EPISODE 1198

[INTRODUCTION]

[00:00:00] JM: Cilium is open source software built to provide improved networking and security controls for Linux systems operating in containerized environments along with technologies like Kubernetes. In a containerized environment, traditional layer 3 and layer 4 networking and security controls based on IP addresses and ports like firewalls can be difficult to operate at scale because of the volatility of the system.

Cilium is EBPF, which is an in-kernel virtual machine which attaches applications directly to code paths in the kernel. In effect, this makes the Linux Kernel programmable without changing kernel source code or loading modules. Cilium takes advantage of this functionality to insert networking and security functions at the kernel level rather than in traditional layer 3 or level 4 controls. This allows Cilium to combine metadata from layer 3 and layer 4 with application layer metadata such as HTTP method and header values in order to establish rules and provide visibility based on service, pod or container identity.

Isovalent, co-founded by the creator of Cilium, maintains the Cilium open source project and also offers Cilium Enterprise, which is a suite of tools helping organizations adopt Cilium and overcome the hurdles of building a secure, stable cloud-native application.

Dan Wendlandt and Thomas Graf are the co-founders of Isovalent. Thomas, the firm's CTO, was the original creator of the Cilium open source project and spent 15 years working on the Linux kernel prior to founding Isovalent. Dan, Isovalent's CEO, has also worked at VMware and Nicira. They joined the show today to talk about why Cilium and Cilium Enterprise are a great choice for organizations looking to build cloud native applications.

I am investing in infrastructure and developer tools companies. If you're building something cool, send me an email, jeff@softwareengineering.com. I would love to hear from you. Also, if

you want to support the show, you can go to softwaredaily.com and become a paid subscriber. Paid subscribers get access to all of our content without ads. So if you want to listen without ads, go to softwaredaily.com and become a paid subscriber.

[INTERVIEW]

[00:02:16] JM: Thomas, welcome to the show.

[00:02:18] TG: Thanks for having me, Jeff.

[00:02:20] JM: Containers have been a big change to how applications are managed. How have Kubernetes and containers changed application management?

[00:02:29] TG: I think the biggest, Jeff, Kubernetes comes in a two-phase shift. Like the first big shift was container and the container image format, which fundamentally changed how we deliver applications. So instead of delivering individual applications which have dependencies on the system with shared libraries where you have to somehow manage that, we moved to a system where an application is shipped as a container image with all the dependency baked in. So you're basically just running entire container images. And that has been a massive change and it has rendered Linux distributions almost entirely obsolete. Like you no longer require a Linux distribution that provides all the dependencies. You can use a very small one. So we have seen all of these container optimized Linux distributions come up that then host container images.

And then the second biggest, or the second big phase, was the introduction of Kubernetes as an orchestration system which fundamentally changed how we actually schedule and run large distributed applications or how we use infrastructure across highly multi-tenant teams. I think both phases are very important although quite different in what they brought. So when we talk about container age and the age of Kubernetes, I think both steps have been strictly required to get to where we are today.

[00:03:47] JM: And what kinds of downstream effects have those shifts had in your mind?

[00:03:54] DW: A couple of ones. First of all, it has fundamentally changed on how application teams deliver applications all the way from how applications are being developed. So we went from large applications, single, so-called monolith applications, to microservices, lightly coupled and so on. We fundamentally changed how CI and CICD operates. These had consequences on how you have to secure your CICD pipeline. But there's also been other changes such as with introductions or with a migration from VMs to containers who are now all of a sudden running much larger numbers of containers compared to a smaller number of VMs, for example. So there's both a people side on how it impacted application teams and how that change of development impacted application security and so on. And there's also a scale side and a multi-cloud infrastructure side on where we run this? At what scale? And that has secured implications and networking implications as well.

[00:04:59] JM: Let's go deeper on the networking implications. How does networking with Kubernetes and containers compare to networking with VMs?

[00:05:06] TG: Yes. This is an excellent question. I think if you look at networking overall and where it came from, obviously in the beginning it was very physical, right? We plugged in cables. Like we plugged in physical servers and plug the cable in and plug the cable into a physical switch to connect networks or to connect servers together. With virtual machines, we virtualized everything. We moved more of these things to software, but we didn't fundamentally change much in how we thought about networking. Instead of having physical cables and physical switches, we now had like virtual cables and virtual switches and virtual routers, but we didn't really move away and we still assumed everything is machine-based, like virtual machines. It's a device and you're connecting devices. If you're talking firewalling and you're talking virtual firewalls, and nothing fundamentally changed. We just virtualized everything.

And with containers, the first move actually was to do the same again for containers. So we very much treated containers as miniature VMs in the very beginning. And if you look at the early Docker time, networking was still done exactly the same way as it was done for virtual

machines. If you look at networking now, if you look at modern cloud native networking solutions, it's fundamentally different. It's closer to a cloud native or a microservice aware networking layer which is not device, not machine-centric, but deeply cares about the actual application, which is in the end what you want to care about. You're not necessarily caring about some machine or some device or the containment in which the application runs in. You care about the application. You want to understand the application. You want to secure the application. You want to monitor the application and so on and so on.

[00:06:52] JM: Can you tell me more about that shift? Like from VMs to containers, and we originally sort of viewed this as just containers are yet another VM. But over time, we're changing our perspective and looking at containers as differently than VMs. Go a little bit deeper on the distinction there.

[00:07:12] TG: A very good example is if you treat a VM and a container in exactly the same way, you can reapply all of the existing security and networking principles that you're used to in the world of VMs. You can use existing IP and port-based firewalling, for example. So yes, it is possible to say, “I'm going to secure all my containers from a networking perspective using IP-based filtering and I'm going to allow the IP of container one to talk to the IP of container two.” It is possible. It will function, right? But it won't scale as you become truly cloud native and as you really scale out.

What you want at that point is actually to start treating applications and understand them. So typical example, in the world of Kubernetes, you won't be running unique containers per se. You will be running services, and those services will have like any number of replicas. Because if your application that you're on is in high-demand, you will have to so-called scale it out. You will have to run many, many replicas of the same application in individual containers to demand or to fulfill with the load demand. And in that context, if you're running multiple replicas, which are identical, from a security perspective, all of a sudden you can start to simplify everything and build a better model.

Very simple example, you have a couple of hundred frontend containers, which implement your frontend service and you're running a couple of dozen backend containers to implement your backend service. From a security perspective what you want to say is my frontend is allowed to talk to my backend and you want to assign unique security identities and you want to treat them as applications. Like this is my frontend. This is my backend. Frontend can talk to backend.

If you do this with so-called security identity, you're defining this allow rule once. You're saying like frontend can talk to backend. And as you continue to scale up and down, nothing has to fundamentally change from your firewall perspective. Compare this to your typical IP-based firewall where every single time you start a new container or you stop a container, you have to update all of your virtual firewalls to now add the IP of the new container or remove the IP of the container that was just removed.

It's very obvious that this later model has much more difficulties to scale. And there's an even more important aspect here because, for example, IP addresses are tied to the online infrastructure. If you're running in multiple cloud providers or if you split your workloads across both cloud providers and maybe on-prem infrastructure, if you're are relying on traditional IP-based filtering, you're somehow tying your security to your infrastructure, which is not what you want. You want the security. You want the visibility to be on a higher level. That's what we call application-aware. So you want to think about services applications. You don't want to think about infrastructure-specific elements such as IPO addresses and ports.

[00:10:24] JM: Okay. Interesting. Tell me more about the problems with network connectivity in containerized applications.

[00:10:33] TG: Typical problems arise from the big fundamental shift in dividing large applications into multiple microservices. So what used to be a function call inside of an application calling from one piece of code into another piece of code is now a network call, right? You have, let's say, my frontend app is talking into the billing app or into the billing service and it's making a GRPC call. And while a memory function call is very, very unlikely to

fail, the network is not 100% reliable, right? The network can potentially fail. So let's say frontend talking to billing service. This operation can fail. And if something can fail, you need to monitor it.

So from a networking perspective, we all of a sudden are responsible for business logic and for business actions, which were previously done inside of an application. We are now in charge or the network is now in charge of making these calls. And it's also in charge of securing these calls. So it's very important that what used to be an application internal detail is now exposed on the network. So you need to secure it. You need to provide visibility. If it fails, you need to be able to tell why they did fail? Is this a problem because the underlying network is not stable? Is it a problem because the network security policy is incorrect? Is it a problem because the service you're talking to is down? And so on. So with the shift to microservices or modern applications, then there's a massive increase in demand on what we ask from our network provider.

[00:12:11] JM: What kinds of new protocols or software applications need to be built to ameliorate that shift?

[00:12:18] TG: I think a lot of that has already been built. So first of all, I think protocols which allow reusability. And good examples of that are HTTP with REST APIs, GRPC and so on. Language formats such as JSON or other formats or other API protocols which allow to effectively program and develop loosely coupled services, which means you can divide your applications and you can develop application components in parallel without creating strict strong dependencies while still being able to reliably transmit the information you need over the network. Good examples of this have been HTTP, GRPC, Kafka and so on. Like these are the typical protocols, which are highly effective in those environments and it's the responsibility of the network to allow for that to happen. And to go further, I think it's actually the responsibility of the networking layer to also provide visibility.

So concrete example here, Cilium, the project that we are developing, gives you visibility into, for example, HTTP, Kafka, Cassandra, Memcached. So instead of just seeing a network call,

“Oh, this service is calling or talking to this service.” You're actually seeing, “Okay, this service named frontend is talking to the backend or is talking to this Kafka cluster and it is consuming on this Kafka topic, for example, or it's making this GRPC call and the GRPC call failed with this status code.” And so on. So I think it has become the responsibility of the network to understand these modern application protocols .

[00:13:57] JM: Okay. So you mentioned Cilium. Explain what Cilium is.

[00:14:01] TG: Yeah. Cilium is a very exciting new open source project that's been around for five years at this point in production in some of the largest Kubernetes clusters out there. It's a cloud native networking and security solution built on eBPF, extended Berkeley Packet Filter, which I assume we'll talk a little bit more about. It's been built from scratch for this new age of cloud native networking or cloud native computing where applications are built as microservices where we have multi-cloud, where you're bridging cloud and on-prem. We're using this modern application protocols where we have new CICD systems. Maybe you're using GitHub, GitOps and so on.

Cilium has been built from ground up for that age. And to tie it all the way back to the beginning, it has stopped assuming this device or a machine-based model, but it's thinking in terms of applications, services and not machines. So it's not thinking IP addresses and ports. It’s thinking frontend. Is talking to backend using GRPC and so on. This is Cilium.

[00:15:08] JM: Okay. So I'd like to revisit Cilium after we talk a bit about eBPF. So eBPF, the extended Berkeley Packet Filtering. What is eBPF?

[00:15:18] TG: Yeah. I think it actually helps a bit to understand my background to understand eBPF. Before diving or a kind of solving networking with Cilium for cloud native, I have been a Linux kernel developer for many years, almost 10 years. Early on, Red Hat, working on REL, Red Had Enterprise Linux, also OpenShift. And I've been developing networking and security subsystems in the Linux kernel for many, years. And the reality was or always has been then, it takes a very, very long time to actually get any new functionality into the Linux kernel and it

takes an even longer time for that Linux kernel functionality to make it to the user, because these users won't be running bleeding edge kernel versions. They will be running a kernel version that is years old. You might be running the latest long term stable release, but your actual major release might be two or three or even four years old.

This is always cost that any sort of functionality that requires kernel code or kernel functionality has to use something that has been in the kernel for many, many years. For example, when we started building out OpenShift, we used kernel pieces that we've baked in 10 years before that, and it's very obvious. Or even if you look at modern container systems, a lot of the container technology is using very old Linux kernel technology. And it's obvious that the kernel team or the kernel developers have not been aware of those use cases when that functionality was written. eBPF fundamentally changes this. So eBPF gives us the ability to dynamically reprogram the behavior of the Linus kernel using a safe and efficient programming language called eBPF.

So unlike a Linux kernel module, which some of you may be aware of, which allows to load additional kernel code in, eBPF allows to do the same, but in a safe manner. So even if you have a malicious eBPF program, you cannot harm the kernel. So it's a fundamentally secure and efficient manner to reprogram the kernel. So we can, for example, make the Linux kernel Kubernetes aware, or we can bake in an HTTP parser into the Linux kernel, or we can teach the Linux kernel about Kubernetes labels or we can teach the kernel about DNS and so on. Topics or features or a certain awareness required for Kubernetes can be baked into the Linux kernel without actually changing Linux kernel code. And that's very powerful. That's the promise of eBPF.

[00:17:48] JM: Why is that so useful to be able to teach the kernel new tricks?

[00:17:52] TG: Because the kernel is fundamentally is basically the platform on which all new containerized workloads run on. If you look at virtual machines, the hypervisor which was running the virtual machines, it didn't matter that much what operating system it was running to the application. Like it was hosting the hypervisor. And then inside of the virtual machine you

would ship your entire guest OS. So you would ship a Linux if the app was Linux-based. So you would ship a Windows kernel if it was a Windows-based app.

With containers, this is different. The containerized application consumes the Linux kernel of the host. So if you have a host running and you have 10 containers running on that host, all those 10 containers will use the same Linux kernel and they will directly interact unless you're using something like clear containers or gVisor or some additional abstraction. They will directly consume the kernel. And the kernel is responsible to do name spacing, to do resource management, to do container security. All of that is the responsibility of the kernel. So the kernel needs to understand these containers in order to effectively provide visibility, security, networking and so on.

[00:19:07] JM: Can you tell me more about the way that this kind of eBPF functionality should get added to the kernel? Like what's the way that the programmer should be interfacing with this new kind of functionality?

[00:19:24] TG: Most importantly, eBPF is very, very, very low level. That's why Cilium exists. So eBPF is meant for kernel developers. It's meant as an alternative to writing kernel source code. So eBPF has not been designed for a standard application developer to write eBPF code, and instead of writing Java code to now write eBPF code. In general, eBPF is targeted and aimed for kernel level engineers. And typically, most usage of eBPF is indirect why a project like Cilium for networking security, or BCC, or BPF trace for tracing, or if you are using maybe an eBPF-based LSM. Or if you're using Kubernetes, you're using eBPF-based sec comp for system call filtering and so on. If you're using Chrome, you're using eBPF for the isolation of Chrome plugins and so on. So like you are using eBPF, but usually indirectly. It's primarily targeted for eBPF or for kernel level developers.

If you want to write your own eBPF program, so let's say you're working on Cilium or another project, then there is an entire tool chain that you can use. So you will write eBPF code as so-called pseudo-C code, because typically you don't want to write a dark bytecode. You could as well. You could write eBPF bytecode, which is x86 assembly like. But typically you

write it in a higher level language and you use something like LLVM or GCC to compile your higher level language into eBPF bytecode that you can then load into the Linux kernel using several load or libraries. And then there is several language bindings from Go, C, Rust, Python where you can write your user space application that can then interact with BPF program as well, which is often why you want to use eBPF, is because you want to extract visibility from the kernel or you want to take control. For example, you want to do filtering of some sort. Then you will always have a user space application portion, which will control that eBPF program as well.

As you load the program into the Linux kernel, the Linux kernel will validate the program. It will so-called pass it through the verifier, and the verifier will validate that the program is safe to run and actually reject it if it's not safe to run it. And otherwise load it. Just-in-time compile it for efficiency. So an eBPF program will run just as efficient as if you would recompile your Linux kernel code, and then load the program. And you would tell the kernel where to load that program. For example, I want to attach this program and run it every time a new network packet is being received or I want to run this program every time a system call is being invoked a particular system call. Or I want to load this program every time a user space application in my application or my in my application code is called, and so on.

So you tell the kernel, “This is my program. This is the bytecode attach it where,” and the kernel will load it for you.

[00:22:29] JM: Great. So you mentioned one interesting thing there. So the kernel is capable of validating whether these programs are safe? How does that work?

[00:22:40] TG: Yes. It's very similar to how JavaScript in a browser is actually operating. So think of eBPF an EBPF program to run in a virtual sandbox where other software, other code is responsible to secure it and to contain it in a sandbox. So it's not hardware assisted. It's code around it that will ensure that even malicious versions of this code cannot harm the linux kernel. Very similar to how your browser is ensuring that even if you have malicious JavaScript code, that malicious JavaScript code cannot harm your browser. So the verifier will scan your

BPF bytecode and it will ensure that the BPF byte code cannot just access arbitrary kernel memory, for example. It will ensure that all the data is actually initialized so that you cannot expose uninitialized kernel memory. It will ensure that all loops are bounded so you cannot loop forever. It will ensure that the overall program complexity is within a certain limit. So you cannot load a program that takes five minutes, because that would stall your entire Linux kernel and so on. So the verifier ensures the program is safe to run by basically validating the actual program logic.

[00:23:58] JM: Gotcha. Is that enough to ensure the safety or is there a need for like open source trusted module sets so that you know that there's an added measure of safety?

[00:24:14] TG: If you look at usage, yes, it is enough. A good example here is all of networking into Facebook data centers is done with eBPF programs. So for example, Facebook has completely replaced their load balancing layer for like internet-facing traffic with eBPF. So that, yes, there is trust. If you look at where Cilium is being used in some of the biggest, most critical Kubernetes clusters – So yes, it is secure. Can it be made more secure? Yes, of course. It will continue to evolve and it's in the interest of everybody to continue adding more and more security layers on top of it. And yes, signing of eBPF programs is currently being discussed as well. So very similar to how signing of Linux kernel modules exists as well where a signature will guarantee that the bytecode that you're loading has been signed by a trusted party. Signing does not prevent or does not replace the validation. So I think it's one more layer on top.

In general, we'll see more and more security simply because of the raw potential of eBPF, we'll see more and more usage of it, which will also drive further security discussion and further this security demand. As common with open source, if something gets very popular, a lot of eyes are on it. Some bugs are found. And a lot of pressure is there to actually secure the overall system. And that's exactly what we have been seeing in the last couple of years.

[00:25:45] JM: All right. So let's go a little bit higher level. Cilium is built on top of eBPF. So what's the interface between Cilium and eBPF?

[00:25:54] TG: So Cilium basically abstracts all of the complexity of eBPF away. If you're using Cilium, you can use Kubernetes natively. You can use the standard Kubernetes interfaces that you're used to and Cilium will implement the standard network interfaces such as CNI, container networking interface, and then translate that into eBPF as necessary. So Cilium uses all of the power of eBPF and can do some groundbreaking new things that are just not possible with things like IP tables, which is an older IP filter, or IPVS, or other kind of more traditional networking implementations or even Open vSwitch. So Cilium uses that power of eBPF but hides the low level complexity of it completely away from the user. Allowing the user to define higher level logic such as, “I'm running this service. I'm running this container image. I want this network policy. I want metrics. Give me network metrics. Give me network flow logs.” So Cilium gives you higher level values, higher level solutions and implements everything under the hood with eBPF.

[00:27:09] JM: What's an example of something that the average Kubernetes user might want out of Cilium?

[00:27:15] TG: There are many, many examples of course, but I think one very typical one – I mean we're seeing like everything from really, really high scale where Cilium really, really excels. But I want to use an even simpler example here. I think something where – Or actually two examples. One is security and one is visibility. Let's dive into visibility first, because I think that's actually the most simple example to use. Typically, if you look at visibility solutions, you have almost two layers. You have a network layer, where the network has been providing so-called five tuple flow locks. Like this IP on this port is talking to this IP on this port. This is typically how the network has been providing visibility so far. So you have like IP-to-IP type connectivity visibility. Or you had applications producing log files. I'm doing X, right?

And I think Cilium with eBPF can bridge the two together. So when Cilium provides eBPF-based visibility, we are not just talking IP addresses. We're obviously including things like service names and pod labels and other Kubernetes metadata, but we can actually dive deeper and understand, as I mentioned, the application protocols. We can't understand the

system calls that applications are making. We can understand the processes that are interacting on the network. We can understand security identities and so on. So we can take it even whole level higher. And the reason why we can do that is because, first of all, the visibility comes from the kernel world. Like you need to be in the Linux kernel to see everything happening, but the kernel does not know about Kubernetes. So how can Cilium provide this? Because Cilium can bake the Kubernetes awareness into the Linux kernel with eBPF. So eBPF allows to provide or to extend the system that gives the ability to extract the visibility into the first place and make that Kubernetes aware. So I think this is a massive step forward.

This allows you to see exactly what's going on from latency measurements, to figuring out is my cluster not behaving because DNS is broken? Yes or no. You can answer these questions very, very easily with Cilium because of the good visibility that eBPF provides. And the second one is security. And as I mentioned, before security as traditionally on the network side has traditionally been provided with firewalls, which is kind of this IP can talk to this IP, or at the application level with something like TLS. I'm doing a mutual TLS or I'm doing authentication between two services or two applications.

Cilium can combine the two again, which is very, very powerful. Obviously, Cilium can do all of the traditional networking filtering if you want to based on IP addresses, ports and so on. But it can actually go much further. It can transparently encrypt. It can do authentication between services, but without making the services aware of that. So you can gain some of the security features that have been reserved to the application world before, but provide them completely transparently to your platform, which is very powerful. Again, only possible because of eBPF, because we can glue everything together. We can use eBPF to make systems which are in a position to provide the security. But what are lacking the actual awareness of Kubernetes, we can make those systems aware of it. That's the power. That's what Cilium brings to the table.

[00:30:44] JM: Let's give an example of an application where you have the Cilium opportunity and you also have other ways you could implement it. So load balancing, for example. You mentioned that Facebook does load balancing with eBPF. How would load balancing with eBPF compare to previous strategies for load balancing?

[00:31:06] TG: We'll actually see here an example where there's a two-step evolution. First of all, like what other alternatives? If we're limited to Linux, other alternatives would include something like IP tables, the very traditional one, which is IP-based. And if you are building an IP tables based load balancer, you're essentially creating a list of very, very – Like a very, very long list of linear rules. Like is the destination IP X? Yes. Then load balance to these three destinations. Then the next rule. Is it Y? Okay. Yes. Then go – So you have this long list of rules and you go through them – For every new connection, you go through that list. And obviously this does not scale at all. It's a very old concept. No wonder like IP tables has been invented 20 years ago. Like we had like 10 megabit network connections back then.

Something like IPVS, like IP Virtual Server from Linux is a bit more modern and you can already scale much better. But it gives you the raw scalability. eBPF can still be a bit better, but there's even a more fundamental difference. eBPF can do the same scale, but also give you visibility at the same time. Very, very powerful. IPVS at this point, when IPVS was invented, nobody cared about like service-to-service communication or microservices and the type of visibility that was required. So we can gain the performance, gain the scale and also give you the required visibility.

Somebody running probably one of the largest Kubernetes clusters out there right now, one said to us, “Like load balancing at scale is not really the challenge anymore, right?” Load balancing at scale while providing the necessary visibility, that's the real key. Because like it doesn't matter that you can load balance millions of packets per second if you cannot figure out why it's broken when it is broken. The key is really to make sure that the system that you're building that will operate at a large scale or at large speeds, they still need to be able to be troubleshootable. It's very important.

But then to maybe add a little bit more visionary piece here, eBPF actually allows to go further. If we typically look at load balancers, they have been operating at the network level, which means that an application is opening a connection, let's say. And maybe let's say a user is opening an app. The app opens a connection. Goes into a data center or goes to cloud.

There's a load balancer in front of the application, which will pick to which replica does this have to go to? This was typically a network-based operation.

With Cilium, we can do something dramatically fundamentally different. We can move the load balancing all the way to the client. And this is nothing new. Client-side load balancing has been there for a long time. Like a lot of the apps that you're using on your phone will be doing client-side load balancing. Even maybe some of your microservices will be doing cloud-side load balancing. Cilium can do so on behalf of your application transparently. So we can translate and do the load balancing operation as part of the system call when your application does the connect system call, which will open and create a new network connection. Very powerful. So you can get the benefits of client-side load balancing without having to change your application.

So even your existing microservice, your existing apps, even your legacy apps can benefit. And this is, again, because eBPF is kind of just – It's almost unfair. It's changing what is possible. Solutions which can only operate on the network level can never get to that system call level. eBPF is very general purpose. It can hook into almost everything inside of the Linux kernel. So all of a sudden we can find completely new approaches on how to solve existing problems.

[00:34:51] JM: So you talked about the idea of the different applications being able to communicate with one another and be more application-aware, like being aware that Kafka cluster is communicating with some service and have that be – I guess I didn't fully understand what you were saying there. You were you were saying that there is an opportunity. Thanks to eBPF and Cilium that you have more application-ware communication. Can you go a little bit deeper on that?

[00:35:21] TG: Yes. Let me try and build up like the stages that you will typically go through. If you just deploy Kubernetes in the standard, let's say, a simple CNI that's just like the networking layer of Kubernetes. That just gives you the connectivity but not more beyond that. Then typically, yes, you will have connectivity between your parts. What will you do if something is not working? You will kind of think back, “What did I do before? Okay, I probably

ran something like TCP dump, “which would show you kind of a packet filter CLI that you can run on your Linux system. And you might start running TCP dump. And what will TCP dump give you? It will give you IP-to-IP. So like you can see, “Okay, this part, this IP, this container is talking to another container with that IP.” And then you will probably start comparing that with output of your kubctl command to try and figure out what IP is this container. And at this point you're probably already wondering, “Okay, this is not good enough. I need something that is more – Like something that is Kubernetes native. So I need something that shows me what is going on in the network while actually showing service names and pod names and pod labels and so on.” This is the first level that Cilium can provide. So it can actually show you what type of network communication is currently going on. And it can also show you if a network packet is being dropped. For example, the network policy is denying this, then it will show you this as well. And it won't just show IP addresses, but it will give actual service names. This is actually also very important if you have some sort of, let's say, a SIEM or some sort of Logstash where you keep such network flow locks in some form.

If you're using IP-based visibility and you're looking at log files which are one week old, looking at an IP address will be useless because that IP address will have been reused ever since by hundreds of different containers.Like containers will come and go. So the meaning of an IP address. And they will reuse IPO addresses. So the meaning of an IP address is useless. So even just that first level of visibility is a – That gives you like persistent static IDs and metadata for visibility is very, very useful.

So let's say you have that and then you're running something like a Kafka and something is failing. Like your application is failing and like the application is saying, “Okay, I cannot produce to this Kafka topic like failure.” Then you will try to dig in and you will see, “Okay, there is a connection between – The connection is successful.” But you won't really know what's going on with the protocol level. So you will start looking into application logs and maybe even into the logs of Kafka. And if this happens occasionally, that might be fine. But what if the network player would actually understand that application protocol as well and would understand failure scenarios? So it would understand, “Hey, this service is returning an HTTP 5

exact status code. Or this is a Kafka error because topic not found or something." And your network layer would also give you that visibility. All of a sudden you are able to instantly troubleshoot.

Another example, maybe even the better one is DNS. How many issues in infrastructure or in application deployment or in failures, how many incidents are related to DNS? How difficult? How easy is it to rule it out? Why does this have to be so difficult? Like it shouldn't. So Cilium gives you metrics. For example, dashboard. You can simply look at which containers are currently seeing DNS resolution failures? Which containers have seen the most DNS failures in the last five minutes? How many failures do I have in general? Which is the most looked-up DNS name and so on? All of those metrics, all of that information that you want to troubleshoot, to monitor a cluster, all of that is collected with eBPF with Cilium.

[00:39:18] JM: This might sound like a really naive question, but how does the communication value for Cilium compared to something like Envoy or a service mesh like Istio?

[00:39:31] TG: I think this is an excellent question. It's also one of the questions that it's not hard to answer, but it can get complicated quickly. So first of all, I think there's a lot of common ground between any service mesh, not necessarily limited to Istio, but let's say Linkerd, Glue, Kuma, Istio, whatever, and Cilium from a high-level intent. So service mesh and Cilium. What do they have in common? What we have in common is we want to move away from this traditional networking, this machine device-based approach. If you look through Istio documentation or through the Linkerd documentation, you're not seeing like IP addresses or machine or like device. It's talking about services, applications. It cares about application protocols and so on. So we all share the same desire and we share the same view that in order to be a more effective networking platform you need to move up and closer to the application and understand it better and provide the visibility to control the security all of that at a higher level.

The main difference is in the implementation. So how is that value being provided to the user and what are the consequences of that? And right now there're two main models. There's the

so-called sidecar-based approach, which is using a proxy like Envoy. It could be an auto proxy as well, but like a typical, an L4 or an HTTP proxy, which is running as a so-called sidecar as part of the application on both ends. And a network logic that will transparently redirect all of their network from that application through that sidecar. So instead of having an app talking to another app directly, all the communication will go through that sidecar. What does this achieve? It gives you transparent visibility and control. So all of a sudden instead of baking the visibility into your application, you can extract it transparently without changing your application. You can enforce, for example, mutual TLS without baking TLS into your application, and so on.

So it's the same desire to transparently gain control and visibility ideally at the application protocol level. So that's the sidecar based approach. The other approach, and that's the approach that Cilium is using, is to do this in the Linux kernel using eBPF.Because guess what? The application is already channeling all of the network communication. Everything that the application does, even if it does not even leave the Kubernetes pod, even if it stays inside the Kubernetes pod and it's actually just talking on the loopback interface, all of that is going through the Linux kernel. So if we gain control over the Linux kernel with eBPF, we also gain control and visibility into everything that the application is doing. This is what eBPF does. This is what Cilium does.

The main difference in consequences are huge. While Cilium with eBPF can gain the visibility and gain the control at really low cost because we're just adding a bit of logic into existing Linux kernel hook points. Whereas a sidecar proxy needs to redirect TCP connections, terminate them, open a new connection to the other side, the other side needs to terminate them again and so. There's a massive increase in latency, in additional resources being consumed. So to summarize it all, the high-level goal is very, very similar between service mesh and Cilium. The implementation is very, very different.

And maybe one other difference, which is less about implementation but more about usability, Cilium really focuses on just using existing Kubernetes principles. So instead of introducing lots and lots of new concepts,we're trying to do everything under the hood while being kind of true

to the Kubernetes core principles. So we're almost hidden and give you all the visibility and to control without requiring you to change much. You can run your Kubernetes clusters the way you want to with service mesh, without service mesh. And Cilium is basically hidden and gives you everything that you needed without kind of putting a lot of consequences on what you have to do.

[00:43:51] JM: You're the co-founder of Isovalent, which is a company built around productizing Cilium and eBPF. Tell me about what you're building.

[00:44:00] TG: Absolutely, yeah. So Isovalent is the company behind Cilium. We've recently announced our Series A funding. We're backed by Andreessen Horowitz, Google and Cisco. We are true believers of open source. So everything we do around Cilium is open source. We have a couple of solutions on top of Cilium that we offer that help our customers to operate Cilium in enterprise-specific context with specific security requirements and so on. But at the core, we are an open source company. We're building the networking and security layer for the cloud native age. And I have made a huge bet on eBPF and Kubernetes four years ago and it's really, really paying off. We're driving some of the biggest Kubernetes clusters. Some of the most sensitive Kubernetes clusters that are currently out there because eBPF gives you just kind of an unfair advantage in what is possible, which has led to eBPF or just led to Cilium becoming kind of the most advanced network layer that is currently out there.

[00:45:07] JM: How do you install Cilium? How do you install eBPF on your network?

[00:45:14] TG: So typically, you can install Cilium on any platform that you want. Whether you're running Cilium in cloud or your Kubernetes in cloud or on-prem, whether you have a metal data center, whether you're using multiple cloud providers. Cilium will run on all of them because all that Cilium needs is a Linux kernel. So right now we're Linux kernel specific. This might actually change in the next couple of months. But right now eBPF per se is a Linux kernel specific technology. So if you're running Linux on your worker nodes, on your Kubernetes worker nodes, you can install and run Cilium. Cilium comes as a – By standard, comes as a Kubernetes daemon set, which runs an agent written in Go on all of your

Kubernetes worker nodes, which will then control the eBPF layer inside of the Linux kernel. And everything that Cilium does and need from a control pane perspective is done via Kubernetes.

Now this sounds like Cilium is Kubernetes-specific. That is not really true. We have actually recently released a version of Cilium that can run on metal nodes or VM workloads as well. What is definitely true is that we're seeing Kubernetes as the new standard of where you want you have your source of truth. Like where you want the network policies to be defined. Where you want the visibility to service and so on. So we're seeing Kubernetes as your starting point and then you hook legacy workloads, VMS, into that system as well. So we're not necessarily container or Kubernetes-specific, but it's typically our injection point, that's typically where our users start out and then they expand beyond the typical container workloads.

[00:46:55] JM: What are the earliest problems that Cilium is going to solve? Like what's the typical problem that I'm looking to solve if I'm an enterprise and I'm adopting Cilium? What's kind of the first thing that I'm typically trying to solve?

[00:47:10] TG: I think it's exactly at the inflection point that I mentioned before, which is once you start caring about visibility, for example, or advanced security. So once you go beyond – Even for relatively simple segmentation work use cases. So Kubernetes has a concept of network policy, which is standardized, which allows you to find what services can talk to what other services. There're multiple network providers that can implement this. The implementations have some minor differences, but they all comply. What they really differ is the amount of visibility they provide, the ability to do compliant around it, the metrics they provide, the troubleshooting around it, the policy management, integration with service mesh. So as soon as you go this one step further where you really care about your workloads. So typically when the really sensitive workloads go in or when the scale goes up, when running TCP dump is no longer feasible, this is where Cilium really shines. It gives you the platform that you want. That gives you the visibility that you want. It gives you the control that you want.

Typically once you go one step beyond just onboarding or migrating your first couple of applications, but you get serious. Another very typical inflection point is once you start doing multi-cluster – So Cilium has multi-cluster ability. So you can, for example, run clusters in different regions or even across different cloud providers and you can connect them together. We have many users running multi-cluster in an edge type scenario. So they have centralized clusters in bigger cloud providers and then edge-based clusters closer to the user. But some services still want to run centralized, for example, some databases. And then edge clusters need to have secure connectivity to the centralized clusters. This is where multi-cluster comes in. Often, when security compliance comes in, for example, I need to encrypt everything because of GDPR. This is where we come in. So very typical for that kind of – The little bit more advanced use case where if pure-based connectivity is not good enough, this is where Cilium comes in.

Obviously, you can also use Cilium for just tiny little use cases. But Cilium has not been kind of defined as this is a stupid small tool. It's been defined as something that gives you the visibility that you need once you have a bit more advanced needs, like whether that's scale, visibility, security requirements and so on.

[00:49:40] JM: I'd like to wind down by talking about the future. I'd like to get your perspective on just the Kubernetes ecosystem as a whole, but particularly from your vantage point of being focused on Kubernetes networking.

[00:49:55] TG: Yeah, that's obviously a very open-ended question. I think even though to all of us who have been in the Kubernetes ecosystem for a while, it looks like Kubernetes is already very far, right? It looks like so much has happened in the last four or five years. But the reality is that Kubernetes is at the very, very, very beginning. Like every year we see kind of an – Like every year it feels like this space is exploding. Like so many more users are coming in. So many more traditional – Like what appeared to be traditional use cases, which would never move away are all of a sudden really focusing on Kubernetes-based strategies, whether that's Telcos, traditional enterprises, databases. Like at this point, almost everything – there's, some of the path is try to be found, to run that on Kubernetes.

What's the most interesting to me I think for the next two years will be how will the universal multi-cloud network plane and security plane look like? Because I think the last couple of years has been very focused on I'm running in one cloud provider or I'm running everything on-prem and I'm using something like OpenShift or I'm using one flavor of Kubernetes. The reality will be is that as the more traditional enterprises come in, it will get messy. And one cloud provider strategy will probably not even – Even if it's one cloud provider, it's going to be some type of hyper cloud use case where some of it will be cloud-based. Some of it might even be serverless or just cloud provider based in some way. But some of it will be on-prem. Some of it will be metal. How do we connect all of that and how do we provide a security and the visibility layer that treats everything in a way that allows usability and troubleshooting and so on? Because what's very clear is that we don't want to bring all of the complexity of traditional data center networking and all of that protocol complexity. We don't want to bring all of that to this new world, which is fundamentally more simple, where because of the cloud native kind of movement, a lot of the assumptions have been simplified. We don't want to undo that. But at the same time we need to connect to a lot of legacy systems. So that will be a big, big challenge. And whether we call that multi-cloud or something else, it will be about like what is the universal connectivity plane that connects everything together while giving you the advocate security and visibility that you need?

[00:52:22] JM: Thomas, thanks for coming on the show. It's been great talking.

[00:52:25] TG: Thanks a lot, Jeff. Thanks for having me.

[END]