EPISODE 869

[INTRODUCTION]

[00:00:00] JM: Software applications running within a host operating system need to be isolated. Isolation prevents security vulnerabilities, such as one application accessing the memory of another. In modern cloud environments, a single physical host might be running multiple virtual machines on top of a hypervisor. Those virtual machines might be divided up into containers. The different virtual machines and containers might be operated by different users, or even different companies.

GVisor is a container sandbox runtime open sourced by Google. GVisor runs containers in a new user space kernel and provides a security system with low overhead. GVisor improves on the previous security properties of containers and multitenancy.

Michael Pratt and Yoshi Tamura work on gVisor at Google and they join the show to talk through the purpose of gVisor and the engineering around the project.

[SPONSOR MESSAGE]

[00:01:08] JM: This episode of Software Engineeringaily is sponsored by Datadog. Datadog integrates seamlessly with more than 200 technologies like Docker and Kubernetes so you can monitor your entire container cluster in real-time. See across all of your servers, containers, apps and services in one place with powerful visualizations, sophisticated alerting, distributed tracing and APM. Now, Datadog has application performance monitoring for Java.

Start monitoring your microservices today with a free trial. As a bonus, Datadog will send you a free t-shirt. You can get both of things by going to softwareengineeringdaily.com/datadog. That's softwareengineeringdaily.com/datadog.

Thank you, Datadog.

[INTERVIEW]

[00:02:02] JM: Michael Pratt and Yoshi Tamura, welcome to Software Engineering Daily.

[00:02:06] MP: Thank you.

[00:02:07] YT: Thank you for having us again.

[00:02:09] JM: Yes. We'll be talking about gVisor and container isolation today. First, I want to talk in the abstract about isolation. What does it mean for software to be isolated?

[00:02:23] YT: Yeah. I mean, I think isolation is a sort of very wide-ranging topic. I think in the context of container sandboxing and these sort of areas that gVisor is in was specifically talking about preventing sort of unwanted interactions between different services on the same physical machine. So if you're running two completely separate containers, they shouldn't be able to access each other's data or memory. They shouldn't be able to crash one another, things like that. There're sort of other forms of isolation beyond that when you're talking about do network services perform [inaudible 00:03:00] on each other. But that's kind of out of scope for this.

[00:03:04] MP: I would also add, we're talking about Sandbox in general with a modern container. But if you look back, the [inaudible 00:03:11] computer, it was always about, "Okay. These are expensive. How can we share it?" Once you wanted to share such kind of a precious resource, then you need to have some sort of isolation and you can think about that's kind of a history to operating system in general.

[00:03:27] JM: Okay. What are the benefits of isolating our workloads?

[00:03:32] MP: Yeah. So, when we're talking about isolating workloads, it's primarily – I guess the key benefit we're talking about is when we have different untrusted workloads, right? So, if we have two workloads from two completely different users and we're sharing the same machine for this resource efficiency perspectives, we don't want them to be able to access each other's data. One customer shouldn't be able to steal another customer's information.

[00:03:57] YT: [inaudible 00:03:58] to add on what Michael just said, it's all about as if multiple users are occupying that particular resource. From operator perspective or even from a user perspective, you don't necessarily want to pay the entire cost of that such resource, but you don't want to compromise the sort of usability or kind of the accessibility to those resource.

So, the rule of this isolation or the benefit of the isolation is that, yeah, you're going to share underneath it, but you're not going to realize it until something goes wrong.

[00:04:33] JM: Let's talk a little bit more about the risks of workloads that are not properly isolated. There's a term called breaking out. Could you define what that means and how that poses a risk to an application environment?

[00:04:51] YT: Can we assume that when you say breakout, it's more specific about the container breakout, or especially the container, like a modern container, like Docker and [inaudible 00:04:59] container.

[00:05:01] JM: Sure. Sure, that's fine.

[00:05:02] YT: Great.

[00:05:03] MP: Yeah. So, when we're talking about breaking out, I mean, usually what we're discussing is we have defined upfront some sort of access that this container should have. It has certain files it can access. It can access certain network resources, so on and so forth. If it has some way to access resources it shouldn't have access to, maybe other containers on the system, then that sort of when we say it's broken out of its define value.

[00:05:28] YT: Basically, the mechanism that use for these container isolation is fairly complex. So, downside of that such kind of resource and as a consequence of this breakout is that – Like I mentioned earlier. You're not supposed to be seeing what other users are doing. A malicious attacker exploits this, basically that person can look into or have control over other people workloads, and worst case, the entire system. That is the problem of this container breakout.

[00:06:00] JM: There are numerous ways of isolating our workloads. Before we talk specifically about containers in gVisor, let's talk a little bit about virtual machines. Describe how virtual machines isolate workloads.

[00:06:16] MP: Yeah. A virtual machine sort of at the high-level is providing – As the name says, a complete virtual sort of physical machine in which a full guest operating system is run. So, you're running Linux inside of this virtual machine which is running on a physical machine. It's running Linux or some other hypervisor.

So, at a high-level, you have some sort of system API that any sort of application can access. For Linux process it's all the existing calls that you're making and so on. For a virtual machine, that's the virtualized hardware resources it can access. It needs to go hypercalls that it might have to talk to the host.

So, the way a virtual machine is isolating workloads is it's giving a distinct machine to each of these workloads and then creating as much of a limited APIs that came in for that machine so that there's a [inaudible 00:07:15].

[00:07:17] YT: Also, keep in mind that virtual machines are actually the most – One of the oldest concept if you look back, or open up the textbook. Because chopping up your machine was the most perhaps simplest way of doing and it had a much more strong or sort of an indication or limitation even. Because once you chop it, you're not going to change it. It's a machine still.

To overcome the sort of a limitation, the operating system sort of get developed. Interesting happening here now is that as a sort of extreme of that, and we understand that, "You know what? We do want to have a stronger isolation even for containers that's built on top of those kind of abstractions and this kind of fancy world of what would be the kind of interesting way to do the container isolation in a proper way?"

[00:08:04] JM: Let's talk a little bit about the resource consumption of our different isolation paradigms. We have virtual machines. We have containers. My understanding is that virtual machines consume more resources than a container does. Could you contrast the overhead of these two operating environments?

[00:08:28] YT: Actually, I think the statement you made is more about impression. What I mean by that is those are actually two different layers, right? Virtual machines are actually one layer, like a couple of layers below of the containers. Therefore, why people say virtual machines are heavy is because even if you actually chop up particular box or resource as a virtual machine. To run your application, to run your container, you need a guest operating system in that virtual machine.

That is a fixed cost. In all the virtual device simulation, those are going to be the fixed cost versus – I'm not talking about gVisor. I'm talking specifically more the traditional need of container. In that case, all you have is the host and you have the operating system and you just share that kind of fixed resource, a fixed cost among a bunch of processing containers. That's probably why people think containers are much more lightweight than the virtual machine, because virtual machine always come with additional fixed cost after the isolation. Just kind of a high-level, but I hope that makes sense.

[00:09:38] JM: It does make sense. So, if we're thinking about the virtualized environment. Let's talk about a purely virtualized environment. There are some approaches to security in this virtualized environment. There are things called seccomp, SELinux, AppArmor. Describe the approaches to implementing security, because it sounds like the main thing we're getting from isolation is this security. So, what are the ways that we can get security in a purely virtualized environment?

[00:10:14] YT: Let me just add a little bit more clarity on the technology you just mentioned, especially the seccomp and AppArmor and SELinux. The reason because those are actually on top of the existing operating system, not necessarily a virtualization per se. What I mean by virtualization is more about chopping up the hardware or leveraging a kind of a CPU support to actually provide a kind of a stronger boundary.

AppArmor, SELinux [inaudible 00:10:40], we call that MAC, mandatory access control. Seccomp filter is more about the kind of reduce in attack surface between your application and the host Linux operating system. With that, I'll pass the microphone over to Michael to talk about.

[00:10:56] MP: Yeah. So I can sort of describe sort of how this work, and I would also say I think it's important to note that these, seccomp, AppArmor and so on, they're not really specific to a virtualized environment. If you're running on a bare metal machine, you can also use these to isolate your workloads.

But at a fundamental level, what all of the technologies you mentioned do is they allow you to sort of implement a specific sort of white list set of rules of what an application is allowed to do. I mentioned earlier you have sort of this system API that an application can access. For a normal container, it's the system calls and files and things that that container access to.

In seccomp for instance, you can go through and specifically say, "These system calls are not allowed. I don't think they're secure or there's some risk in them. So I'm going to just block them. Similarly with AppArmor, SELinux, allows sort of putting access control on files and other sorts of resources. So, they provide a good way that you can take an individual application and put a pretty tight boundary around it to sort of limit that system API it has access to.

That does come with some downsides, primarily that you really do have to specifically build your filters and white lists around that one application you have. It's a lot harder to come up with a sort of general purpose white list. Anything that you are blocking, that application just sort of fundamentally can't use because it's blocked.

[00:12:20] JM: So, you said this is security technology that's agnostic of a virtualized environment. This is something you apply on a per process basis?

[00:12:30] YT: Pretty much yes, right? So, if you do have the kind of knowledge about particular application, right. You don't need to actually expose at other kind of attack surface, because that could lead to a further compromization.

Actually, this mechanism are all implementing operating system and it's actually more geared toward like, "Okay. If I understand the properties of application, then I'm going to apply it." So it's not necessarily virtualized or virtualization. It's actually more about controlling the interface. Controlling the capability in the application operating system.

[00:13:07] JM: I think I'd like to just ask you guys for a brief overview of the architectural stack of a typical host. So, we've got the host machine. We've got a hypervisor. We've got virtual machines. We've got containers. Then there are some layers that are in between these things. Could you just paint a picture for the listeners who are less familiar with these architectures? What does the stack look like?

[00:13:37] YT: Yeah. Just keep in mind, we're talking about various technologies and sort of implementations that were sort of developed from different contexts. So, I can totally understand from a user perspective, it is overwhelming. So I'm just going to list up from the bottom stack. Just to make sure, a user doesn't have to use all of them. A user can actually pick and choose. But I'll just list all the stack in word so that we can actually have a kind of clear – Like a TCP/IP networking as well. You can have a bunch of things you get confused, but I'll try to kind of layer things from the bottom. Okay?

[00:14:12] JM: Got it.

[00:14:13] YT: So, at the bottom, you have the hardware apparently, and this hardware, depending on the processor and architecture, you may or may not have the virtualization support, and there are different kind of virtualization, but I'm not going into it too much. But, usually, on top of a hardware, before the operating system comes into play, usually there could be a hypervisable or hypervisor, because it needs to kind of chop up the machine. So it has to kind of sit on top of the hardware. Again, if you're not going to use a virtual machine, this hypervisor is not needed. So, it may or may not exist. Sometimes people call it bare metal, occasionally hypervisor. All right.

On top of that, there's an operating system, which will basically say, "Oh, I got the machine. I got the machine, and my role is to provide an abstraction or like a shared resource among the applications." So there's an operating system up top.

Then there comes a container before getting into the application. Container, again, is sort of, it's a little bit kind of – I might be wrong, but an extension to the traditional operating system, which will actually provide more like a little bit of virtualized view to the application as if it has kind of some kind of a more like, or as if it owns an operating system. From an application perspective, you won't be able to see other people. For example, if I have Yoshi's app, I won't able to see Michael's app on the other side, or even I don't know Michael is a user. Those are kind of the container abstraction that comes on top of an operating system.

Then you have finally the process or the application, which has, "Okay. This is a unit of an execution on top of the operating system." These are kind of how I will list the basic stuff in the first place.

[SPONSOR MESSAGE]

[00:16:08] JM: As a software engineer, chances are you've crossed paths with MongoDB at some point. Whether you're building an app for millions of users or just figuring out a side business. As the most popular non-relational database, MongoDB is intuitive and incredibly easy for development teams to use.

Now, with MongoDB Atlas, you can take advantage of MongoDBs flexible document data model as a fully automated cloud service. MongoDB Atlas handles all of the costly database operations and administration tasks that you'd rather not spend time on, like security, and high-availability, and data recovery, and monitoring, and elastic scaling.

Try MongoDB Atlas today for free by going to mongodb.com/se to learn more. Go to mongodb.com/se and you can learn more about MongoDB Atlas as well as support Software Engineering Daily by checking out the new MongoDB Atlas serverless solution for MongoDB. That's mongodb.com/se.

Thank you to MongoDB for being a sponsor.

[INTERVIEW CONTINUED]

[00:17:31] JM: To go a little bit deeper into this typical stack. There's something called a virtual machine monitor. A virtual machine monitor allows virtualized hardware to be exposed to a guest kernel that would spin up on top of the underlying host. Describe what a virtual machine monitor is in more detail.

[00:17:54] YT: Got it. Let me take a step and then pass it over to Michael. So, I mentioned hypervisor, and I think you're spot on that I didn't kind of skip the VMM, virtual machine monitor, to make it simple. But, usually, hypervisor and virtual machine monitor kind of work together.

What I mean by that is hypervisor is there to kind of chop up mostly the CPU-oriented sort of resources. For example, let's take a look at a KVM. KVM itself is a hypervisor, kernel-based virtual machine, that works with Linux to provide a hypervisor capability.

Now, from a guest operating system, it's great, but it still needs to be sort of have access of virtual devices other than CPU, other than memory, such as networking, disk. There are bunch of devices, keyboard, microphone, you got everything. Usually, those are actually handled by a component called QEMU, quick emulator I think. It was initially an emulator, but it became a virtual machine monitor as it evolved.

So after the hypervisor says, "Okay, this guy should be having its own virtual device," but this hypervisor itself cannot provide an abstraction, and then it will forward it to QMU or any kind of virtual machine monitor to again provide a device view beyond CPU and memory. That's kind of how they split the role. I think that's going to come on our picture that we're seeing the modern hypervisor and VMM kind of layout.

[00:19:27] MP: Yeah. I think one thing to also note when we're talking about this is usually with virtualization, we're often talking about hardware virtualization, things like Intel's VMX, the sort of extension on the CPU that support virtualization. The important thing to note about hardware virtualization extensions is they don't provide all of the capabilities to create a virtual machine. Intel's capabilities are mostly about providing a sort of additional privilege level so we can have this privileged guest operating system, but it doesn't actually have ownership of the whole machine. But it doesn't provide virtualized hardware, for instance. So that's where things like

QEMU or other VMMs come in is that there in software is sort of emulating what that virtual hardware would do that's not provided by the actual CPU virtualization.

[00:20:15] JM: Okay. Let's start to move the conversation towards containers. Containers have been around since before Docker. We know that Docker containers provide some kind of isolation. There has been a legacy of different container isolation models as I understand. How has that model of isolation within a container changed overtime?

[00:20:44] MP: So, I think when we're looking at containers historically, they've sort of I think evolved overtime. There's not really one thing that I would necessarily call a container. You can go back and I would say – You want to say the original container is just sort of the idea of [inaudible 00:21:01], which is like it's a unique feature from – I don't know, probably the 80s, where you can say, "This folder now looks like it's the root for my file system." So, now, you can sort of make the file system look different from an application. Which is kind of the basics of what containers still do today, but that was only doing very simple file system operations. It's fairly easy to break out of, I didn't want other sorts of isolation.

So it's kind of evolved from there. Much, much more recently, we've had newer forms of container isolation. All the features that modern Linux containers use that each sort of add different types of isolations. So you're adding isolation for file systems. You're adding isolation between being able to even see other processes. You're adding sort of like CPU memory constrained isolation and so on.

[00:21:51] YT: I think on top of what just Michael said, the complexity property of the container as for technology and the history is that it comes from more on the incremental part. I just want you to know that, actually, Google has been sort of at the center and on the modern sort of container development. For example, C groups. I think in the 2007 to 8 timeframe. The context there is that – It kind of predates me, even, but I knew it. The context here is that to squeeze out the most cost efficient infrastructure, you need containers, you need operating system interaction.

However, you need to augment the operating system capability at that time of Linux to provide a good resource isolation. How that's I understand, how C group got introduced and how the Linux container gradually evolved.

The part that – Docker is a great sort of kind of phenomenal kind of milestone in my opinion, because the traditional Linux container did have sort of isolation capability, like Michael mentioned. But what was sort of lacking was usability. The great part of Docker in my opinion was the sort of revolutionary usability came into play in addition to all the sort of like kind of geeky isolation in a technology. So it became more like a usable thing and it became more like an application deployment unit that people didn't have to worry about how to set up a container or I wanted to just deploy at my app as if I'm running a virtual machine. I think that is a sort of kind of inflection point if you look back what happened throughout the last nearly 20 years.

[00:23:37] JM: I think you alluded to this idea that containers are not really a fixed thing. The idea of a container, from what I understand, is an abstraction unit that sits on an operating and divides up the file system resources and the CPU resources in some way that allows people to make more efficient use of a host operating system. Then there's many, many, many ways that we could do that. Would you say that's an accurate description of what a container actually is?

[00:24:18] YT: I would agree with that, and I think you forgot one another important, is memory. Memory is also very precious resource, right? In case of virtual machine, you need to pre-allocate such memory, because it is a machine. On the other hand, if it's an application or application container, which actually don't even claim memory or the application doesn't reclaim the memory, the operating system doesn't have to. Why would you? You don't have to. If a memory is not actually used, you can actually reclaim and swap it out and swap it in whenever necessary.

These are all very strong capability that the operating system containers by nature have. I think that's sort of a – I'm already jumping, probably going ahead a little bit. But that's another – It's one of the key motivation for gVisor, because we did want to offer a stronger. We did want to overcome with the limitation a container had into an isolation perspective. But we didn't want to lose that very strong capability of the containers, because the resource efficiency is so critical for us.

[00:25:25] JM: Right. Yeah. We can just imagine, if we tried to run a multi-user environment in Apple operating – Mac operating system in a naïve way. You can just imagine, “Oh! I can access the same file system as Jo, and I can look at Jo’s files. I can look at the processes that Jo is running. This would be the most naïve form of running containers in the same environment. Running two different user’s workloads,” and that would obviously not be preferable. So we can see that we need to do something around shielding these workloads from each other.

So, what are the security issues that we have seen in the wild from the current containerized environments? Why is the issue of workload isolation important and how has a lack of a proper isolation impacted people?

[00:26:23] YT: You might know, because it’s a very good question. I think there are ton of probably cases. You don’t mind if we just going to pin down to one particular case to kind of highlight example. It’s not only example, but just like a highlight.

[00:26:36] JM: Sure. Absolutely.

[00:26:38] YT: [inaudible 00:26:37] is probably the –

[00:26:39] MP: So, one particular example of this is – I think this was in 2016.

[00:26:45] YT: 16, yes.

[00:26:45] MP: A vulnerability called dirty cow, which was basically a high-level sort of a race condition and Linux kernel’s handling of copy and write memory. Basically, copy and write memory is a way that the kernel saves resources and avoids duplicating memory a bunch of different processes. Then when someone changes it and changes their own personal copy, it gets moved to its own private version so that no-one else sees it.

There was a race condition where, basically, you could avoid that copy and end up seeing someone else’s private memory. So, two different processes – One process could basically read another across this memory that it wasn't supposed to be able, and that can give it access to any data it has.

So, this is just one of many examples of – It was a vulnerability in the Linux kernel. It was, I think, fixed fairly easily, but it was still there for a long time. It had to be discovered and sort of one of these issues of we're sharing this kernel resource, which is great for efficiency, but does have this issue that when some issue is found, this single bug is potentially letting you break out of your [inaudible 00:27:53].

[00:27:54] YT: Jeff, like we discussed last time. I think it's more like a trend that we're sort of being preparing for. What we mean by that is no matter what the importance of the operating system or the kernel will remain or even be broader. There are more and more features coming. In that situation, the possibility of such bugs that lead to security incidents may increase. Thus, looking at the past artist is just a more like an example and way to understand kind of the potential threat. It doesn't necessarily mean the same would happen. However, it's more about in the future, as things grow, we don't know where the new holes are or even the holes are never uncovered yet.

I think that's why we're sort of being careful. We wanted to use container for resource efficiency, but we need to start thinking about how can we add additional shield, like you mentioned, to prevent, or like augment, the trend that we're seeing?

[00:28:58] JM: Before we get to gVisor specifically, there is a model for container isolation, where every container gets its own virtual machine. This is something called hypervisor containers. Why would we want to give every container its own virtual machine, and is this a compelling model for workload isolation?

[00:29:21] YT: I think it's, again, natural direction to a certain degree. What they [inaudible 00:29:28] relevant technology project, a product try to do is that, "You know what? We're just going to have a strong isolation boundary by using the virtualization's hard isolation capability." I think it's a sort of one design, one choice, to go down a path. By adding, by chopping up, by allocating container into – Sorry. Allocating that container into a VM, meaning that a VM can only have one container, one unit, one user there. Then even something goes wrong, just contain that virtual machine so that we can protect the rest from it. It's a sort of on the high-level principle that I can see.

[00:30:14] JM: So, Yoshi, we explored gVisor on a previous episode. But I'd like to go deeper into it in this episode. Let's start with just a simple overview of gVisor for people who have not heard that episode. What is gVisor?

[00:30:32] YT: Okay. Let me take that and let Michael to kind of add more to details. So, gVisor in a nutshell, they're very, very high-level focused on three things. Stronger isolation or defense in depth, and the resource efficiency of the container and the speed and also without modifying the applications. So, basically, it comes down to strong security or defense in depth, resource efficiency and speed. Then, the kind of ease of use in a big large three pillar. We actually say this in the website too.

Going down to the mechanism to make that happen. So, I'm going to think starting from a concept to the actual implementation. We sit between the application or container and host operating system. We provide separate independent kernel. I think it's important to say this clearly, because I realized that for people who are assessing these technologies, the sort of isolation kind of resonate very well. We do offer a gVisor kernel called Sentry that would emulate the behavior of the host operating system so that the container running on top of gVisor does not necessarily kind of directly talk to the host operating system, which will significantly reduce attack surface and even separate out the world. So we're virtualizing the operating system, on the online operating system was gVisor. We're not virtualizing the machine. We're virtualizing the operating system in a much more secure fashion.

[00:32:13] JM: Okay. Michael, is there anything you would want to add to that description of gVisor?

[00:32:18] MP: Yeah. I think that was a really big description. One think I guess I would add is that there's sort of two key principles that we're sort of following with having our own user space kernel, but that is sitting in between the application and the kernel. That is admission the system API before that we're trying to sort of reduce the attack surface on the host or basically on anything outside of the container. We're doing that through a sort of multi-layered approach, where basically the applications only get to talk to our kernel.

So, because they can only talk to our kernel, they can't directly even sort of attempt to attack the host kernel. Then there's an additional boundary where our kernel does it self talk to the host kernel, but through a more limited API. So, like we talked about seccomp earlier. Our kernel sits inside of its own seccomp sandbox. Limits the things it's allowed to do. So, even were someone to find the vulnerability in the Sentry and sort of gain control of that. They're still sitting inside of a sandbox with a reduced surface to the host that they now need to figure out how to get around.

[00:33:24] YT: Yeah. The one Michael just said is really important, because we always being kind of view gVisor as one entity, which is absolutely true. But, we then use gVisor as sort of entity. There's even two layers of your isolation, and that's the standard that we've been using inside of Google. Meaning, like Michael said, even an attacker can compromise the gVisor kernel called Sentry. That is still under very limited tied environment. An attacker has to do a lot of work to even go through that to eventually even talk to the host kernel. I think that's kind of a multi-layer defense is the kind of the one key principle at the implementation of gVisor.

[00:34:08] JM: GVisor is a fairly low-level technology relative to where many people spend their time. So it may not be intuitive where in the stack this thing actually sits. We talked a little bit about the high-level architecture of virtual machines and containers and I think people are starting to get a sense for kind of how that fits together. As I said, you've given an outline for that. Where in that stack does gVisor sit?

[00:34:39] YT: It's an amazing question, and I will actually have to admit that gVisor is the interesting thing the way that it can sit in a multiple place. Always go one by one. So, first of all, consider gVisor a more like a new paradigm, I will say. The reason is because if I just try to apply to the kind of vertical stack, it can move around in multiple places from a technical viewpoint. So, consider more a gVisor a new kind of paradigm that we're seeing the container world.

But let me just go back to that sort of a vertical step very quickly. In gVisor, we have two modes. We have two modes, which work fully differently. KVM mode, as it states KVM. GVisor, you said like some hypervisor. You have KVM. What does it mean? It's one of the FAQs we get. So I'll start from there.

So in case of a KVM mode, it will probably sit at the hypervisor and then there's gVisor coming in to back to the operating system layer. What it means is that the container is contained within this sort of virtualized environment. It's not a virtual machine, but it's a virtualized environment using the virtualization technology that we discussed at a hypervisor level.

Now, the application will think, "Okay. I have my own kernel and it's all –" The application cannot sit inside a virtualized environment, but it is in the virtualized environment. To implement that operating system view, it goes back to the – It will level up, up level, to the operating system layer again, because it's sitting as a kind of user lane kernel. Then all the flows will go through and come back. So, it's kind of – I mean, I think white board to describe that, right? It goes down on to the bottom. It comes back, and then comes down, and come back to the container world again. So, that's why we see if – I think this is really one of the kind of challenging part of understanding gVisor. I will just say think of it as a paradigm. It's a little bit better to say so that you can detach from the vertical layer. But from a mechanism perspective, go down to the bottom and then come back [inaudible 00:36:49] view. Then pipe through it. That's a KVM mode.

The [inaudible 00:36:54] mode is a lot more simpler, because it does sit probably between the container and the operating system in a nice way. The system calls from the container is intercepted out of the operating system, redirected to the user end kernel and then the rest of the thing will similarly follow to the host operating system for the safe calls. So, it's probably nicer to just say that, yeah it sits between the container and the host operating system. But you can imagine this two most already kind of touches around the multiple layers of sort of like stack that we just described. Consider more gVisor like agnostic to those architecture. The whole point was to sort of somehow insert a virtualized operating system view to the containers so that it does not have direct access to the underlying resources, which is operating system in this case.

[SPONSOR MESSAGE]

[00:37:56] JM: You probably do not enjoy searching for a job. Engineers don’t like sacrificing their time to do phone screens, and we don’t like doing whiteboard problems and working on tedious take home projects. Everyone knows the software hiring process is not perfect. But what’s the alternative? Triplebyte is the alternative.

Triplebyte is a platform for finding a great software job faster. Triplebyte works with 400+ tech companies, including Dropbox, Adobe, Coursera and Cruise Automation. Triplebyte improves the hiring process by saving you time and fast-tracking you to final interviews. At triplebyte.com/sedaily, you can start your process by taking a quiz, and after the quiz you get interviewed by Triplebyte if you pass that quiz. If you pass that interview, you make it straight to multiple onsite interviews. If you take a job, you get an additional $1,000 signing bonus from Triplebyte because you use the link triplebyte.com/sedaily.

That $1,000 is nice, but you might be making much more since those multiple onsite interviews would put you in a great position to potentially get multiple offers, and then you could figure out what your salary actually should be. Triplebyte does not look at candidate’s backgrounds, like resumes and where they’ve worked and where they went to school. Triplebyte only cares about whether someone can code. So I’m a huge fan of that aspect of their model. This means that they work with lots of people from nontraditional and unusual backgrounds.

To get started, just go to triplebyte.com/sedaily and take a quiz to get started. There’s very little risk and you might find yourself in a great position getting multiple onsite interviews from just one quiz and a Triplebyte interview. Go to triplebyte.com/sedaily to try it out.

Thank you to Triplebyte.

[INTERVIEW CONTINUED]

[00:40:15] JM: When I spin up a container that includes the protections of gVisor. What happens? Where is the gVisor added and what is allocated to give me the protections of gVisor?

[00:40:34] YT: I see. I think you're talking about kind of – I think it's good to go through a boot flow, like a launch process of gVisor. I think Michael can talk about it more.

[00:40:42] MP: Yeah. So, when gVisor starts up, fundamentally, gVisor starts and it's just a normal Linux process. Your container runtime are like Docker, for instance, starts this gVisor process and it's going to sort of initialize itself to be a kernel. It's given like here's the file system you're going to serve. Here's like the command line of the program that should actually be run. What it's going to do is it has – There's a bunch of internal kernel data structures that get initialized, but aren't particularly interesting and really do much per se.

But we prepare. The kernel has its own memory management system and has its own virtual file system. That sort of hits that up. Then we actually go – As Yoshi mentioned, we have like KVM versus ptrace and we're sort of setting up the execution environment. Basically, what gVisor operates on is some sort of application execution environment where it can sort of map memory into that environment and it can run code in that environment, and that might be like virtual machine, like VCBU thread. That might be a little ptrace process for the ptrace.

So, we set up one of those. We grab the binary from the file system that's going to be run. We load it into memory and we sort of start executing in that environment and then can start handling calls that come back to us as they come.

[00:42:03] YT: Jeff, the interesting part of this whole thing is that whenever we talk [inaudible 00:42:08] to improve gVisor, the way that we intercept the syscalls is somewhat limitless to a certain degree in a way that if there are new architectures, there's a new environment, there probably will be a different way of capturing.

The reason I said paradigm in such way is that as far as there is a way that the platform owner can provide us the right way to it. Then, basically, gVisor will come to it and say, "Yup, we can just hook it up so that we can provide a virtualized view." We're kind of agnostic to the way that we intercept the syscalls.

[00:42:44] JM: Cool. What is the overhead of gVisor? How much additional penalty is this adding to my system?

[00:42:57] YT: It's very challenging to quantify the overhead. However, please take a look at the document and gVisor.dev, because have done an intensive sort of analysis about that. It's kind of difficult to convey it entirely through the podcast. So, I'll work on Google and reading the doc.

In addition to the report, we also open source the sort of tools and the process how we measured the data so that it's reproducible and also it can be applied to other mechanisms if anyone want to do. So, I will recommend that, in general, from my blog post or to talk at Next. For any sort of kind of a CPU-oriented sort of workloads, just within like roughly 2%, depending on the system call sort of a frequency, right? If you have like an FFM peg sort of a transcoding workload which will offer kind of the data and then – And you kind of nicely batch the IOs. The overhead will be around like 5 percentage.

Versus if you have a very sort of short internal, like a frequent IO hitting kind of application for whatever reasons, then overhead could be around 30%. So it varies a lot, but I don't want to sort of say this is only single digit. Rather, I wanted to point to our gVisor.dev, because we have done the analysis in a much more cohesive way.

[00:44:23] MP: Yeah. I think one thing I would add real quick is I think – Yoshi alluded to this. Is that the primary place where you see overhead added is in that cost of actually intercepting the applications calls to the operating system. In other areas, things like if something purely on CPU, you're basically getting no overhead because it's just run. Also because gVisor is running as a sort of normal process on the host. Things like startup is very quick. Your memory usage is usually very, very similar to running without a sandbox. So, it really generally that sort of some calls tend to be the area we're mostly interested in looking at.

[00:45:02] YT: Yeah. But, Jeff, I want to sort of – Because we're like talking about the future a little bit. Virtual machines are relatively new. It's just probably 15 or like roughly – Virtual machine with an Intel CPU, let's say. Before that, before like around 2003. There is no way to do today's virtual machines. It was only either emulating with QEMU or [inaudible 00:45:28] sort of para-virtualization, and Intel added that VTX capability and kept improving.

What I'm trying to say here is that, actually, gVisor is also a very new paradigm in that sense. The overhead that we're talking here is just because it's a little bit different from today's architecture, like the traditional, the current virtual machine you're seeing used to be. In other words, that's why I want to call it more like a paradigm, is this is something useful and we can actually get support from underlying hardware. This cannot be the kind of fundamental problem in terms of its limitation in terms of performance perspective.

[00:46:04] JM: Okay. Let's illustrate with an example. So let's say I have an environment where there aer two workloads that are running. Give me a description for how those workloads would be further isolated by gVisor relative to a world without gVisor.

[00:46:24] YT: Yeah. So in this world we're talking is basically it's all about we're reducing the attack surface of the host. The application has – I don't know what the number is now. 350 system calls it can make to Linux and they have all sorts of options. In addition to that, there's lots of special interesting files that have interesting sort of surface in the kernel. Without gVisor, this workload can just sort of catch any of those. If they have a bug, maybe it can exploit them.

Let me back up a little bit. I think, Jeff, you're probably talking about how kind of apart these two containers could be. So, let me take a little bit of high-level kind of a stab at it.

So, we talked about – I think virtual machine world. You get the virtual machine, you get the kernel. So you have kind of the, okay, easy to understand kind of way. GVisor, you can apply the same actually, because like we're talking, each container will have its own kernel. Having its own kernel for an application, that's the limit of the world. That's the sky, literally.

So, by having gVisor's kernel attached to this container. That world is already contained, right? It's sort of an application cannot go further beyond the world that Sentry provides that you use as a kernel. This kernel, for a container, one. [inaudible 00:47:49] get its own kernel. It's not the same kernel. It's the own kernel. The only thing we share is [inaudible 00:47:55] kernel that the container can never ever going to be – No. I will say never, but cannot talk at all. So that's a sort of a way that how these are sort of separated, right? The application cannot see the other side's world. Does that make sense?

[00:48:10] JM: It does. Yes. There are a lot of engineering challenges to building gVisor. Yoshi, I think my understanding is you're a little bit more on the project management side of things, and Michael is deep in the weeds. I was looking at the GitHub commits. I think you're number three or four in terms of how much commitment you've made to gVisor.

Could you both talk about your perspective on implementing and building gVisor and some of the bigger engineering challenges you've had to solve?

[00:48:43] YT: My role as a product manager of Google Kubernetes Engine and gVisor, my role is sort of find and deliver products that users love and give sort of an insight to engineering. What is for me amazing part, like the stunning part of gVisor is, yes, I have a sort of background researching virtualization virtual machines before it existed and the causes until sort of a virtualization. This is sort of – When I found out gVisor inside Google and like how they build, it's just like mind-blowing.

Indeed, like I mentioned, this is one of perhaps the hardest or like a challenging project that I could think about where I never even thought about doing that kind of thing. But they deliver. The team deliver and kept improving and improving, gathering internal users, now external users. I just only have simpler – So it's kind of a tech geek, like a team on gVisor and like a person like Michael.

So, with that, I'll pass it over to Michael.

[00:49:43] MP: Yeah. So, gVisor – You're definitely right. It's a very sort of challenging ambitious project that I found super interesting to work on. From the technical challenge side, I think one of the biggest, most interesting areas we've worked with obviously is the Sentry intercepts application system calls and handles them. That means, effectively, the API we're providing, the applications is the Linux API. Linux is big. It has, like I said, 350 system calls, something like that and there's lots of interesting subtle behavior that we basically need to make sure we get right.

So, it has been a very interesting challenge to sort of built out all those interfaces and really get all the nuances correct. This is where we come down to. We've written – I think we have something like 1,500 different unit tests for system calls that we run. We write these tests, "Is this how Linux works?" See if the test passes. Okay. Now, we need to make sure that you guys has the same behavior. There's just been all sorts of little nuances that you don't even think about or realize until you go to implement in and find out that Linux does something really weird that you read the main page for and it not mentioning the main page was wrong or whatever, like there are some subtle differences.

[00:50:59] JM: Well, as we begin to wrap up, I realized we probably still have not touched on the gratuitous detail of gVisor. But I'd love to know how this fits into the broader perspective of what Google is working on in open source and in the cloud. Because as I understand, this is kind of technology that has existed within Google for a while, but it's probably too tightly coupled to your architecture to just immediately open source what you have internally. So you kind of have – You have the added benefit of getting a reason to do a rewrite on internal software within Google. So it's kind of this clean slate. But I would also just love to know how it fits into the broader strategy of Google and Google Cloud.

[00:51:51] YT: So, to be clear specifically on gVisor, besides those apparent platform specific ones, the code that you see – For example, Sentry, Gofer, those mechanisms, the core part of gVisor is what it is, right? We wanted to open source for – Because we thought that's in an interesting approach for isolation. We wanted to sort of get feedback from the outside world. So, in that sense, I don't think you don't need to necessarily – There's no complexity around the gVisor inside or outside in a broader perspective.

[00:52:30] JM: Cool. So, it's pretty direct open sourcing. How do you imagine it impacting the kind of broader mission of Google's open source and cloud efforts right now?

[00:52:41] YT: So, I think you're talking a little bit more specific why we open source gVisor, right?

[00:52:46] YT: Sure. Yeah.

[00:52:48] YT: So, roughly, at that time when we open source gVisor in early 2018, there are probably two things that are worth mentioning. One is that, it's kind of a litany. But yes we did develop container isolation technology called gVisor, and we gradually started seeing the VM-based approach at that time [inaudible 00:53:07] for example coming up in open source. Therefore, we thought that, "You know what? VM approach is totally understandable." We realized that it maybe it's only us that actually went down this route. Isn't it actually really important from a broader industry perspective to kind of share our own kind of approach and see how that goes? It was actually really more for – That's a sort of a container secure world with the work that we have kind of done.

On the other front, the Kubernetes has already kind of become mainstream already at that time, and the container isolation again was sort of one of the key challenges. The sort of – That was actually called runtime cloud, but there was a demand for defining interfaces for sandboxes.

Because of the momentum, we felt that it's important to share what we have sort of built and see how that fits into the wider industry and also Kubernetes community. That was how we kind of open source gVisor back in 2018.

[00:54:11] JM: Okay. Well, Yoshi and Michael, thank you so much for coming on the show. Yoshi, thanks for your second appearance.

[00:54:16] YT: Absolutely.

[00:54:17] JM: It's been fun talking about gVisor. I'm sure we'll have more opportunities to talk about stuff in the future.

[00:54:22] YT: Absolutely. Maybe we need a more further, further detail, deep down. It may not be enough, but we'll see. Yeah.

[00:54:29] JM: How low can we go? Okay, guys. Well, thanks a lot.

[END OF INTERVIEW]

[00:54:36] JM: Commercial open source software businesses build their business model an open source software project. Software businesses built around open source software operate differently than those built around proprietary software.

The Open Core Summit is a conference before commercial open source software. If you are building a business around open source software, check out the Open Core Summit, September 19th and 20th at the Palace of Fine Arts in San Francisco. Go to opencoresummit.com to register.

At Open Core Summit, we'll discuss the engineering, business strategy and investment landscape of commercial open source software businesses. Speakers will include people from HashiCorp, GitLab, Confluent, MongoDB and Docker. I will be emceeing the event, and I'm hoping to do some on-stage podcast-style dialogues.

I am excited about the Open Core Summit, because open source software is the future. Most businesses don't gain that much by having their software be proprietary. As it becomes easier to build secure software, there will be even fewer reasons not to open source your code.

I love commercial open source businesses because there are so many interesting technical problems. You got governance issues. You got a strange business model. I am looking forward to exploring these curiosities at the Open Core Summit, and I hope to see you there. If you want to attend, check out opencoresummit.com. The conference is September 19th and 20th in San Francisco.

Open source is changing the world of software and it's changing the world that we live in. Check out the Open Core Summit by going to opencoresummit.com.

[END]