CHAPTER 1

Unikernels: A New Technology to Combat Current Problems

At the writing of this report, unikernels are the new kid on the cloud block. Unikernels promise small, secure, fast workloads, and people are beginning to see that this new technology could help launch a new phase in cloud computing.

To put it simply, unikernels apply the established techniques of embedded programming to the datacenter. Currently, we deploy applications using beefy general-purpose operating systems that consume substantial resources and provide a sizable attack surface. Unikernels eliminate nearly all the bulk, drastically reducing both the resource footprint and the attack surface. This could change the face of the cloud forever, as you will soon see.

What Are Unikernels?

For a functional definition of a unikernel, let's turn to the burgeoning hub of the unikernel community, Unikernel.org, which defines it as follows:

> Unikernels are specialised, single-address-space machine images constructed by using library operating systems.

In other words, unikernels are small, fast, secure virtual machines that lack operating systems.

I could go on to focus on the architecture of unikernels, but that would beg the key question: why? Why are unikernels *really* needed?

Why can't we simply live with our traditional workloads intact? The status quo for workload construction has remained the same for years; why change it now?

Let's take a good, hard look at the current problem. Once we have done that, the advantages of unikernels should become crystal clear.

The Problem: Our Fat, Insecure Clouds

When cloud computing burst on the scene, there were all sorts of promises made of a grand future. It was said that our compute farms would magically allocate resources to meet the needs of applications. Resources would be automatically optimized to do the maximum work possible with the assets available. And compute clouds would leverage assets both in the datacenter and on the Internet, transparently to the end user.

Given these goals, it is no surprise that the first decade of the cloud era focused primarily on how to do these "cloudy" things. Emphasis was placed on developing excellent cloud orchestration engines that could move applications with agility throughout the cloud. That was an entirely appropriate focus, as the datacenter in the time before the cloud was both immobile and slow to change. Many system administrators could walk blindfolded through the aisles of their equipment racks and point out what each machine did for what department, stating exactly what software was installed on each server. The placement of workloads on hardware was frequently laborious and static; changing those workloads was a slow, difficult, and arduous task, requiring much verification and testing before even the smallest changes were made on production systems.

The Old Mindset: Change Was Bad

In the era before clouds, there was no doubt in the minds of operations staff that *change was bad*. Static was good. When a customer needed to change something—say, upgrade an application—that change had to be installed, tested, verified, recorded, retested, reverified, documented, and finally deployed. By the time the change was ready for use, it became the new status quo. It became the new static reality that should not be changed without another monumental effort.

If an operations person left work in the evening and something *changed* during the night, it was frequently accompanied by a 3 AM phone call to come in and fix the issue before the workday began… or else! Someone needed to beat the change into submission until it ceased being a change. Change was unmistakably bad.

The advent of cloud orchestration software (OpenStack, CloudStack, openNebula, etc.) altered all that—and many of us were very grateful. The ability of these orchestration systems to adapt and change with business needs turned the IT world on its head. A new world ensued, and the promise of the cloud seemed to be fulfilled.

Security Is a Growing Problem

However, as the cloud era dawned, it became evident that a good orchestration engine alone is simply not enough to make a truly effective cloud. A quick review of industry headlines over the past few years yields report after report of security breaches in some of the most impressive organizations. Major retailers, credit card companies, even federal governments have reported successful attacks on their infrastructure, including possible loss of sensitive data. For example, in May 2016, the *Wall Street Journal* ran a story about banks in three different countries that had been recently hacked to the tune of $90 million in losses. A quick review of the graphic representation of major attacks in the past decade will take your breath away. Even the US Pentagon was reportedly hacked in the summer of 2011. It is no longer unusual to receive a letter in the mail stating that your credit card is being reissued because credit card data was compromised by malicious hackers.

I began working with clouds before the term “cloud” was part of the IT vernacular. People have been bucking at the notion of security in the cloud from the very beginning. It was the 800-pound gorilla in the room, while the room was still under construction!

People have tried to blame the cloud for data insecurity since day one. But one of the dirty little secrets of our industry is that our data was never as safe as we pretended it was. Historically, many organizations have simply looked the other way when data security was questioned, electing instead to wave their hands and exclaim, “We have an excellent firewall! We’re safe!” Of course, anyone who thinks critically for even a moment can see the fallacy of that concept. If firewalls were enough, there would be no need for antivirus programs or email scanners—both of which are staples of the PC era.

Smarter organizations have adopted a defense-in-depth concept, in which the firewall becomes one of several rings of security that surround the workload. This is definitely an improvement, but if nothing is done to properly secure the workload at the center of consideration, this approach is still critically flawed.

In truth, to hide a known weak system behind a firewall or even multiple security rings is to rely on *security by obscurity*. You are betting that the security fabric will keep the security flaws away from prying eyes well enough that no one will discover that data can be compromised with some clever hacking. It's a flawed theory that has always been hanging by a thread.

Well, in the cloud, *security by obscurity is dead!* In a world where a virtual machine can be behind an internal firewall one moment and out in an external cloud the next, you cannot rely on a lack of prying eyes to protect your data. If the workload in question has never been properly secured, you are tempting fate. We need to put away the dreams of firewall fairy dust and deal with the cold, hard fact that your data is at risk if it is not bolted down tight!

The Cloud Is Not Insecure; It Reveals That Our Workloads Were Always Insecure

The problem is not that the cloud introduces new levels of insecurity; it's that the data was never really secure in the first place. The cloud just made the problem visible—and, in doing so, escalated its priority so it is now critical.

The best solution is not to construct a new type of firewall in the cloud to mask the deficiencies of the workloads, but to change the workloads themselves. We need a new *type* of workload—one that raises the bar on security by design.

Today's Security is Tedious and Complicated, Leaving Many Points of Access

Think about the nature of security in the traditional software stack:

1. First, we lay down a software base of a complex, multipurpose, multiuser operating system.

2. Next, we add hundreds—or even thousands—of utilities that do everything from displaying a file's contents to emulating a hand-held calculator.
3. Then we layer on some number of complex applications that will provide services to our computing network.
4. Finally, someone comes to an administrator or security specialist and says, "Make sure this machine is secure before we deploy it."

Under those conditions, true security is unobtainable. If you applied every security patch available to each application, used the latest version of each utility, and used a hardened and tested operating system kernel, you would only have started the process of making the system secure. If you then added a robust and complex security system like SELINUX to prevent many common exploits, you would have moved the security ball forward again. Next comes testing—lots and lots of testing needs to be performed to make sure that everything is working correctly and that typical attack vectors are truly closed. And then comes formal analysis and modeling to make sure everything looks good.

But what about the atypical attack vectors? In 2015, the VENOM exploit in QEMU was documented. It arose from a bug in the virtual floppy handler within QEMU. The bug was present even if you had no intention of using a virtual floppy drive on your virtual machines. What made it worse was that both the Xen Project and KVM open source hypervisors rely on QEMU, so all these virtual machines—literally millions of VMs worldwide—were potentially at risk. It is such an obscure attack vector that even the most thorough testing regimen is likely to overlook this possibility, and when you are including thousands of programs in your software stack, the number of obscure attack vectors could be huge.

But you aren't done securing your workload yet. What about new bugs that appear in the kernel, the utilities, and the applications? All of these need to be kept up to date with the latest security patches. But does that make you secure? What about the bugs that haven't been found yet? How do you stop each of these? Systems like SELINUX help significantly, but it isn't a panacea. And who has certified that your SELINUX configuration is optimal? In practice, most SELINUX configurations I have seen are far from optimal by design, since the fear that an aggressive configuration will accidentally keep a legitimate process from succeeding is quite real in many people's minds. So many installations are put into production with less-than-optimal security tooling.

The security landscape today is based on a fill-in-defects concept. We load up thousands of pieces of software and try to plug the hundreds of security holes we've accumulated. *In most servers that go into production, the owner cannot even list every piece and version of software in place on the machine. So how can we possibly ensure that every potential security hole is accounted for and filled?* The answer is simple: *we can't!* All we can do is to do our best to correct everything we know about, and be diligent to identify and correct new flaws as they become known. But for a large number of servers, each containing thousands of discrete components, the task of updating, testing, and deploying each new patch is both daunting and exhausting. It is no small wonder that so many public websites are cracked, given today's security methodology.

And Then There's the Problem of Obesity

As if the problem of security in the cloud wasn't enough bad news, there's the problem of "fat" machine images that need lots of resources to perform their functions. We know that current software stacks have hundreds or thousands of pieces, frequently using gigabytes of both memory and disk space. They can take precious time to start up and shut down. Large and slow, these software stacks are virtual dinosaurs, relics from the stone age of computing.

Once Upon a Time, Dinosaurs Roamed the Earth

I am fortunate to have lived through several eras in the history of computing. Around 1980, I was student system administrator for my college's DEC PDP-11/34a, which ran the student computing center. In this time before the birth of IBM's first personal computer, there was precisely *one* computer allocated for all computer science, mathematics, and engineering students to use to complete class assignments. This massive beast (by today's standards; back then it was considered petite as far as computers were concerned) cost many tens of thousands of dollars and had to do the bidding of a couple hundred students each and every week, even though its modest capacity was multiple orders of magnitude below any recent smartphone. We ran the entire student computing center on just

248 KB of memory (no, that's not a typo) and 12.5 MB of total disk storage.

Back then, hardware was truly expensive. By the time you factored in the cost of all the disk drives and necessary cabinetry, the cost for the system must have been beyond $100,000 for a system that could not begin to compete with the compute power in the Roku box I bought on sale for $25 last Christmas. To make these monstrously expensive minicomputers cost-effective, it was necessary for them to perform every task imaginable. The machine had to authenticate hundreds of individual users. It had to be a development platform, a word processor, a communication device, and even a gaming device (when the teachers in charge weren't looking). It had to include every utility imaginable, have every compiler we could afford, and still have room for additional software as needed.

The recipe for constructing software stacks has remained almost unchanged since the time before the IBM PC when minicomputers and mainframes were the unquestioned rulers of the computing landscape. For more than 35 years, we have employed software stacks devised in a time when hardware was slow, big, and expensive. Why? We routinely take "old" PCs that are thousands of times more powerful than those long-ago computing systems and throw them into landfills. If the hardware has changed so much, why hasn't the software stack?

Using the old theory of software stack construction, we now have clouds filled with terabytes of unneeded disk space using gigabytes of memory to run the simplest of tasks. Because these are patterned after the systems of long ago, starting up all this software can be slow—much slower than the agile promise of clouds is supposed to deliver. So what's the solution?

Slow, Fat, Insecure Workloads Need to Give Way to Fast, Small, Secure Workloads

We need a new type of workload in the cloud. One that doesn't waste resources. One that starts and stops almost instantly. One that will reduce the attack surface of the machine so it is not so hard to make secure. A radical rethink is in order.

A Possible Solution Dawns: Dockerized Containers

Given this need, it is no surprise that when Dockerized containers made their debut, they instantly became wildly popular. Even though many people weren't explicitly looking for a new type of workload, they still recognized that this technology could make life easier in the cloud.

NOTE

For those readers who might not be intimately aware of the power of Dockerized containers, let me just say that they represent a major advance in workload deployment. With a few short commands, Docker can construct and deploy a canned lightweight container. These container images have a much smaller footprint than full virtual machine images, while enjoying snap-of-the-finger quick startup times.

There is little doubt that the combination of Docker and containers does make massive improvements in the right direction. That combination definitely makes the workload smaller and faster compared to traditional VMs.

Containers necessarily share a common operating system kernel with their host system. They also have the capability to share the utilities and software present on the host. This stands in stark contrast to a standard virtual (or hardware) machine solution, where each individual machine image contains separate copies of each piece of software needed. Eliminating the need for additional copies of the kernel and utilities in each container on a given host means that the disk space consumed by the containers on that host will be much smaller than a similar group of traditional VMs.

Containers also can leverage the support processes of the host system, so a container normally only runs the application that is of interest to the owner. A full VM normally has a significant number of processes running, which are launched during startup to provide services within the host. Containers can rely on the host's support processes, so less memory and CPU is consumed compared to a similar VM.

Also, since the kernel and support processes already exist on the host, startup of a container is generally quite quick. If you've ever watched a Linux machine boot (for example), you've probably noticed that the lion's share of boot time is spent starting the kernel and support processes. Using the host's kernel and existing processes makes container boot time extremely quick—basically that of the application's startup.

With these advances in size and speed, it's no wonder that so many people have embraced Dockerized containers as the future of the cloud. But the 800-pound gorilla is still in the room.

Containers are Smaller and Faster, but Security is Still an Issue

All these advances are tremendous, but the most pressing issue has yet to be addressed: security. With the number of significant data breaches growing weekly, increasing security is definitely a requirement across the industry. Unfortunately, containers do not raise the bar of security nearly enough. In fact, unless the administrator works to secure the container prior to deployment, he may find himself in a more vulnerable situation than when he was still using a virtual machine to deploy the service.

Now, the folks promoting Dockerized containers are well aware of that shortfall and are expending a large amount of effort to fix the issue—and that's terrific. However, the jury is still out on the results. We should be very mindful of the complexity of the lockdown technology. Remember that Dockerized containers became the industry darling precisely because of their ease of use. A security add-on that requires some thought—even a fairly modest amount—may not be enacted in production due to "lack of time."

NOTE

I remember when SELINUX started to be installed by default on certain Linux distributions. Some people believed this was the beginning of the end of insecure systems. It certainly seemed logical to think so—unless you observed what happened when people actually deployed those systems. I shudder to think how many times I've heard, "we need to get this server up now, so we'll shut off SELINUX and configure it later." Promising to "configure SELINUX when there's time" carries about as much weight as a politician's promise to secure world peace. Many great intentions are never realized for the perception of "lack of time."

Unless the security solution for containers is as simple as using Docker itself, it stands an excellent chance of dying from neglect. The solution needs to be easy and straightforward. If not, it may present the promise of security without actually delivering it in practice. Time will tell if container security will rise to the needed heights.

It Isn't Good Enough to Get Back to Yesterday's Security Levels; We Need to Set a Higher Bar

But the security issue doesn't stop with ease of use. As we have already discussed, we need to raise the level of security in the cloud. If the container security story doesn't *raise the security level of workloads by default*, we will still fall short of the needed goal.

We need a new cloud workload that provides a higher level of security without expending additional effort. We must stop the "come from behind" mentality that makes securing a system a critical afterthought. Instead, we need a new level of security "baked in" to the new technology—one that closes many of the existing attack vectors.

A Better Solution: Unikernels

Thankfully, there exists a new workload theory that provides the small footprint, fast startup, and improved security we need in the next-generation cloud. This technology is called *unikernels*. Unikernels represent a radically different theory of an enterprise software stack—one that promotes the qualities needed to create and radically improve the workloads in the cloud.

Smaller

First, unikernels are small—very small; many come in at *less than a megabyte* in size. By employing a truly minimalist concept for software stack creation, unikernels create actual VMs so tiny that the smallest VM allocations by external cloud providers are huge by comparison. A unikernel literally employs the functions needed to make the application work, and nothing more. We will see examples of these in the subsection "Let's Look at the Results" on page 21.

Faster

Next, unikernels are very quick to start. Because they are so tiny, devoid of the baggage found in a traditional VM stack, unikernels start up and shut down amazingly quickly—often measured in milliseconds. The subsection "Let's Look at the Results" on page 21 will discuss a few examples. In the "just in time" world of the cloud, a service that can be created when it is needed, and terminated when the job is done, opens new doors to cloud theory itself.

And the 800-Pound Gorilla: More Secure

And finally, unikernels substantially improve security. The attack surface of a unikernel machine image is quite small, lacking the utilities that are often exploited by malicious hackers. This security is built into the unikernel itself; it doesn't need to be added after the fact. We will explore this in "Embedded Concepts in a Datacenter Environment" on page 19. While unikernels don't achieve perfect security by default, they do raise the bar significantly without requiring additional labor.

CHAPTER 2

Understanding the Unikernel

Unikernel theory is actually quite easy to understand. Once you understand what a unikernel is and what it is designed to do, its advantages become readily apparent.

Theory Explained

Consider the structure of a "normal" application in memory (see Figure 2-1).

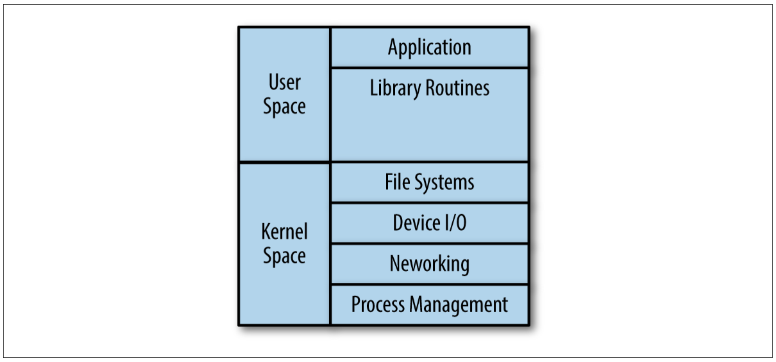

Figure 2-1. Normal application stack

The software can be broken down into two address spaces: the kernel space and the user space. The kernel space has the functions covered by the operating system and shared libraries. These include low-level functions like disk I/O, filesystem access, memory management, shared libraries, and more. It also provides process isolation, process scheduling, and other functions needed by multiuser operating systems. The user space, on the other hand, contains the application code. From the perspective of the end user, the user space contains the code you want to run, while the kernel space contains the code that needs to exist for the user space code to actually function. Or, to put it more simply, the user space is the interesting stuff, while the kernel space contains the other stuff needed to make that interesting stuff actually work.

The structure of a unikernel, however, is a little different (see Figure 2-2).

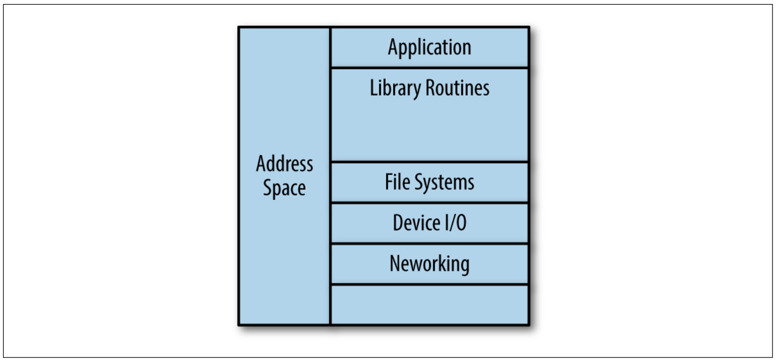

Figure 2-2. Unikernel application stack

Here we see something very similar to Figure 2-1, except for one critically different element: there is no division between user and kernel space. While this may appear to be a subtle difference, it is, in fact, quite the opposite. Where the former stack is a combination of a kernel, shared libraries, and an application to achieve its goal, the latter is one contiguous image. There is only one program running, and it contains everything from the highest-level application code to the lowest-level device I/O routine. It is a singular image that requires nothing to boot up and run except for itself.

At first this concept might sound backward, even irrational. "Who has time to code, debug, and test all these low-level functions for every program you need to create?" someone might ask. "I want to leverage the stable code contained in a trusted operating system, not recode the world every time I write a new program!" But the answer is simple: unikernels do at compile time what standard programs do at runtime.

In our traditional stacks, we load up an operating system designed to perform every possible low-level operation we can imagine and then load up a program that cherry-picks those operations it needs as it needs them. The result works well, but it is fat and slow, with a large potential attack surface. The unikernel raises the question, "Why wait until runtime to cherry-pick those low-level operations that an application needs? Why not introduce that at compile time and do away with everything the application doesn't need?"

So most unikernels (one notable exception is OSv, which will be discussed in Chapter 3) use a specialized compiling system that compiles in the low-level functions the developer has selected. The code for these low-level functions is compiled directly into the application executable through a *library operating system*—a special collection of libraries that provides needed operating system functions in a compilable format. The result is compiled output containing absolutely everything that the program needs to run. It requires no shared libraries and no operating system; it is a completely self-contained program environment that can be deposited into a blank virtual machine and booted up.

Bloat Is a Bigger Issue Than You Might Think

I have spoken about unikernels at many conferences and I sometimes hear the question, "What good does it do to compile in the operating system code to the application? By the time you compile in all the code you need, you will end up with almost as much bloat as you would in a traditional software stack!" This would be a valid assessment if an average application used most of the functions contained in an average operating system. In truth, however, an average application uses only a tiny fraction of capabilities on an average operating system.

Let's consider a basic example: a DNS server. The primary function of a DNS server is to receive a network packet requesting the translation of a particular domain name and to return a packet containing the appropriate IP address corresponding to that name. The DNS server clearly needs network packet transmit and receive routines. But does it need console access routines? No. Does it need advanced math libraries? No. Does it need SSL encryption routines? No. In fact, the number of application libraries on a standard server is many times larger than what a DNS server actually needs.

But the parade of unneeded routines doesn't stop there. Consider the functions normally performed by an operating system to support itself. Does the DNS server need virtual memory management? No. How about multiuser authentication? No. Multiple process support? Nope. And the list goes on.

The fact of the matter is that a working DNS server uses only a minuscule number of the functions provided by a modern operating system. The rest of the functions are unnecessary bloat and are not pulled into the unikernel during the compilation, creating a final image that is small and tight. How small? How about an image that is less than 200 KB in size?

But How Can You Develop and Debug Something Like This?

It's true that developing software under these circumstances might be tricky. But because the pioneers of unikernel technology are also established software engineers, they made sure that development and debugging of unikernels is a very reasonable process.

During the development phase (see Figure 2-3), the application is compiled as if it were to be deployed as software on a traditional stack. All of the functions normally associated with kernel functions are handled by the kernel of the development machine, as one would expect on a traditional software stack. This allows for the use of normal development tools during this phase. Debuggers, profilers, and associated tools can all be used as in a normal development process. Under these conditions, development is no more complex than it has ever been.

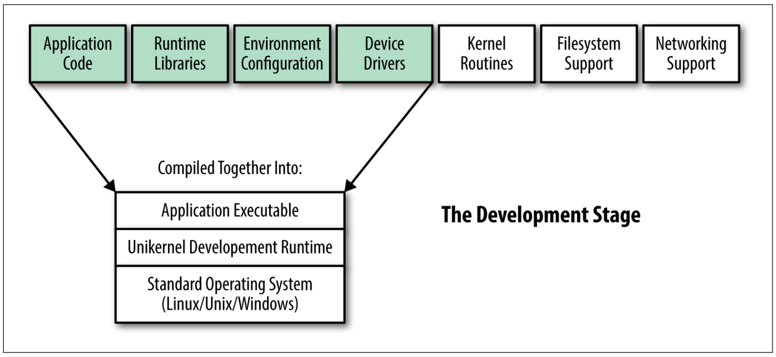

Figure 2-3. Unikernel development stack

During the testing phase, however, things change (see Figure 2-4). Now the compiler adds in the functions associated with kernel activity to the image. However, on some unikernel systems like MirageOS, the testing image is still deployed on a traditional host machine (the development machine is a likely choice at this stage). While testing, all the usual tools are available. The only difference is that the compiler brings in the user space library functions to the compiled image so testing can be done without relying on functions from the test operating system.

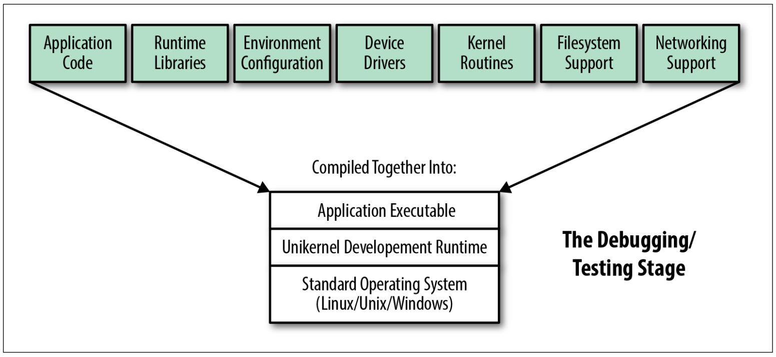

Figure 2-4. Unikernel testing stack

Finally, at the deployment phase (see Figure 2-5), the image is ready for deployment as a functional unikernel. It is ready to be booted up as a standalone virtual machine.

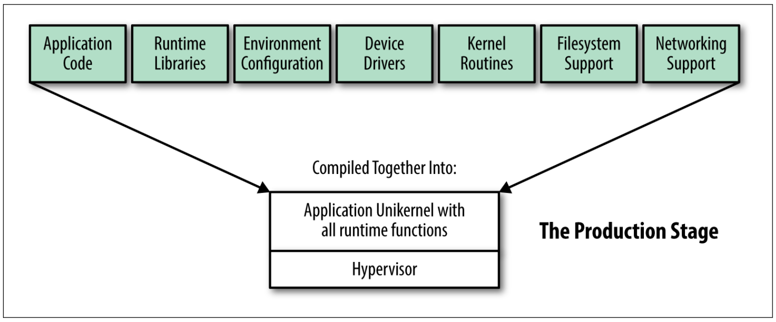

Figure 2-5. Unikernel deployment stack

Understanding the Security Picture

Consider the ramifications to security when deploying a unikernel in production. Many pieces that are routinely attacked or compromised by malicious hackers are absent:

- There is no command shell to leverage.
- There are no utilities to co-opt.
- There are no unused device drivers or unused libraries to attack.
- There are no password files or authorization information present.
- There are no connections to machines and databases not needed by the application.

Assume that a malefactor has discovered a flaw in the application code or, perhaps, the network device driver. She has discovered a way to break the running code. Now what? She cannot drop to a command line to begin an assault on the information she wishes to obtain. She cannot summon thousands of utilities to facilitate her end. She has to break the application in a clever enough way to return the desired information without any tools; that can be a task that is much more difficult than exploiting the original flaw. But what information is available to be won? There are no password files, no links to unused databases, no slew of attached storage devices like you frequently find on a full general-purpose operating system. Not only is the attack surface small and the number of assets to co-opt small, but the unikernel is very unlikely to be a target-rich environment—the desirable information available is extremely limited. On top of this, the ability to convert the unikernel into an attack platform for further malice is also extremely limited.

Around this security footprint, which is present by default, we can now optionally wrap a second layer like the Xen Security Modules (XSM) or similar security construct. XSM is very similar to SELINUX, except that it is designed to work in a virtual environment. Where SELINUX can be difficult to properly configure on a multiprocess operating system, XSM around a unikernel should be much easier to configure because you must consider only the needs of a single process. For example, if the application has no need to write to disk, go ahead and disable write access with XSM. Then, even an extremely clever malicious hacker will be unable to penetrate the unikernel and write something to disk.

Embedded Concepts in a Datacenter Environment

Despite the fact that the unikernel concept is very new for the cloud, it is not actually new for the software industry. In fact, this is virtually identical to the process used in embedded programming.

In embedded programming, the software is often developed in a traditional software development environment, allowing for the use of a variety of normal software development tools. But when the software is ready, it is cross-compiled into a standalone image, which is then loaded into the embedded device. This model has served the embedded software industry successfully for years. The approach is proven, but employing it in the enterprise environment and the cloud is new.

Despite the proven nature of this process in the embedded world, there are still claims that this puts an unacceptable limitation on debugging actual production systems in the enterprise. Since there is no operating system environment on a unikernel production VM, there are no tools, no debuggers, no shell access with which someone can probe a failure on a deployed program. Instead, all that can be done is to engineer the executable to log events and data so that the failure might be reconstructed on a development system, which still has access to all the debugging tools needed.

While I can sympathize with this concern, my personal experience leads me to believe it is somewhat of a red herring. In my career, I have been a developer, a product manager, and a technology evangelist (among other jobs), but the bulk of my 35 years in this industry has been spent as a software services consultant. I have spent over two decades working on-site with a wide range of clients, including many US civilian federal agencies and Fortune 100 customers. In all that time, I cannot recall a single time where a customer allowed debugging of a production system for any reason. It was always required that on system failure, data and logs were exported onto a development platform, and the failed system was placed back into service immediately. We had to analyze and reconstruct the failure on a development box, fix the code, test, and then redeploy into production.

Now I don't deny that there are some production systems that are made available for debugging, but my experience suggests that access to production systems for debugging during a failure is not at all as common as some people think. And in many large organizations, where the benefit of unikernels can be quite significant, the loss of production debugging is no loss at all. I and others in my role have dealt with this restriction for years; there is nothing new here. People in our industry have successfully debugged failures of complex software for years without direct access to production systems, and I see no reason why they will fail to do so now.

Trade-offs Required

Objections aside, the value received from the adoption of unikernels where they are appropriate is much greater than any feared cost. Our industry has become so infatuated with the notion of endless external clouds that we sometimes fail to realize that every VM has to reside in a datacenter somewhere. Every VM launched requires that there be a host machine in a rack consuming energy to run, and consuming yet more energy indirectly to be kept cool. Virtual machines require physical machines, and physical machines have limitations.

About a decade ago, I had a customer who had built a very large datacenter. He was still running power in the building when he said, "You know, this datacenter is full." I was puzzled at first; how could the building be full when he hadn't even run power to half of it? He explained, "The datacenter may not look full right now, but I know where every machine will go. When every machine is in place, there won't be room for even one more!" I asked him why—why build a datacenter that will be maxed out on the day it is fully operational? The answer was simple; he went to the local electric utility company, and they told him the maximum amount of power they could give him. He built his new datacenter to use exactly that much electricity. If he wanted to add more capacity, he'd have to build an entirely new datacenter at some other location on the electric grid. There simply was no more electricity available to him.

In the world of the cloud, we suffer under the illusion that we have an endless supply of computing resources available. That's the premise of the cloud—you have what you need, as much as you need, when you need it. And, while that promise can be validated by the experience of many users, the truth behind the curtain is quite different. In the era of cloud, data has swelled to previously unimaginable size. And the number of physical servers required to access and process requests for that data has become enormous. Our industry has never had as many huge datacenters worldwide as we have today, and the plans for additional datacenters keep piling up. Building additional datacenters in different locations due in part to power concerns is *extremely* expensive.

We cannot reduce the demand for information—that demand is, in fact, demonstrably growing year over year. So the prudent solution is to reduce the impact of the demand by shrinking the resource footprint for a large percentage of the workloads. We need to take a nontrivial portion of the workloads—including web servers, application servers, even smaller databases—and reduce their resource demand to a fraction of their traditional sizes. That way, we can actually fit significantly more processing demand into the same power footprint. And, if we can take some permanent services that sit idle portions of the time, yet require resources in their waiting state, and transform them into transient services that come and go with demand, we can reduce the demand even more. If I could have told my customer with the new datacenter that we had a plan to reclaim even 20 percent of his capacity, he would have jumped for joy so high we would have had to scrape him off the ceiling. Today, I believe we could reclaim much more capacity than that in many datacenters by using unikernels.

Let's Look at the Results

The amount of resources reclaimed can be enormous when evaluated on a per-service basis. Compared to the traditional fully loaded VM, unikernels can be extremely small. Let's consider a few examples.

The team behind MirageOS—one of the oldest unikernel projects—has built a number of basic services on their unikernel platform. One example is a functional DNS server that compiles to 184 KB in size.[1] Note that the size is measured in *kilobytes*, not *megabytes*. In a world where we routinely talk about gigabytes, terabytes, or petabytes, the very word "kilobytes" has become foreign to our ears. Many of us haven't spoken the word "kilobytes" since the last millennium! And yet we see the reality that a fundamental network service like DNS can be instantiated in an image that is *less than one-fifth of a megabyte in size!*

But that is just the beginning. MirageOS also has a web server that compiles in at a miniscule 674 KB. That's a working web server that weighs in at a little more than two-thirds of a megabyte. And they have an OpenFlow learning switch that comes in at a paltry 393 KB —less than four-tenths of a megabyte. You could store two of these in a single megabyte of disk space and still have room left over.

But MirageOS is not unique in its tiny footprints. The team behind the LING unikernel runs a website based on their project. Their site provides a special button in the upper-right corner of the home page. By clicking the button, the user can see all sorts of statistics about the running instance. One of these metrics is a diagram labeled "Memory Usage." At the time of this writing, the instance I was connected to was using a total of 18.6 MB. A small server with only 4 GB of memory could potentially host more than 200 of these website VMs; just imagine what a large server could do.

Then there is the ClickOS project. These people have done extensive work building and benchmarking a variety of network function virtualization (NFV) devices. The results aren't shabby: consider the impact of a network device that starts up in about 20 milliseconds, occupies less than 6 MB of memory, and can successfully process over 5 million packets per second. Suddenly, the rules have changed.

These are but a few examples of essential cloud services that have been transformed into ultra-small unikernel instances. It's easy to see that if we can implement many of the normal services that populate our clouds and datacenters as unikernels, we stand to reclaim a considerable amount of underutilized resources.

1 If you attended any of my previous talks about unikernels, you may have heard me cite the size of the DNS image at 449 KB. I have been informed by a member of the MirageOS team that the current size is just 184 KB. This is one time I am glad to be wrong!

CHAPTER 3

Existing Unikernel Projects

There are a number of existing unikernel systems currently in the wild. But it's important to note that new ones appear all the time, so if none of these strike your fancy, a quick search of the Web might yield some interesting new ones. The following information is accurate at the time of this writing (summer of 2016), but the list of players and quality of the efforts could be markedly different in just a few months.

MirageOS

One of the oldest and most established unikernel efforts, the MirageOS project continues to be a spearhead for thought leadership in the unikernel world. Several MirageOS contributors have authored academic white papers and blogs that have helped propel unikernels into the popular consciousness. In addition, many of the generally accepted concepts around unikernels have been generated from members of this team. Many key people from this project (as well as some other unikernel engineers) went on to form Unikernel Systems, a company that was purchased by Docker at the beginning of 2016. So Docker now employs a significant portion of the MirageOS brain trust.

The MirageOS project is built on the OCaml language. If you are not familiar with OCaml, OCaml.org defines it as "an industrial strength programming language supporting functional, imperative and object-oriented styles." The relative obscurity of the OCaml language combined with the academic background of the principal contributors led many people to initially dismiss MirageOS as a mere university project. However, the spread of unikernel technology into other languages and commercial organizations has garnered MirageOS the respect it is due.

MirageOS allows for the construction of unikernels that can be deployed on the Xen Project hypervisor. MirageOS is both stable and usable, having reached release 2.0 in 2014. The project website has copious documentation and a few samples and tutorials to help the novice start to build unikernels. It also contains a number of white papers and presentations that continue to influence unikernel efforts across the world.

HaLVM

HaLVM is another of the earlier unikernel efforts, begun in 2006 and released in 2008 by Galois, Inc. The HaLVM is built on the Haskell programming language, and allows Haskell programmers to develop unikernels in much the same way as they develop normal Haskell programs, with some limitations.

The initial goal for the HaLVM was to ease the process of experimenting with the design and testing of core operating system components, but it quickly expanded to serve as a general-purpose platform for developing unikernels. Galois's first major product using the HaLVM was a multi-unikernel, collaborative IPSec implementation. More recently the HaLVM has been used as the base of a network defense product from FormalTech, Inc. (CyberChaff), and as the basis for Tor nodes.

Currently, the HaLVM—now at version 2.1—supports the Xen Project hypervisor, including both standard server and desktop deployments as well as Amazon's EC2. Future versions in the 3.0 series are slated to support a wider variety of systems, including KVM and bare-metal deployments.

LING

LING is yet another mature unikernel effort. It is a product of the Erlang-on-Xen project, written (appropriately) in Erlang for deployment (also appropriately) on the Xen Project hypervisor. The project website contains a number of interesting artifacts, including the ability to look under the covers of the unikernel that is running the website itself. If you press the Escape key, or click on the upper-right corner of the home page, you can see the resources being consumed by the website, which typically runs in only about 18 MB of memory.

The website also has a number of interesting links, including a list of use cases and a couple of working demos. The demos include Zerg and self-learning Go-Moku implementation. The Zerg demo is especially interesting, since it demonstrates a use case called the “Zero Footprint Cloud.” This use case talks about having unikernel-based VMs appear the instant they are needed and disappear as soon as their job is done. This concept is the basis of transient microservices, which will be discussed in Chapter 6.

ClickOS

The systems discussed so far have had a fairly wide focus. ClickOS, on the other hand, has a tight focus on the creation of NFV devices. A project from NEC’s European labs, ClickOS is an implementation of the Click modular router wed to a minimalistic set of operating system functions derived from MiniOS, which will be discussed under Chapter 4. The result is a unikernel-based router that boots in under 30 milliseconds, runs in only 6 MB of memory, and can process 10 Gbits per second in network packets. That’s a fast, small, and powerful virtual network device that can be readily deployed on the Xen Project hypervisor.

Rumprun

While most of the other unikernel systems reviewed so far rely on somewhat obscure languages, Rumprun is a very different animal. Rumprun is an implementation of a Rump kernel (which will be discussed under Chapter 4) and it can be used to transform just about any POSIX-compliant program into a working unikernel. Through Rumprun, it is theoretically possible to compile most of the programs found on a Linux or Unix-like operating system as unikernels (whether that is worthwhile is another issue; just because it is possible to render a program as a unikernel doesn’t mean that it is necessarily useful to do so). That represents a huge advance in unikernel capabilities.

Rumprun came to my attention in the spring of 2015 when it was announced that Martin Lucina had created the “RAMP” stack. Those of us in the open source world are well familiar with the LAMP stack: Linux, Apache, MySQL, and PHP (or Python or Perl). Martin ported NGINX (which is a bit faster and lighter than Apache), MySQL, and PHP to unikernels using Rumprun, thus creating the RAMP stack. Suddenly, one of the most common workloads in the world can be instantiated as a series of unikernels—and that’s just the beginning. Rumprun has the potential to bring the power of unikernels to workloads already found in datacenters worldwide.

OSv

OSv was originally created by a company called Cloudius Systems as a means of converting almost any application into a functioning unikernel. It is unique among existing unikernels in that it provides a general-purpose unikernel base that can accept just about any program that can run in a single process (multiple threads are allowed, but multiple processes are not). As a result, OSv is a “fat” unikernel; the results are measured in megabytes, rather than kilobytes. It is also more versatile than most, supporting a number of popular hypervisors, including KVM, Xen Project, Virtualbox, and VMware.

Cloudius Systems originally targeted OSv as an engine to run Java virtual machines by simply dropping a WAR file into it. However, its architecture can run just about any single-process solution, so many languages can be used, such as C, C++, Ruby, Perl, and many more.

Cloudius Systems has since been transformed into a new company called ScyllaDB. They are no longer building a business plan centered on servicing OSv, but the beauty of open source is that even though the company has changed direction, the software lives on. The OSv community is now maintaining the software, which is fully operational. It is also being used to power the Mikelangelo project, a European effort to create a working next-generation cloud framework.

A list of unikernel instances maintained by the OSv team can be found on their website. Other unikernel instances, not maintained by the OSv team, can be found in various locations on the Internet.

IncludeOS

Still in the prototype stage (release 0.8 occurred in June 2016), IncludeOS is an attempt to be able to run C++ code directly on a hypervisor. It was born from a university research project in Norway. Unlike most other unikernels, the hypervisor target is not the Xen Project hypervisor but KVM. IncludeOS also works on VirtualBox and Bochs.

And Much More in Development

These are only a few of the unikernels already in existence. There are other efforts, like Clive written in the Go language, Runtime.js for JavaScript, and even Drawbridge from Microsoft. Every couple of months, I see signs of new unikernels appearing in the wild. The list keeps growing, and even the unikernel list over at Unikernel.org has to struggle to stay up-to-date.

It is fascinating to realize that groups embracing this technology range from university research projects to small consultancies to the most giant companies in the industry. This last group is especially impressive. Even though unikernels are just beginning to break into the IT mindset, several major companies have already invested significant time in their development. IBM, NEC, Microsoft, EMC, Ericsson, and Docker have already made notable contributions to unikernel technology. It is rare to see so many large, existing organizations embracing a fledgling technology so early.

CHAPTER 4

Ecosystem Elements

A group of individual unikernel projects is interesting, but if there is no ecosystem developing around them, the advancement of this technology will slow to a crawl. However, that is not the case here. There are, in fact, a number of ecosystem projects supporting the development and use of unikernels. The following are only a handful of the most interesting ecosystem projects.

Jitsu

Jitsu demonstrates the promise of the amazing agility of unikernel-based workloads. Jitsu, which stands for "Just-in-Time Summoning of Unikernels," is actually a DNS server. But unlike other DNS servers, it responds to the DNS lookup request while simultaneously launching the unikernel that will service that address. Because unikernels can boot in milliseconds, it is possible to wait until someone has need for a service before that service is actually started. In this case, someone asking for the IP address of a service actually generates the service itself. By the time the requester sees the response to their DNS query, Jitsu has created the service that is associated with that IP address.

This ability to generate services as quickly as they are needed is a major game changer in our industry. We will see more of this in "Transient Microservices in the Cloud" on page 39.

MiniOS

MiniOS is a unikernel base from the Xen Project. Originally designed to facilitate driver disaggregation (basically, unikernel VMs that contain only a hardware driver for use by the hypervisor), MiniOS has been used as the base for any number of unikernel projects, including MirageOS and ClickOS. By itself, MiniOS does nothing. Its value is that, as open source software, it can be readily modified to enable unikernel projects. It leverages the functionality of the Xen Project hypervisor to simplify the task of unikernel development (refer to the subsection "Xen Project Hypervisor" on page 31 for additional information).

Rump Kernels

The Rump Kernel project has facilitated some of the most interesting advances in unikernel development. The concept of Rump Kernels comes from the world of NetBSD. Unlike most operating systems, NetBSD was specifically designed to be ported to as many hardware platforms as possible. Thus, its architecture was always intended to be highly modular, so drivers could be easily exchanged and recombined to meet the needs of any target platform. The Rump Kernel project provides the modular drivers from NetBSD in a form that can be used to construct lightweight, special-purpose virtual machines. It is the basis for Rumprun (see Chapter 3), a unikernel that can be used to power a wide range of POSIX-like workloads. The RAMP stack was created *without changes to the application code*. The bulk of the work was in modifying the compilation configuration so the unikernels could be produced.

Why would the compilation configuration be an issue? Keep in mind that the process of creating a unikernel is a cross-compilation. The target system is not the same as the development system. The development system is a fully functional multiuser operating system, while the production target is a standalone image that will occupy a virtual machine without any operating environment. That requires a cross-compile. And cross-compilation requires that the build process make the right choices to create usable output. So the source code may remain unaltered, but the compilation logic requires some work in some cases.

Xen Project Hypervisor

The Xen Project hypervisor was the first enterprise-ready open source hypervisor. Created in 2003, the Xen Project created a concept called *paravirtualization*, which has been heavily leveraged by most unikernel efforts to date. Most hypervisors use hardware virtualization—that is, the guest VM sees emulated hardware that looks identical to real hardware. The VM cannot tell that the hardware devices it sees are virtualized, so it employs the same drivers to run the devices that it would on an actual hardware-based server. That makes it easy for the operating system on the guest VM to use the hardware, but it isn't very efficient.

Consider an emulated network device. The guest operating system on the virtual machine sees a piece of networking hardware (let's say an NE2000). So it uses its NE2000 software driver to package up the network data to be acceptable to the hardware and sends it to the device. But the device is emulated, so the hypervisor needs to unpack the network data that the guest VM just packaged and then repack it in a method suitable for transport over whatever actual network device is available on the host hypervisor. That's a lot of unnecessary packing and unpacking. And, in a unikernel, that's a lot of unneeded code that we'd like to remove.

Xen Project is capable of providing hardware virtualization like any other hypervisor, but it also provides paravirtualization. Paravirtualization starts with the concept that some guest VMs may be smart enough to know that they are running in a hypervisor and not directly on server hardware. In that case, there is no need for fancy drivers and needless packing and unpacking of data. Instead, Xen Project provides a very simple paravirtualized interface for sending and receiving data to and from the virtualized device. Because the interface is simple, it replaces complex drivers with very lightweight drivers—which is ideal for a unikernel that wants to minimize unnecessary code.

This is one reason why the Xen Project has been at the forefront of unikernel development. Its paravirtualization capabilities allow unikernels to have a very small and efficient footprint interfacing with devices. Another reason is that the project team has helped foster unikernel innovation. The MirageOS project is in the Xen Project incubator, so that team has had significant influence on the direction of the hypervisor. As a result, the hypervisor team has been consciously reworking the hypervisor's capabilities so it can handle a future state where 2,000 or 3,000 simultaneous unikernel VMs may need to coexist on a single hardware host server. Currently, the hypervisor can handle about 1,000 unikernels simultaneously before scaling becomes nonlinear. The development work continues to improve unikernel support in each release.

Solo5

Solo5 is a unikernel base project, originating from the development labs at IBM. Like MiniOS, Solo5 is meant to be an interface platform between a unikernel and the hypervisor. Unlike MiniOS, the target hypervisor is KVM/QEMU rather than Xen Project. Where Xen Project leverages paravirtualization to allow the unikernel to talk to the hypervisor, Solo5 contains a hardware abstraction layer to enable the hardware virtualization used by its target hypervisors.

UniK

UniK (pronounced "unique") is a very recent addition to the unikernel ecosystem, with initial public release announced in May 2016. It is an open source tool written in Go for compiling applications into unikernels and deploying those unikernels across a variety of cloud providers, embedded devices (for IoT), as well as developer laptops or workstations. UniK utilizes a simple Docker-like command-line interface, making developing on unikernels as easy as developing on containers.

UniK utilizes a REST API to allow painless integration with orchestration tools, including example integrations with Docker, Kubernetes, and Cloud Foundry. It offers an architecture designed for a high degree of pluggability and scalability, with a wide range of support in a variety of languages, hardware architectures, and hypervisors. Although quite new, this is a project worth watching.

And Much More…

This is far from a definitive list of unikernel ecosystem elements. The reality is that the era of unikernels has just begun, so the development and refinement of new unikernel ecosystem elements is still in its infancy. There is still a large amount of work to be done to properly control unikernels in popular cloud orchestration systems (like OpenStack). Outside of the cloud, plenty of opportunity exists for projects that will deal with unikernel management. For example, there have been demonstrations of Docker controlling unikernels, which could become part of Docker's supported capabilities before long. And Jitsu makes sense for certain workloads, but how can unikernels be dynamically launched when a DNS server is not the best solution? We can expect that additional solutions will emerge over time.

It is important to understand that unikernels and their surrounding ecosystem are propelled by open source. While it is technically possible to create closed source unikernels, the availability of a wide variety of open source drivers and interfaces makes creation of unikernels much simpler. The best illustration of that is the Rump Kernel project, which heavily leverages existing mature NetBSD drivers, which themselves sometimes draw on original BSD code from decades ago. By using established open source libraries, Rump Kernels specifically—and other unikernels in general—can spend far less time on the drudgery of making drivers work and spend more time doing innovative unikernel tasks.

CHAPTER 5

Limits of the Solution

All technologies have their limits, and unikernels are no different. This section will discuss some of the things to keep in mind when you're considering a unikernel solution.

Unikernels Are Not a Panacea

For all their advantages, unikernels are not a panacea. In a post-unikernel-enabled cloud, there will be many non-unikernel workloads. Undoubtedly, there will be complex stacks that simply won't lend themselves to implementation as a unikernel. But that's fine—by turning some workloads into unikernels, we now have all the more resources to give to the complex old-school stacks. When we reduce the footprint of unikernel-capable workloads, we make plenty of room for beefier tasks.

Practical Limitations Exist

The keep-it-simple concept that enables unikernels necessarily comes at a price. Not every solution will be suitable for implementation as a unikernel. Others may need some architectural modification to fit within the unikernel concept. And still others will work with no code modifications whatsoever.

So what are the key limitations in unikernel implementations?

Single Process (but Multiple Threads)

For the simple unikernel stack to work, there is no room for the complexity of multiple process handling. Once you have multiple processes, the overhead rises dramatically.

Multiple processes require process management. There has to be a way to start a process, stop a process, inspect the status of a process, kill a misbehaving process, and so forth.

And all of these capabilities are just the tip of the iceberg when you need to support multiple processes. It's not difficult to understand why single processes are needed to make the ultra-light images that characterize unikernels.

Despite the lack of multiple processes, however, multiple threads are often supported by unikernel architectures. Threads are much lighter weight, requiring only a fraction of the overhead needed for multiple processes. If the workload can exist as a single process with one or more threads, it is a candidate for a unikernel.

Systems such as Rumprun and OSv allow for many existing workloads to make the leap into the unikernel world as long as they don't fork new processes. But what if your current application forks new processes? Or what if it currently employs multiple processes that rely on interprocess communication? Is there any way to make them into unikernels?

The answer to that last question is "With modifications, maybe." If your multiple process code can be modified to use multiple threads, your application could still be a unikernel candidate. Likewise, if your application that relies on interprocess communication can be modified to use intermachine communication, you could possibly load the distinct processes into separate unikernels and allow them to speak to each other across machines. As with all design decisions, someone intimately knowledgeable in the solution architecture will need to decide if it's worth employing a change in architecture to remake the solution as a unikernel.

Single User

Unikernels are fiercely single user. Multiple users require significant overhead. When you have different users, you must have authentication logic to verify the user's identity. And you need user-based privileges to say what that user can do. And you need file protections to determine what that user can touch. Plus you had better include a security system to record authentication failures.

But why would a single-process virtual machine need separate users anyway? The machine image will run the program it is intended to run and access the files it is designed to access. It doesn't need to log in. It doesn't need to ask, "Who am I?"

If your workload is like a web server or application server, it always starts with a certain user personality and there is no need for it to be concerned with mulitple user identities. If what you really want is a multiuser timesharing system, don't try to make it into a unikernel.

Limited Debugging

As previously covered, unikernels have very limited debugging capabilities in their production form. It is easiest to debug failures of unikernels by reproducing the failure on a development platform, where the application exists as a standard application and debugging tools abound. So in the architecture of the production unikernel, it is important to include enough logging so that a failure can be properly reconstructed in the development environment.

As I've said, in my experience, debugging production systems is rarely permitted in the real world, so loss of debugging instrumentation is more like the loss of an illusion than any real functionality. If someone claims that a certain failure can *only* be debugged on a live production system, they have a problem already—very few enterprise environments will permit such work, with or without unikernels.

Impoverished Library Ecosystem

Due to the recent nature of unikernels, the list of callable functions in most library operating systems is still only a subset of those available in a fully mature operating system like Linux. If a given application requires the use of some library function that has yet to be implemented by the targeted library operating system, it will be incumbent on the developers to implement that function (and, hopefully, release it to the unikernel project for future support and maintenance). If an application has a lot of these unimplemented functions, a developer may decide either to provide these functions, or to wait until such time as more of the needed libraries are available.

What Makes for a Good Unikernel Application?

The question of what makes a good unikernel application is a lot easier to answer than the question of what *doesn't* make a good unikernel application. We just covered a number of the limitations of the architecture, so anything that doesn't violate one of these limits remains a candidate for compiling as a unikernel. These conditions include:

- Does not need multiple processes on a single machine
- Can work as single user
- Can be instrumented internally to contain whatever diagnostics may be needed for debugging

In addition, there are some requirements that might actually suggest that a unikernel would be appropriate:

- Something that needs subsecond startup time
- Anything that might make sense as a transient microservice (which we will explore shortly in "Transient Microservices in the Cloud" on page 39)
- Something that will be exposed to the Internet (or has been violated in the past), and therefore needs the highest levels of security
- Something that may need to scale into very high numbers

But what programs should *not* be unikernels? That's actually a very tough question. Aside from the architectural limitations we've already discussed, there isn't a really good set of criteria to exclude something from being built as a unikernel. This is, in part, due to the newness of the technology; as people try building unikernels, we will undoubtedly gain a better feel for applications that aren't the best fit. Applications with lots of external dependencies may not be the best candidates, but time will tell. Also, unikernels have already proven to be surprising. When I first started speaking about unikernels, I used to make the statement that databases were probably not great candidates—and then Martin Lucina ported MySQL to a Rumprun unikernel and that assumption went out the window. So the question of what should not be made into a unikernel is still open.

CHAPTER 6

What's Ahead?

It's important to remember that this is only the beginning of the journey. Unikernels are not the destination; they are, I believe, the path to a new future in the cloud. What can we expect along that path? We won't know for sure until we get there, but here are a few ideas.

Transient Microservices in the Cloud

The potential impact of unikernels extends well beyond areas like resource utilization and system security. The arrival of the unikernel is poised to facilitate a radical reassessment of software architecture at the highest levels. With unikernels, one of the fundamental assumptions of most solution architectures is no longer valid: we cannot assume that all services in our architectures are persistent.

As we discussed earlier, machines were once expensive, large, and slow. This meant that each machine was loaded with thousands of software programs to meet the diverse needs of a large number of users. As a result, these expensive machines needed to be *persistent*—they had to be on any time anyone might possibly need any of the thousands of programs on the machine. So, by default, most machines were powered on and running 24 hours per day, 365 days per year, for the entire service life of the machine. People could assume that the needed software was always ready and waiting to process, even if it meant that the machine would sit idle for a significant part of its life.

But now, as we have seen, the rules have changed. Hardware is cheap, compact, and fast. There is no need to make virtual machine instances multifunction. And there is no reason to assume that any one VM is persistent; we can simply summon a new unikernel VM instance to handle a request that comes over the network whenever it appears.

This is a particularly crucial concept for proper implementation of the *Internet of Things* (IoT). In a world where every button you press, every knob you turn, and every switch you flip will cause an event to be launched into the network, nonpersistent services make perfect sense. Why have response agents sitting idle for hours waiting for something to happen, when they could just be generated with the event itself? The Internet of Things makes the most sense when you can support *transient* microservices, which come and go with each event that is generated.

But the nature of transient microservices is a challenge to our current implementations of the cloud. Right now, most cloud orchestration systems assume that all workloads are essentially persistent. In most cloud engines, the orchestration layer is responsible for the creation of a service. After waiting some time for the service to start, the orchestration layer then queries the new service to find out if it is ready to process requests. Then it periodically pings the service to make sure it is alive, well, and has adequate capacity. And, finally, it will tell the service when it is time to shut down.

That model, however, goes out the window in the world of transient microservices. As a transient microservice is actually generated from the request on the network for the microservice (see the subsection "Jitsu" on page 29 for an example), there is no time to waste waiting for some slow orchestration engine to process the request and generate the service. The transient microservice needs to appear in milliseconds in order to prevent delays in responding to the pending request. There is no need for the orchestration layer to ask the service if it is ready to work. And, since the service is self-terminating, there is no longer a need for the orchestration layer to check wellness, verify capacity, or issue a shutdown order. Depending on the nature of the service, it may have a total lifespan measured in tenths of a second.

This may seem very liberating at first, but it raises the question, how do we know that the request was successfully handled? Should the service somehow record the handling of the request within the orchestration layer? Or record it in a database? How should errors be handled? Within the service? Within the orchestration layer? Within the calling program? Whose responsibility is it to correct the problem?

These are actually not particularly difficult problems, but since we have not traveled this road in the cloud before, best practices don't yet exist. The industry will need to examine these issues, think them through, and bake them into a new type of cloud that supports both persistent and transient services. It's not overly hard, but it is quite different. Cloud orchestration systems will need to create a new architecture to handle these transient workloads. And now is the time for orchestration systems to create those designs so they are ready when the transient workloads arrive—because they will.

A Possible Fusion Between Containers and Unikernels

It seems every time I give a talk about unikernels these days, there are always a couple of people sold on containers who look at me incredulously. They seem to believe that I am proposing the destruction of their favorite technology—but that couldn't be further from the truth.

I don't believe that unikernels will cause the death of containers. Quite the opposite: I think unikernels will likely cause container technology to make serious leaps forward in the near future. The reason for this is the open source nature of both the technologies involved.

If we were talking about two closed source alternatives, we would likely see a death match ensue. The two products would battle it out and the best marketing department would probably win (if you think the best *technology* would win, I humbly suggest a closer inspection of the history of our industry is in order; OS/2 Warp is a good example).

But given that we are talking about competing open source technologies, the outcome is likely entirely different. When open source technologies collide, someone normally has the bright idea of combining the best elements of both into a new solution. Since the code is open, cross-pollination is the order of the day. I fully expect that in five years, the “unikernel versus container” debate will be ancient history, and the two technologies will produce something that will leverage the strengths of each.

We are already seeing early signs of a new synthesis in the Docker community. At DockerCon EU 2015 in Barcelona, members of the MirageOS team took the stage to demonstrate Dockerized unikernels. This could have been dismissed as a one-off skunkworks hack, but just a few months after that demonstration, Docker announced the purchase of Unikernel Systems, a fledgling company consisting of many of the most notable unikernel engineers on the radar. It’s clear that the fusion of Docker, containers, and unikernels has just begun. The time to get on board the train is right now.

In another advance, 2015 saw the emergence of Hyper_, an open source project from HyperHQ. These folks figured out how to run a container image directly on a hypervisor, thus reducing the resulting attack surface by removing the container host system entirely. As these and other efforts continue to provide real innovation, I expect that the future of the cloud will have a fusion of the best ideas. Whether the resulting code comes from the container camp, the unikernel camp, or somewhere else is yet to be determined. But I believe the unikernel concept will be alive and well in the next-generation cloud.

This Is Not the End of the Road; It’s Only the Beginning

As I have already said, this is just the beginning of something new—something desperately needed in our industry. The status quo of slow Internet-accessible workloads laden with unnecessary software requiring huge amounts of resources providing a gigantic attack surface is dead. Old-style workloads may persist deep in the bowels of the compute center if they must, but the notion of an Internet populated with full stacks waiting to be exploited is dead. But, like so many apocalyptic zombie stories that have inundated popular media in recent years, it still plods on lifelessly, creating havoc wherever it goes. We need to bury it already and work on building its successor.

Get ahead of the curve and begin developing unikernel awareness now, because, in my not-always-humble opinion, it will be part of your future very soon.

For further information, consider consulting the web pages of the individual unikernel projects listed in Chapter 4. You can also check out the resources at Unikernel.org, which seeks to be the informal hub of the unikernel world. In particular, I suggest the following documents:

- *Unikernels: Rise of the Virtual Operating System*, by Anil Madhavapeddy and David J. Scott; published in *ACM* in 2014
- *The Rise and Fall of the Operating System*, by Antti Kantee; published in *USENIX Login* in October 2015
- *Unikernels, Meet Docker!*, video demonstration by Anil Madhavapeddy; published on YouTube in November 2015
- *The Design and Implementation of the Anykernel and Rump Kernels* by Antti Kantee; work in progress
- *Next Generation Cloud: Rise of the Unikernel*, presented by Russell Pavlicek at Southeast Linux Fest; published on YouTube in May 2015