EPISODE 916

[INTRODUCTION]

[00:00:00] JM: Isolation is a fundamental concept in computer science. Software workloads are isolated from each other in order to keep resource access cleanly separated. When programs are properly isolated, it is easier for the programmer to reason about the memory safety of that program. When a program is not properly isolated, it can lead to problems such as security flaws, where one program can access the information of another, and poor isolation can also lead to garbage collection problems, or running out of disk space.

Isolation takes many forms, including individual processes, containers and virtual machines. Techniques for isolation evolve overtime, and a more recent technology that can assist with isolation is WebAssembly, which is an execution format that can run a variety of languages, which compile down into the WebAssembly binary format.

For previous episodes about WebAssembly, you can listen to many of the shows in our archives. Tyler McMullen is the CTO at Fastly, which is a cloud provider that focuses on edge computing systems, such as content delivery networking. Tyler has written and spoken about WebAssembly in detail. He's been on the show previously to talk a bit about WebAssembly, and he joins the show for another episode about computational isolation and how WebAssembly presents new efficiencies for engineers that are looking to isolate their workloads.

We talk about a broad range of topics, including WebAssembly. Full disclosure; Fastly, where Tyler works, as a sponsor of Software Engineering Daily.

[SPONSOR MESSAGE]

[00:01:43] JM: As a company grows, the software infrastructure becomes a large, complex distributed system. Without standardized applications or security policies, it can become difficult to oversee all the vulnerabilities that might exist across all of your physical machines, virtual machines, containers and cloud services.

ExtraHop is a cloud native security company that detects threats across your hybrid infrastructure. ExtraHop has a vulnerability detection running up and down your networking stack from L2 to L7, and it helps you spot, investigate and respond to anomalous behavior using more than 100 machine learning models.

At extrahop.com/cloud, you can learn about how extra hop delivers cloud native network detection and response. ExtraHop will help you find misconfigurations and blind spots in your infrastructure and stay in compliance. Understand your identity and access management payloads to look for credential harvesting and brute force attacks and automate the security settings of your cloud provider integrations. Visit extrahop.com/cloud to find out how ExtraHop can help you secure your enterprise.

Thank you to ExtraHop for being a sponsor of Software Engineering Daily. If you want to check out EextraHop and support the show, go to extrahop.com/cloud.

[INTERVIEW]

[00:03:18] JM: Tyler McMullen, welcome back to Software Engineering Daily .

[00:03:20] TM: Hey! Thank you very much happy. Happy to be here.

[00:03:22] JM: Today we're going to talk about isolation, and WebAssembly isolation specifically. But let's start with just that computer science fundamental term. What does the term isolation mean?

[00:03:34] TM: So I guess the way that I think about this is that isolation can actually mean a whole bunch of different things. When you break down like the different parts of computer science that actually like breaks out even more different variations for what it can mean. So if you've seen a couple talks I've given about this, for instance, in like control theory. There's a concept of isolation. That's obviously not what we typically mean when we're talking about it as it refers to, say, like web applications and so on.

So, to me, it kind of like fundamentally breakdown into like two primary different ones. One of them is resource isolation. So when we talk about, say, like containers. Typically, what we're talking about is actually this concept of resource isolation. Meaning that you're running some sort of code and the resources it has available to it, whether that'd be memory, or CPU, or disk, or really like access to anything. That part can be controlled and we can know exactly what it's able to access. That's I think the primary thing that people are talking about when they talk about isolation, yeah.

[00:04:38] JM: What are some mechanisms by which we can get isolation in our modern software?

[00:04:43] TM: So like kind of the most fundamental one, the one that people accomplishes this for most people most of the time, and they probably don't even think about, is the entire concept of a process, right? It's so fundamental to the way that we do computing now that like it seems like it's almost just an inbuilt concept, but it's not. It hasn't always been there. This entire concept comes from like the multiprocessing days when we're trying to do timesharing across mainframes and such.

So the concept of a process is essentially a non-cooperative multitasking that operating systems do for you. So what a process effectively is is a region of memory. It's a virtual memory block and some metadata that lives in the kernel that essentially isolates it from other processes and isolate its memory from the processes, the memory of other processes on the system.

That's probably the most fundamental one. Then like as we get a little bit more modern, like from the opposite end of the spectrum, you get virtual machines, right? What is different about these is that they're essentially attempting to isolate a different set of resources, right? So virtual machines take a very different approach to attempting to isolate software, whereas processes are like a kernel level, "I'm keeping this virtual memory separate from this other virtual memory."

Virtual machines are kind of a – They're really trying to make it like so that you are emulating an entirely new computer effectively is what's happening there. Then on like the far other end of the spectrum, you have virtual machines in a different sense. So this is like virtual machine. The first one I'm referring to there is like something like VMware. Then we use virtual machines in a different way, which is when we're talking about things like the JVM, right? The Java Virtual Machine is a different type of virtual machine. So that's attempting to do isolation not necessarily in the form of a process, but in the form of emulating a different type of processor.

It's a really hard thing to explain now that I think about it, as I'm trying to work through how to explain all these different forms of isolation. We use this term in so many different ways to mean so many different things. But, fundamentally, what it really comes down to is that like you have multiple different programs running on some set of hardware and you want to keep them separate from each other. It's effectively what we're talking about.

[00:07:11] JM: One reason we want to keep them separate from each other is that security issues can result from failures in isolation. Why is that?

[00:07:23] TM: Why is that? So I think that comes up from several different places. One of them, like kind of the most obvious one, is that if you're running code that you don't trust, that doesn't come from a trusted source, and you don't have a way to isolate that, to keep it separate from the other things that are running on the machine or even from the system that is running that code itself. That can obviously be like a really large problem. That code can break out of whatever sandbox it's in. If it can break out of whatever sandbox it's in, it can really cause like some serious devastation there.

The other way that this comes up is code that you do trust that has a bug in it. It has like some flaw in it. So even in the case where you do trust the code, like if someone does find a way to penetrate that code, it could actually – If you don't have isolation in place for that system, that could mean that the attacker could actually break into other parts of your system as well. So even in the case where you do trust the code, like isolation is a pretty important concept.

[00:08:28] JM: You've touched on the importance of isolation in the context of multi-tenancy. So multi-tenancy can let us run two workloads on the same host. Maybe we're talking about – I mean, I guess we could say multi-tenancy in the context of two processes running on the same host. But I think generally we're talking about multiple virtual machines or multiple containers. These are multiple things that sort of look like an operating environment running on the same host. What's the relationship between multi-tenancy and isolation?

[00:09:07] TM: Yeah. So I think multi-tenancy is in fact just a form of isolation. In this case, usually what people typically are talking about when you refer to multi-tenancy is it's not just about a process or anything like that. It's really about making a user feel as if an entire system is just them running on it. I think that goes beyond a process. It goes back to like data layers and things like that.

I mean, for instance, like one of the very earliest things that we had to do at Fastly is that we took Varnish. Varnish is a big open source proxy, reverse proxy for HTTP. So the way that Varnish was originally designed was not as a multitenant system. So it was designed such that like it was expected that you are running this thing on your own hardware in your own data center. With that came a bunch of decisions.

So, for instance, it was simple things like only one configuration could be loaded at a time. So that's obviously like you need to have multiple different configurations loaded if you're going to be a multitenant system. But it also comes down to like some different decisions that you make in the software engineering itself. So error handling I think is an interesting aspect of this.

For instance, in like the original version of Varnish, if you had certain types of errors in your configuration, this would result in an assert being thrown inside of the server. So the entire server would crash, which if it's just your server, that might be an acceptable thing to do. However, if you're a multitenant system, then a configuration having an error resulting in the entire server crashing would be like a pretty, pretty big problem, right? Really, multi-tenancy is about making it feel as if everyone is entirely separate, when in fact you are all running on exactly the same hardware and software for that matter.

[00:11:04] JM: Can you describe the isolation properties of virtual machines and containers?

[00:11:12] TM: When we’re talking about virtual machines versus containers, the way I think about this is that they are effectively virtualizing at two different layers of the stack, right? So virtual machine is trying to make it feel as if you haven't like your entirely own set of hardware, right? So you can virtualize a network card. You can virtualize like multiple different processors, like the entire concept of VCP's, for instance, right?

So that's at like hardware layer, is that it's attempting to do this. The way that's implemented ends up being like via a bunch of special like operations that have been added. This is like relatively new development, but like new operations that have been added to modern processors to make this fast.

When we're talking about containers though, that's actually at like the kernel layer of your operating system. So containers are actually effectively just a process. That's like the crazy secret of containers, is that containers are really just a process. So along with that process comes a set of restrictions.

So in the Linux world, a lot of this looks like name spaces, right? So a process can be associated with a namespace, and that namespace can be associated with like certain resources on the system. Whether that's a certain sections of the file system, or along with like hardware layer, things like network cards and so on. So, effectively, what a container is trying to do is make it seem as if you are running in your own operating system, whereas a virtual machine is trying to emulate the fact that you are running on your own hardware entirely.

[00:12:53] JM: What are the shortcomings of these modern isolation techniques for multitenant system? So if we're talking about containers or virtual machines? You work at a cloud provider essentially, Fastly's CDN. What are the ways in which these virtual machines and containers are insufficient for a company that's doing multi-tenancy at scale?

[00:13:19] TM: Yeah. So depending on like the way your company works and the way the cloud provider works, some of the existing technologies for this are actually perfectly sufficient. If what you're trying to do is to give your customers the experience of having their entirely own virtual private server type of set up. A virtual machine actually makes a lot of sense for that.

I think it's when you get down to like the serverless paradigm that things start to break down a little bit, because at that point, it's not just about like, "Oh! I want to isolate my customers from each other." It's also about you want to make your customers able to isolate their users from each other as well, right?

So in the case of Fastly, this ends up manifesting in the form of, “I really would like to make it so that every individual request that comes through our system is isolated from each other. Not just every customer of ours, but every customer of theirs is isolated from each other as well,” which is like a subtle but like impactful difference.

So in our case, what that actually ends up looking like is if we wanted to do this using existing software would look like, “Well, I need to spin off a container for every request that comes into our system. Our systems are doing tens of thousands of requests per second,” and that's just fundamentally not a thing that is possible, right?

We’ve gotten containers down to the point where like you can start them really quickly. Sometimes it's down to like the single digit millisecond level, which is really impressive. Some of the work that Amazon and others have done on this over the last few years has been like really impressive. But when you consider that a request flowing through a Fastly system is typically processed in under 1 millisecond, you can see how that kind of falls apart, right? So that's kind of how we've ended up on trying to find a different way to do this isolation.

[00:15:21] JM: So if I understand the problem correctly, Fastly is a CDN. It needs to have very fast response times. The container spin up time might be in some ways too expensive. It takes too long to spin up a container for an individual request in a way that satisfies your fast name.

[00:15:49] TM: Yeah, exactly. More or less, that's really how it works. That’s the issue that we’re trying to address.

[00:15:55] JM: Okay. So let's go a little bit deeper on that.

[00:15:58] TM: Sure.

[00:15:59] JM: Why are there tradeoffs between speed and isolation?

[00:16:06] TM: That’s a great question. The reason that there is a tradeoff between speed and isolation, it really comes down to the fact that isolation is always going to end up requiring more work, right? There’s a bunch of different ways that you can do this.

In the case of, let's say, a container of some kind. The additional work comes in the form of having to call into the kernel, set up a new process, set up all the requirements for it, and then at that point actually, like once you have all that all set up, it's kind of – In the case of a container, it’s moved most of the work to the beginning, right? So it's added a bunch of processing and like work to like the creation of the process, right?

In the case of a virtual machine, it’s similar, except that then there's also additional overhead just based on virtualizing so much of the hardware. But there are other ways that you can go about this. So in our case, like one of the ways that we’re working on this is to actually add a lot of the isolation work into the compiler. So the code that the compiler produces is fundamentally more amenable to isolation. However, that of course does then eventually end up adding additional overhead to the execution of that code.

So it always ends up being some set of tradeoff, because really like what you're trying to do is to supervise the code that is running. So you can either do that in like a systems-y way in the form of containers and processes and so on, or you can do it in a way that actually involves like instrumenting the native code that the compiler has generated such that it is even more amenable to this. But either one of them adds overhead in one form or another.

[SPONSOR MESSAGE]

[00:17:56] JM: Cruise is a San Francisco based company building a fully electric, self-driving car service. Building self-driving cars is complex involving problems up and down the stack. From hardware to software, from navigation to computer vision. We are at the beginning of the self-driving car industry and Cruise is a leading company in the space.

Join the team at Cruise by going to getcruise.com/careers. That’s G-E-T-C-R-U-I-S-E.com/careers. Cruise is a place where you can build on your existing skills while developing new skills and experiences that are pioneering the future of industry. There are opportunities for backend engineers, frontend developers, machine learning programmers and many more positions.

At Cruise you will be surrounded by talented, driven engineers all while helping make cities safer and cleaner. Apply to work at Cruise by going to getcruise.com/careers. That's getcruise.com/careers.

Thank you to Cruise for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[00:19:17] JM: Just to go a little bit deeper on that idea. Can you talk more about how compilation or interpretability can lead to better isolation, better speed? Why does compilation and interpretation fit into this conversation about isolation?

[00:19:37] TM: Yeah! This is like a really fun part. It's what I spent the last like 2-1/2 years. So, okay. So if you consider like virtual machines and containers and so on, one of the assumptions that they have is that you're going to provide them some native code that is already compiled. You as a container have no way of knowing what's in that code. You have to assume that it could contain any set of operations in any particular order that are trying to do every single thing possible to breakout, right?

So the ways in which you need to be able to provide the isolation there have to be like significantly more heavyweight, right? Because you have no guarantees about what that code is going to try to do. So when you look at like interpretation or like sandbox compilation, we changed that model a little.

So interpretation is kind of the easier example of this. If I write code that interprets some other code, I can guarantee what exactly that code is going to be able to do, because I've written the interpreter. They can only do whatever the interpreter is allowed to do. The compilation side of it is actually really similar.

So if I'm doing the compiling to native code, I get to say like what code I will actually generate. So for instance, if we're worried about a user uploading code that attempts to make some sort of sys call into the kernel, and I don't want to let them do that, let's say. So if I'm in control of the compiler, what I can do is say, "Okay. Well, I'm considering this function to be entirely off-limits.

Great! Now, I just won't generate the code that calls that function, easy enough.” Right? But in the case of, again, like containers or something like that, that code might be attempting to call whatever it wants. So you have to put other sorts of safeguards in place to prevent it from doing so.

[00:21:38] JM: Okay. Great. Let's begin to talk about WebAssembly. How does WebAssembly isolate workloads?

[00:21:45] TM: So there’s a couple primary different ways that WebAssembly isolates workloads. I like to think of them as like one is control flow, so control flow side of things. The other one is the memory side of things. So in WebAssembly, like any memory access is expected to be bounce checked. The amount of memory that you have, an amount of memory that you are allowed to access are very explicit in WebAssembly.

So, essentially, if you're running a bunch of WebAssembly programs and one of them tries to access memory that is out of bounds, according to WebAssembly spec, that is guaranteed to be caught, which takes care of a lot of vulnerabilities that native code systems typically run into. It also potentially adds quite a bit of overhead, but we have some tricks for dealing with that.

The other one is in the form of control flow. So this looks like – This is actually one of the more interesting parts of WebAssembly. So unlike native code or in like the typical way that processes are handled, WebAssembly separates the code from the data entirely. So if you think about the way that a process works, a process has a bunch of virtual memory, and the way that a process starts is that the kernel allocates this in virtual memory. It gives it its space. Then loads the binary code into that virtual memory and then tells the processor to jump to a particular location in there?

So what that ends up meaning is that the code is able to access itself ultimately, right? So like this entire concept ends up leading to a wide variety of different vulnerabilities in modern systems. WebAssembly attempts to sidestep that entirely by making it impossible for the code to refer to itself. So any memory accesses you're making in WebAssembly actually refer to like a different set of memory. It refers to this concept called linear memory, which is considered to be entirely separate from the code, which ends up meaning that like, for instance, jumping – One of the big ways that like – Some of the big ways that native code tends to go wrong is that you end up jumping to arbitrary locations within memory, which ends up letting you do all sorts of shenanigans.

So WebAssembly entirely sidesteps that by basically going like, “No. You can only call functions that are actually functions, and you can only jump to locations that have been explicitly set aside as places that you're allowed to jump to. Likewise, the code cannot read itself and cannot modify itself in any way.” So it does a bunch of things in order to prevents the types of vulnerabilities that we see in like modern native software.

[00:24:23] JM: So you’re talking about code that can read itself or can look at itself. I think that term is called reflection, or at least in the Java world is called reflection. What kinds of issues result from a program being able to observe itself?

[00:24:38] TM: Oh, yeah. That's fair. Yeah. So reflection is part of it. It’s interesting. It’s like a slightly different concept, because reflection, when you're talking about something in like the Java world, it’s referring to like the ability to observe the AST, the syntax of the program that you're executing. Whereas in this case, what we’re talking about is like the actual native code. So we’re talking about like assembly operations, right?

[00:25:06] JM: Right.

[00:25:06] TM: Similar to that one, and can actually lead to similar vulnerabilities as well. So I guess just as like one example, one of the big and like scarier vulnerabilities that like C code, for instance, typically has or can typically be attacked with is something that's referred to as return-oriented programming.

So what this refers to is essentially you set up the stack of the program such that you return to locations that are not actually call sites. This is a weird concept. Essentially, what this lets you do is to say, “I'm actually going to return. When I hit the return of a function, I'm actually going to go back to a place. I'm going to go to a place in the program that I didn't actually call it from.”

So what this means is that I could actually use this to jump to a location in a different function that's in the middle of the function, for instance. If you can set up a chain of these, what it can let you do is essentially like have a bunch of these different widgets that do different small operations, and you can chain them together to accomplish larger attacks against the system.

It's a really hard concept to explain, but all of these sorts of attacks are actually completely mitigated by WebAssembly, because you can't jump to specific locations. You can jump to arbitrary locations within a WebAssembly program. Everything has a very structured control flow to it. There's no concepts of jumps, for instance.

Likewise, things like function pointers, which are also often abused, right? C in native code, a function is really just a location in-memory. In WebAssembly, a function is a very well-understood concept. Functions have an index, and you can't just jump to an arbitrary location and pretend that it's a function. Yeah.

[00:27:00] JM: This way of operating in the WebAssembly runtime – I mean, we did a full show, you and I. The last time we spoke was mostly about the WebAssembly runtime. So in this new WebAssembly runtime, are there any engineering problems that are kind of like unsolved engineering problem? If we want to get these safety guarantees that we're talking about, is there anything that's unsolved in terms of trying to achieve them?

[00:27:34] TM: Yeah. Yeah, for certain. I think probably the biggest unsolved thing that we're still working on right now is making it safe for WebAssembly to interact with the outside world. So, essentially, what I'm describing here is if you want to, say, like run a WebAssembly program either in your browser or on a server somewhere. At the moment, and especially even earlier than now, there wasn't really a standard way to do that.

So what that ended up meaning was that everyone was kind of inventing their own their own way, right. So, for instance, like in the browser, if you wanted to callout and – I don't know, like write something to the console, let's say, right? There wasn't any built-in way to do that in WebAssembly and there wasn't any standard way to do that in WebAssembly, which means that if you're actually having to write all of those imported calls for each individual program, in addition to being a bunch of work, it guarantees it's going to have security flaws in it as well.

So one of the things that we've been working on along with Mozilla and several other companies is been this concept of WASI, the WebAssembly System Interface. So this is kind of the first attempt at creating like a standard set of modules that use a capabilities-based security method in order to kind of make it like a standard interface essentially to let programs interact with the outside world.

Initially, what that ends up looking like is a lot like how do I print something to the console, or the terminal for that matter? How do I listen on the network socket? How do I write to a network socket? How do I make HTTP requests and so on?

But we can kind of imagine a future where we can describe lots more concepts with something like WASI. So maybe that's in the form of how do I interact with a hardware security module, for instance? Those things have flight pretty well-known interfaces. We could actually build a standard interface for them into WASI. Meaning that if you wanted to run some code that interacted with that in the browser, we could let you do that. If you wanted to run that code as well on a server, it could work just the same.

So what we’re trying to do there is kind of wrap up the two concepts of like how do I safely interact with the outside world with – How do I interact with the outside world such that it works across different platforms? Trying to wrap those two things up into a single project.

[00:30:02] JM: Do WebAssembly modules – If you're going to create a WebAssembly module or compile down into WebAssembly, are there tighter constraints around how much memory you're using or where you are laying out this program in-memory? If we’re talking about that relative to containers, are you more constrained in terms of how your program actually gets laid out?

[00:30:29] TM: So a little bit. So WebAssembly at the moment –I think where you're going with this, and I think this is probably the biggest constraint. WebAssembly programs currently only really have the concept of 32-bits of memory. So pointers at 32-bits, and what that ends up meaning is that the maximum amount of memory that you can use for one of these programs is about 4 GB. That's probably the biggest restriction at the moment. Of course, if you're trying to do anything that doing, say, like a computed goto. If you're familiar with like the concept of a computed goto in C or assembly language, that's clearly not going to work in WebAssembly either.

So there are some restrictions, but the restrictions that exist are actually not nearly as bad as one would expect, despite the fact that WebAssembly gives you those concepts, like structured control flow and linear memory and so on. But it turns out that we can, using those, actually still emulate the vast majority of the types of programs that people want to people want to execute.

[00:31:26] JM: Can you just tell me more about how WebAssembly memory management works?

[00:31:32] TM: Sure. So WebAssembly memory is, again, like this concept of linear memory. That is a different thing than virtual memory, because linear memory doesn't have holes in it, which is like kind of a big thing in virtual memory. So if you’re familiar with virtual memory, you have 64-bits, you have a 64-bit like virtual memory space. It's actually 48 bits on most modern hardware, but that aside. Yeah, the 64-bits of space, and that virtual memory, only some of it, only a very small percentage of it at any time is actually mapped to physical memory, right? So that leaves like interesting problems where you can seg faults and so on.

WebAssembly takes a very different approach with this linear memory. So WebAssembly memory starts at zero and grows upward. It only ever grows upward. You can’t have holes in it. The primary reason for that is for like making bounds checking cheaper. But regardless, actually it makes the entire concept a lot more simple as well.

So it starts at zero and it's organized into pages. Pages in WebAssembly are 64 kilobytes large. If you want to grow the memory, there is an explicit grow memory operation that happens in WebAssembly. So you grow that. You can grow up one page at a time. Grow up more. I can go all the way up to about 4 GB.

Interestingly though, for typical programmers, the way that you interact with this is actually no different though than the way that you would interact with memory in any other like language, right? So if you're writing in C for instance, you would still just call malloc, and the WebAssembly runtime that you're using has a way of emulating or not even just emulating, but implementing malloc to use linear memory.

So, for instance, the way that this works inside of lucet along with WASI is that if you want to allocate some memory, you just call malloc. Malloc actually runs inside of the sandbox, and malloc goes, "Okay. Do I have enough memory in order to give this allocation to the user? If I don't, what I will do is call the these special WebAssembly grow memory function." In that case, that grow memory function pops out of the sandbox back to the host, which will then go, "Okay. Well, is this user allowed to use more memory? If they are, then great. Let me Just give them another 64 kilobyte page. If they're not, then it lets us make the decision just say no or terminate the sandbox entirely."

So what's neat about WebAssembly's memory management model is that it gives you very granular control over. It lets you have a lot of control over exactly like what the users are allowed to do.

[SPONSOR MESSAGE]

[00:34:21] JM: Looking for a job is painful, and if you were in software and you have the skillset needed to get a job in technology, it can sometimes seem very strange that it takes so long to find a job that is a good fit for you.

Vettery is an online hiring marketplace to connects highly-qualified workers with top companies. Vettery keeps the quality of workers and companies on the platform high, because Vettery vets both workers and companies. Access is exclusive, and you can apply to find a job through Vttery by going to vettery.com/sedaily. That's V-E-T-T-E-R-Y.com/sedaily.

Once you're accepted to Vettery, you have access to a modern hiring process. You can set preferences for location, experience level, salary requirements and other parameters so that you only get job opportunities that appeal to you. No more of those recruiters sending you blind messages that say they are looking for a Java rock star with 35 years of experience who's willing to relocate to Antarctica. We all know that there is a better way to find a job.

So check out vettery.com/sedaily and get a $300 sign-up bonus if you accept a job through Vettery. Vettery is changing the way people get hired and the way that people hire. So check out vettery.com/sedaily and get a $300 sign-up bonus if you accept a job through Vettery. That's V-E-T-T-E-R-Y.com/sedaily. Thank you to Vettery for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[00:36:10] JM: When you're thinking about this from the perspective of a cloud provider, so in the container world, when I talk to people, cloud providers, and they're talking about scheduling containers on to big clusters of machines, they just think of their cluster of machines as a bunch of places where they can just throw these different containers and then these containers will be subject to the kind of isolation flaws that you mentioned earlier.

In the WebAssembly world, I'm imagining just of a series of linear memory situations. I guess you could do like round robin to get – As your users are scheduling WebAssembly programs, you could have some interesting bin packing questions across your different linear memory segments. But I guess, I'm just wondering. From the point of view of a cloud, provider when you start scheduling WebAssembly workloads on to these machines in a linear fashion, what kind of isolation benefits do you get? If I'm a cloud provider and I schedule my workloads into WebAssembly modules instead of containers, what am I getting out of that?

[00:37:32] TM: Yeah. So, again, I think a lot of what makes this better is the granular nature of it. Again, a container, you're kind of trying to emulate like this whole idea of like having your own operating system that you're running inside of. Again, a container is really just a process, right? So you have like a very clear way to execute that process, right? So you can move a process on to a machine, and that's about it. Then you can do the scheduling within that machine.

WebAssembly gives you a like much, much more granular control over how this works. So one way we might do it is to say like, "All right. Well, I have a bunch of processes running on one machine that are all executing these WebAssembly modules." So within each one of those processes, I might have a set of threats, which are also allocated to executing these WebAssembly modules, right?

So I might have 16 threads in this process. 16 threads in another process, and I know that I need to execute this WebAssembly module. So I could actually farm it off to anyone of these and I could assign it to any individual thread in there.

Likewise, though, given that we control the compilation of these, like the native compilation of these WebAssembly modules, what we can actually do is make it so scheduling is a much less hard problem for us, because we can say like, “Okay. Well, at this point in the program, I can actually inject some code that says, “Hey, if you're still running at this point, maybe we should let someone else run on this thread.” So this is like a cooperative multithreading set up.

So we can actually enforce cooperative multithreading into this where we can essentially go like, “Okay. This WebAssembly module runs for a while and then it's going to do some IO work. So it’s going to pop off of the thread and let someone else use it for a while.” So we get the site very granular, like down to like individual milliseconds and even less ability to control what is running and where it is running and what it has access to.

That's I think probably for me like the biggest difference there. But, ultimately, containers or virtual machines or WebAssembly modules, they're all really trying to accomplish the same thing here. They’re all trying to be scheduled to be as efficient as possible while also keeping them as isolated as possible.

[00:39:47] JM: How fast is the WebAssembly stack improving? The reason I asked that question is because I just remember this blog post I saw from – I think it was from Till Schneidereit and Lin Clarke where they mapped out the state of WebAssembly, and there was like 50 different regions in the WebAssembly stack and they all had like variable states of maturity. I'm just wondering how it's maturing.

[00:40:18] TM: Yeah, it’s a great question. First of all, I love Till and Lin. They’re like a human beings and excellent engineers as well. So first of all, shout out to Till and Lin.

Okay. So regarding like the maturity of the whole WebAssembly ecosystem for that matter. Man! The speed with which it is improving is actually kind of terrifying to me. So we've been running a bunch of benchmarks and such lately against like Lucid and other compilers and runtimes, and like for a lot of the benchmarks, we are within striking distance of native speed, which is for how young Cranelift, and Lucid, and WASM time and all of these projects that are working together. For as young as they are, like that's to me super impressive. I'm really pleased with how well that has gone.

On the other side of things, you have like the languages themselves that are compiling to WebAssembly, and that one is an interesting mix back. We have some that are improving really rapidly and have gotten to like production quality within a shockingly short amount of times. So I think that Rust, and C, and C++ are like some great examples of this.

Rust in particular is like – The WebAssembly code it produces is efficient and it works, which has been pretty awesome to use. But there's still so much work to do there, and so much of it actually comes down to the WebAssembly standard itself and how quickly we can get new features standardize that allow new languages to use it. The big one in the future for this is built-in garbage collection and WebAssembly.

So if you think about, A, the types of languages that people typically want to run inside of a browser, and B, the fact that browsers already have garbage collection built into them. One thing that is intended to be built into what WebAssembly eventually is the concept of like garbage collection primitives. So we could say, "Hey, this is an object that needs to be garbage collected. Okay. This object is now garbage," for instance, as two of the small operations involved with that.

What this would let us do is to make it so not only do we allow much more dynamic languages into WebAssembly, but also they could show up and not have to bring their own garbage collection without them, because each of the WebAssembly runtimes would be required to implement their own garbage collection and make it as efficient as possible. That's I think one thing that like when we get that, I think we'll see so many more languages come on to WebAssembly. I'm really looking forward to that one personally.

[00:42:53] JM: You were talking about the benchmarking process for understanding the state of WebAssembly. Tell me more about that benchmarking. I think you said that the benchmark is how fast WebAssembly code runs relative to 10 native code?

[00:43:09] TM: Yeah. That's how we benchmark ourselves anyway. That's not how everyone does it. If you're running a browser, that might not be the best benchmark for you. You might be compared with.

[00:43:18] JM: By the way, that's like native x86?

[00:43:22] TM: Oh yeah.

[00:43:23] JM: Assembly code?

[00:43:23] TM: Absolutely.

[00:43:23] JM: Okay.

[00:43:24] TM: Yeah. So the way that we typically do that is that we will take a C program or a Rust program or whatever we would like with that and we'll compile it with a normal compiler. Maybe it's Clang. Turn on optimizations and produce native code with it. Then we will take the same code, compile it to WebAssembly and then compile that with Lucid and then run that and just compare them directly.

[00:43:48] JM: Sorry to interrupt you. I should define this. What is Lucid?

[00:43:51] TM: Oh, yeah. Sorry. Lucid is the open source compiler and runtime for WebAssembly that we at Fastly have developed and open sourced earlier this year.

[00:44:00] JM: Okay.

[00:44:01] TM: So we take the native code that we have compiled and we take the code that was compiled to WebAssembly and then compiled with Lucid to native code that way, and then just compare the execution time of each of them. So there's some things that in WebAssembly are kind of fundamentally always going to be a little bit slower, at least until we can find clever ways to optimize them.

One of those is things as simple as like function pointers, which can affect like dynamic dispatch within languages. Those are always going to be a little bit slower because they need to be type checked before we will allow you to call them. Likewise, certain types of memory accesses are always going to be a little bit slower, and a lot of that really comes down to just the safety guarantees that WebAssembly provides for us. I guess we consider those to be worth it, but regardless of those two things, we’re still at this point getting like within spitting distance of like the native code speeds as well, which I think is like a pretty impressive thing. Yeah.

[00:45:02] JM: When you look at those benchmarks, how extrapolable are they? Can you look at the benchmarks and say, “Okay. Based on these benchmarks, the entire web is like almost as fast as a native assembly code,” or do you feel that just is covering some characteristic set of functionality?

[00:45:26] TM: Yeah. I don't think it’s actually like terribly extrapolable. Each of these benchmarks will typically cover like some – They're kind of a worst-case scenarios for specific types of code. So, for instance, we have some benchmarks that just do random memory access and totally random patterns, for instance, or, well, seemingly random patterns, which is not a thing that you would typically see in like a normal program.

Likewise, we have some that just do like tons of floating-point math, which that's not the typical behavior of like a web application or something else, right? But what they do is like they let us exercise each of those individual like kind of little subsystems within it to make sure that each of those are getting faster. Then overtime, I think when we get more and more experience with how people are actually using WebAssembly, we’ll be able to develop more and more actually representative ways of benchmarking all the different types of programs that we’re running.

[00:46:25] JM: We talked about this a little bit last time, but tell me how WebAssembly fits into your work as CTO at Fastly.

[00:46:33] TM: Yeah. So from my perspective, WebAssembly is clearly the right way for edge computing to go in the future. I think we’re still early days with that, especially as like the number of languages is slowly growing. The number of languages that one can use to compile to WebAssembly is slowly growing. But the guarantees that it provides and the ability to run it near native speeds and to use a minimal amount of memory and to provide the safety to be able to provide multitenant support even in the face of tens of thousands of requests per second, there’s nothing else out there that allows us to do that.

So from my perspective, the way that Fastly is clearly going is we've always said that we are an edge cloud of sorts. The initial way that that manifested itself was as an extremely configurable CDN, right? We allowed you to do some small level of edge computing since our very earliest days, but our goal has always been to make it possible to do more and more at the edge. To move more and more of your applications away from some central server somewhere out toward where your users actually are.

So from my perspective, WebAssembly is like really the clear path forward to allowing people to do more and more computation and do more and more complex logic close to their users. Yeah. I just think it's the only way forward at this point.

[00:48:02] JM: What's the hardest engineering problem you've worked on within Fastly in the last year?

[00:48:06] TM: WebAssembly.

[00:48:08] JM: I’m sure WebAssembly is hard. But I bet that's like more like the fun. It’s like the fun kind of hard.

[00:48:15] TM: I can’t take credit for this one myself. This is actually like more importantly some folks on my team. One of the hardest problems we've been working on lately is adding debug support into the Lucid compiler and runtime. So the way that debugging works when you're using like GDB or like an actual debugger is that it uses this thing called dwarf.

Now, dwarf is one of those formats that most programmers don't really ever have to think about. They see, “Okay. Cool. My native code has dwarf symbols built into it. What does that mean?” Well, I had never really had to dig into this before. So I had always assumed it was just – It’s like a source map, right? It just maps these instructions mapped to this line in my source code. So if I pull up the debugger, that's how it you knows what my backtrace should look like. It turns out that’s not the case at all.

The way that dwarf actually works is that it is its own language. It is its own stack-based language that is embedded into your other program to allow you it to figure out how to generate backtraces and how to debug your code. So we've been trying to add support for this to lose it, and it is just – It is nightmarishly complex and so poorly documented. This is one of those formats that was created quite a long time ago, and it's like almost mythical at this point. So I have to give like a huge shout out to our team working on Lucid here at Fastly whose managed to like dig their way through all of these legends and figure out how it actually works under the hood. So that should be rolling out pretty soon. If you’re using Lucid as an open source project, we should have debug support in there pretty soon.

[00:50:02] JM: I was going to ask about that, about how hard it is to debug in WebAssembly these days. Yeah. I know we’re almost out of time. Can you shed any more light on the debugging experience for a WebAssembly developer?

[00:50:17] TM: Yeah. So I guess the short answer is that it's getting better. It's still not great yet, but it's so much better than it was about a year ago. The way that we were debugging our WebAssembly programs that we're compiling back then was basically like pulling up the WebAssembly source and looking at like the native assembly instructions and trying to map them back and forth and figure out what they actually referred to in a higher-level language. It was really rough.

So thanks to the work of some folks at Mozilla, there is now a – Let’s see. I believe it’s a draft standard at this point for adding special debug sections into WebAssembly programs. I believe that a couple browsers can actually interpret those now, and Lucid knows how to interpret those now as well to be able to provide a much more reasonable debugging experience to users. So, ultimately, what we’re trying to do is account for the fact that, ultimately, like it's executing native code. If you want to figure out what that native code looks like in its original source language, you actually have to map that back through the WebAssembly code as well. So trying to create that mapping through WebAssembly has taken some time, but we're getting there.

[00:51:29] JM: Okay. Well, Tyler, thank you for making the time to come on Software Engineering Daily. It's been a real pleasure talking to you.

[00:51:35] TM: Hey, I really appreciate the time. Thanks a lot, Jeff.

[END OF INTERVIEW]

[00:51:47] JM: Software can improve our lives, but the business motivations of software sometimes conflict with user desires and may hurt us instead of helping us. Rehack is a reverse hackathon that uses humane design to tackle this conflict. Rehack is devoted to making today's technology healthier, fairer and more humane with as little business comprises as possible.

How can our software improve our psychological state rather than stressing us out? How can we redesign our software products to make social interactions more productive and enriching instead of introducing feelings of isolation and inauthenticity? Rehack emphasizes humane design, usability and positive mental health.

Rehack is being hosted by Princeton University this November and they're looking for sponsors who are interested in supporting their mission. For $500 to $3,000, your company can support Rehack with resources for the hackathon. Event specifics can be found on recheck.co. That's R-E-H-A-C-K.co. Rehack hopes to use higher level product thinking, UI and UX design and human computer interaction to find ways to improve today's tech products.

I'm a fan of what Rehack is doing and I think it's a great idea for a hackathon. I think it's very positively motivated. If you can sponsor them, they could really use the help. Go to rehack.com to learn more about Rehack and details on how to support them.

Thanks to Rehack for being a friend of the show, and I look forward to seeing the awesome projects that come out of Rehack.

[END]