. This paper suggests a new approach to virtualize at the runtime layer.

One of the harder questions in this space is how to handle state. In traditional environments, state for the file system and network is handled in the kernel. The authors suggest moving as much kernel state as possible into the virtual machine through a user-space networking stack and FUSE filesystem. They also suggest explicitly depicting each state object as an addressable server (each with its own IP address), allowing operators to easily migrate and update applications since there is clean separation of a program's code, stack, and heap.

Through innovations in memory allocation, garbage collection, and managing state, Alto seems to be the closest solution to securing processes minimally while giving a new set of controls to operators. As someone who has spent quite a bit of time thinking about the problems faced by creating a minimal, virtualized container runtime, I truly enjoyed the problem statements and solutions this paper laid out.

DETERRING ATTACKERS IN YOUR APPLICATION

Chaff Bugs: Deterring Attackers by Making Software

Buggier; Zhenghao Hu, Yu Hu, and Brendan Dolan-Gavitt
https://arxiv.org/abs/1808.00659

Defense of software and systems usually consists of correcting bugs that can be exploitable and building software with more than one layer of security, meaning that even if attackers penetrate one layer of the system, they must also penetrate another layer to discover anything of value. Static analysis of code helps automate some of this today but is still not a guarantee of software security.

People tend not to take "security through obscurity" seriously, but there is some value to the technique. Address space layout randomization is an example of this approach. It comes at a performance cost, however.

This paper describes a new approach to slowing down attackers trying to exploit your system. Because this approach automatically injects nonexploitable bugs into software, an attacker who finds said bugs will waste precious time triaging the bug in order to use it maliciously and will fail. In some cases the bugs injected will cause the program to crash, but in modern distributed systems this is unlikely to be an issue because many programs use process pools, and high-availability systems, like those that use containers, typically have a policy for automatically restarting the program on crash.

The bugs injected come in two forms: those that overwrite unused data, and those that overwrite sensitive data with nonexploitable values. The former is fairly straightforward: inject unused variables into the code and ensure the dummy variable is placed directly adjacent

to the variable that will be overflowed. In the latter case of overwriting sensitive data, the attacker's input value is overconstrained, meaning it has a defined set of constraints that are by design forced eventually to be zero, through bitmasks and controlling the pathway that the data is passed through.

The key insight in this paper is that instead of trying to decrease the number of bugs in your program, you could increase them but make them nonexploitable, thereby deterring attackers by wasting their time. There is still a performance overhead brought on by the overconstrained checking of inputs, and it is an open question whether the attackers could find patterns in the injected bugs to rule them out automatically. This was, however, enough to fool tools such as gdb, which considered the bugs "exploitable" and "probably exploitable." Could future versions of this approach be designed differently to be more useful to open-source projects? Having the source code would surely give attackers an advantage in discovering which bugs were real and which were injected.

THE FUTURE OF SECURING APPLICATIONS IN A USABLE WAY

The container ecosystem is very fast paced. Numerous companies are building products on top of existing technologies, while enterprises are using these technologies and products to run their infrastructures. The focus of the three papers described here is on advancements to the underlying technologies themselves and strategic ways to secure software in the modern age.

The first paper rethinks VMs in modern environments

purely as mechanisms for running applications. This allows for the creation of minimal VMs that can behave just like containers in terms of memory overhead, density, and boot time. The second paper takes this a bit further by packaging the monitor in the unikernel. This is an extremely usable way to execute unikernels since the operator does not have to install a VM manager. It also allows for a more minimal monitor, limiting the attack surface. IBM's recently launched Nabla container runtime is an example of those approaches. Both papers leverage unikernels and have an open question as to whether unikernels can eventually be as easy to build as containers are today. This will be a hurdle for those implementations to overcome.

The third paper suggests a whole new approach which also gives operators a new set of controls for managing state. Through isolation at the address space and tying each piece of state to an IP address, operators gain clear controls over a program's code, stack, and heap. Alto not only innovated as far as isolation techniques but also in terms of operability and control.

This should push forward methods for easily debugging the applications running in minimal VMs. Until these applications can be debugged as easily as standard Linux containers, adoption by most practitioners will be slow, as the learning curve is higher.

Finally, isolation is not the only way to secure applications. The last paper could inspire others to devise new methods of automating ways to deter attackers.

Giving operators a usable means of securing the methods they use to deploy and run applications is a win for everyone. Keeping the usability-focused abstractions

provided by containers, while finding new ways to automate security and defend against attacks, is a great path forward.

Jessie Frazelle *works for Microsoft in the cloud organization. She was a maintainer of Docker and has been a core contributor to many different open-source projects in the container ecosystem and outside of it. She has a strong love of usable, uncomplex interfaces, performance, and security, specifically technologies around isolation.*