EPISODE 1417

[INTRODUCTION]

[00:00:00] JM: Snyk is a platform for security that started with open source scanning and has expanded into container security, infrastructure as code and other products. Snyk is a simple product to use, but has hidden complexities that build large data structures to manage and scan code dynamically. In a previous episode, we discussed the core Snyk product. In today's show, we talk about the engineering behind sneak. And CEO, Guy Podjarny, joins the show again to talk through the architecture of Snyk, and how the company has evolved to serve a variety of use cases.

If you want to reach over 250,000 developers monthly, check out Software Engineering Daily sponsorships. You can send us an email, sponsor@softwareengineeringdaily.com to learn more about sponsorship packages. Thanks for listening.

[INTERVIEW]

[00:00:44] JM: Guy, welcome back to the show.

[00:00:47] GP: Thanks for having me.

[00:00:48] JM: I want to talk about product expansion to start with. The Snyk core product of vulnerability scanning expanded to a variety of other areas, code vulnerabilities, and container management, infrastructure as code. And I want to get a sense for how those different categories have been – How you face them from a management perspective and how you've managed to – If you found commonalities as you've moved from vertical to vertical.

[00:01:28] GP: Yeah, for sure. So I think it's important to understand the lens through which we look at the world. So Snyk is a developer first security company, right? Or a developer security platform. We're always starting from the point of saying, "If I'm a software developer, and I am looking to build secure software, what's the right solution for me? What's the right company I want to engage with? What's the right tool in question?"

And so that ethos was always the goal, even when we launched Snyk, which at the time was just called snake, but now it's called Snyk Open Source, the sort of the notion of open source libraries. It was already meant to be a series of products, because the view, and that has changed over time, is that as a developer, you don't really think about every piece of your application entirely in isolation. You are aware your open source libraries are different, are separate to your code, separate to the Docker file. You might be editing separate to the Terraform file or Helm chart you might be editing. But fundamentally, you are building your application.

And so the view or the sort of the thread that remains and that guide us is, if I'm a developer, and I want to build a secure application, what do I need to be successful? And so through that lens, we built Synk Open Source, and that builds a certain set of methodologies around what it means to build a developer-focused security product.

And we can dig into them more, but they include things like the fact that while an auditor's job is defined issues and help you prioritize, the developer's job is to fix issues. And we don't really get paid for maintaining a nice and tidy backlog. You are expected to actually modify and address those issues. The fact that when you tell a developer about a problem, and they zoom out, they don't see risk. The security person might tell them about a vulnerable library and they zoom out. They might look at which other applications might be using this library or which data assets. Or a developer's first question is, "Where am I using this library? And how does it interact with my application?" Provide this application context.

So there's a bunch of these types of learnings. And so what we've done is we've brought them gradually into these different spaces. We actually started with container. And we launched – There's a whole depth-first versus breadth-first type conversation. We built depth-first tier to developers. So we tackled container security from a Docker file container perspective and say, "Okay, if I'm coming developer, I'm now being kind of given the decision power over which operating system will be done. And given the responsibility of ensuring that it's patched, what do I need to succeed?" And we built in our product. And a lot of the hooks were the same, because it's always software. So you want to find out when you're in a VS code, or whatever it is your editor is surrounding and you're editing Docker file. If you're making a security mistake, you

want something to flag it. When you are checking it into Git, you want something as a part of the automated code review that tells you you're about to add a vulnerable library. When you are told about a vulnerability in a container, you want to fix **[inaudible 00:04:33]** that tells you or some ability to say, “How do I fix this issue?” So a lot of those remained.

And then some things are unique, like in containers, the notion of registries became more important and the idea of connecting to Kubernetes and seeing what's deployed. When we've gotten into code, IDE’s grew in importance as compared to maybe what they were for Snyk Open Source components. Infrastructure as code introduces a lot of cloud drift and cloud state. So each of those products kind of had its own core ethos around the developer philosophies around how you build it. It had a fair bit of repetitive technology that we could leverage. And then it had some new aspects that were more unique to its specific purpose. Does that make sense?

[00:05:22] JM: Absolutely. So when you take something like infrastructure as code – Let's talk about infrastructures code specifically. How do you find the frequent misconfigurations or problems that an infrastructure as code deployment might suffer from and keep that vulnerability index updated over time?

[00:05:50] GP: Yeah. So it's a good question. And it's it has a little bit of multiple answers. I'd say – For all of these products, and infrastructure as code as well, there are two layers. The first layer is application intelligence. It's the ability to understand what's going on. And so you need to be able – In the case of infrastructure as code, maybe it's a Terraform file or a cloud formation. You need to be able to parse it, to understand it, to understand the relationship in it, whether it's modules that get included. Whether it's indeed its life journey a little bit when you talk about tracking drift. So all of that we categorized as application intelligence, right?

In open source, it was around building the dependency tree. In code, it's around the program analysis of understanding data flow **[inaudible 00:06:36]** flow through the application. And then on top of that, you have the security intelligence. That manifests differently. We end up having kind of two buckets. In code and in container – Sorry. In Snyk Open Source and in Snyk container, security intelligence is primarily embodied in the vulnerability database. So we have a whole system, kind of back office system, that tries to listen to open source activity, right? Some

repo maintainer. Some open source project maintainer fixes a vulnerability. Says as much in the release notes, and then that's kind of the end of the journey there.

So we try to listen to this ether of GitHub, and we find it as a candidate of, “Hey, someone said vulnerability, or someone said denial of service, or whatever it is.” Gets to analyst. The analyst decides if it's a real thing or not. Like a Snyk employee analyst. If it's a real thing, they curate, they clean it up, they build it in the database. And that's what we now seek out as we inspect dependencies in applications that use them or in customer sites.

In the case of infrastructure as code, and in the case of code, it's more about rules than about a vulnerability database. So it's about which practices do we look for. And there are varieties. And there's like a knowledge base of what's secure and what's not secure. And the platform is the one that needs to know that Kubernetes, by default, runs containers as root. And so that is the default. And therefore, you have to actually have added a line that says, “Run not as root,” to make it secure, or an understanding of the permissions model of some AWS component and saying, “Well, what does this mean?” So there's a curated knowledge base of all these cloud components, and Terraform entities and such that are known. And then there are some rules that say, “Well, if these properties are there, or if these combinations of properties are there, then that's insecure.” And prompt that to the user at the relevant times.

The points in which you prompt it, they become, again, fairly consistent. When you think about it, you build software. You build it in IDS. When you commit code, and you create pull requests, that’s’ the code review process. So you want to flag that. So IDE is really around pre-emptive. Code review is really when the helpful team member comes along and tells you, “Hey, there's a mistake here. You might not want to continue.” You can put guardrails and controls that actually say, “Thou shalt not pass,” in the build process. Maybe you have multiple thresholds of them in the build versus the deploy, or whatever it is. And then you want to track what's been deployed so you see you haven't drifted away, because someone might go to the AWS console and check a box. So we call that drift. And open up. Because you want to just troubleshoot something. You just open a port so you can SSH into a machine. Suddenly, you turn it into something that's insecure. And so even your infrastructure as code script was secure. Your plan might have configured it securely. The cloud deployment is insecure right now. And so we want to flag that and help you know what to do. That's the rough flow.

This security intelligence is an evolution, right? Like just like an application intelligence, security intelligence is never ending. You never know everything. And the application intelligence is stack dependent. So we have to – I mentioned before, we build depth first. Mostly that manifest in saying we support a stack and we really aim to support it well. And then we expand to the other stacks as opposed to kind of skimming the surface and having a shallow understanding of something across many stacks right away.

[00:10:01] JM: There's a focus on speed in a security scanning system. Because if you want to put it in your CI/CD pipeline, for example, then you want to have a really effective – CI/CD can get bottlenecked. And you want it you want it to be fast. I mean, generally for all. Obviously, for all code, you want to be fast. But how has speed and performance been maintained in Snyk? Do you have performance testing? Or are there programming language selection for code scanning and for the vulnerability scanning that you can speak to?

[00:10:42] GP: It's a great question. So performance absolutely matters. I don't know if you know this, that I was CTO at Akamai before. So I was sort of touting performance for a long time. And I know all the business cases, even on the business front of how 100 milliseconds on the webpage can actually affect the conversion for a bit. I think developers are especially sensitive to performance.

When you think about performance for us, we think there's sort of two maybe levels to it, or maybe even three. The first is the design of the solution. So the design of the solution has to be one that is fast. Probably, Snyk code is the best example of that, literally an order of magnitude faster than the alternatives. We acquired an engine over here, a company called Deep Code, which built this incredible thing. And we ruled out like the reason that a company drew us like a lot of it was the speed and accuracy. But a lot of it was really the speed and how that applies. That's not just because of some smart implementation. It's the fundamental algorithms of how the program analysis is done. And so over there, that's significant.

And that allows for a thing, like you mentioned, the build process. But actually, the bigger test is in the IDE. Performance when done right, affects user behavior. You're in your development environment. You hit save. And you want the squiggly line to sort of show up to tell you that

you've made a security mistake. There's very little latency tolerance in that exercise. It has to be fairly immediate.

The build process also can't be slow. But it's not that sensitive. But you just need to be proportional to the amount of tests that are being run. So a lot of it is done at architecture. You need to think about the Delta testing. For instance, Snyk open source is able to test a branch fast enough, that when we tested a pull request, really what we do is we run two tests. We test the branch that you are about to merge to and the new branch you're representing, and we do a diff. And it's because the test is fast enough. And that's the design.

So that bit, I think we've done very well. The second bit, we've hit and miss the actual implementation. So in Snyk, we opted for a Node.JS stack as a whole. We started in the Node.JS world. That was the world that we've initially only supported. Like Snyk's original solution was purely for NPM dependencies as we've grown dramatically since. So we've chosen Node.JS mostly because of the familiarity people with the tool and those surroundings. And when it was light, that was still performant. And we had all sorts of tests that made sure that it stayed performant. As we started using NPM and Node.JS for the purpose of scanning non-node applications and things like that, I gotta say, we strayed off of it. And recently, we've built far more optimized versions of the CLI. And there's more work going on in it. It goes back a little bit to the architecture.

We do monitor, and we have tests that constantly test the performance. I'd even split that up further just from the software geek perspective here, which is, there's the notion of boot time. How fast are you in a small case? How much overhead does your, say, CLI or a client provide? That's the case in which Node.JS is not awesome. If you're scanning a small project, then the choice of language makes a big difference. And we're pretty seriously working on the more native language variant there for speed of boot time and injection in various places.

And then there's the second aspect, which is when you deal with projects that are elaborate, for instance, for dependencies, there's this – What we call them big trees. These applications that might have 1000 different applications. In those cases, that choice of language makes less of a difference. And it's much more algorithmic. It's much more around how did you encode these? And are you holding them in memory? And how well you cache them? So we monitor those.

There's a third type of performance, which we've really made a big dent in over the last year. It's one of scale. If you're about to – Like as the company grows, demonstrating, showing you 30 vulnerabilities on a page takes a certain amount of time. If I need to show you 30,000, that page might not be very usable at various parts of the application. And that's to be frank. We didn't originally have those.

So what we do is we use a speed curve to monitor a bunch of those web interface performance. We do that all the time. We have increasingly elaborate CI tests that assess the sort of the end to end performance as well. And look, like I think in any growing or scaling startup, at the beginning, you don't – Like you try to design for scale, but you don't really – When it's an order of magnitude, I think you're almost always prone to have missed stuff. So I think the way to address it is not so much by expecting to have avoided all the performance problems, but rather being able to quickly respond. So that when a problem occurs, are you on it? Do you know about that problem quickly?

For us, we knew about the problem originally more from customer feedback than necessarily monitors. Once we've kind of rallied to fix it, we've introduced monitors to ensure that we don't regress. And I think there are always ways to sort of improve. But I think, generally, that's the right approach to it, which is don't try to fully anticipate and design for the extreme, unless that's the core competency you're aiming for. But rather, just be responsive and adaptive when those needs occur. I think the distinction here is important, right? Like the design, and the principal, and the core competency is that first one that I described, which is the ability to create a good developer experience, a great developer experience in the IDE and to get the build that has to be fast. And the adaptive part is the sort of scale of, when you're going to support a humungous organization and you aggregate information, how do you work there? Fine. We didn't anticipate all of that, and want to evolve it over time.

[00:16:23] JM: You talked a little bit there about memory management. And I'd like to know more about when a code scan is occurring and/or a vulnerability scan is occurring, you're trying to manage that memory footprint? What does cause significant memory consumption? Is it analyzing the code or holding – Do you have to hold like vulnerabilities in-memory or are you – I

guess, give me a little bit more of a sense of what the runtime actually looks like when you're doing vulnerability scanning.

[00:17:00] GP: Yeah, yeah, for sure. I think, look, I'll give you two examples that are very distinct on one of the dependency side and one of the code side. On the dependency side, it really is around the size of the dependency tree that you're holding in-memory. And a lot of it was around how do you codify all sorts of repetitions. And so dependencies, your dependencies, your dependencies, dependencies. They come back. They use the same dependencies again. It creates all sorts of cyclical routes that you want to manage. Sometimes you can ignore those cyclical routes. Sometimes you actually need to know about them, because you need to apply some – Like try to mimic the duplication logic that the package managers themselves use.

So for instance, when they encounter the same library being required, but in different versions, depending on the language. Some of them allow that and some of them don't. And so the naive interpretation that we originally had didn't quite fuss about that too much. And that worked well enough for the NPM world, in terms of how it held these dependencies in-memory as it expanded the graph. As we got into languages like Scala and Gradle to an extent, Go, where everything is a module, these graphs get pretty humongous. And we needed to create a better data model for it that gets calculated in real time.

And also, at times, you also have to – There are some flags that you can choose for those edge cases. You have to choose to trade off some accuracy for speed. Basically saying, “Well, how nested do you want to get to address those.” And this gets further demonstrated more on the server side when you talk about remediation paths. So when you say, “Okay, you're using A, that uses B, that uses C.” And so it's fine, that's an easy tree to solve. C is vulnerable. Now you need to find the version of B that gets you to a safe version of C. And then the version of a that gets you to a safe version of B. And so the number of paths there starts getting pretty big. So again, kind of algorithmic, you just choose a little bit of how much – That single path that I described right now doesn't sound like a big memory problem. But if you can also get to see through five other journeys through your dependency tree, then suddenly choosing the best thing that you should do right now to fix a vulnerability gets a little bit more tricky.

And so over time, we've had, I think, three iterations of this data model. The first iteration was the naive one that worked pretty well for NPM, for Ruby, for most of Maven. And then we started running into Gradle and Scala and more complex. And we had an evolved version of it that was better, but still kind of had some limits. And then now we have a third version that I think is actually been scaling very well. So that's one example. It's really around the dependency graphs, and how much do you hold them in-memory as you process them?

A totally different one is in code. It's also a graph, but it's pretty fascinating. So the way static analysis players typically work, static analysis engines typically work, is that they traverse the code and they try to create data flow. So they need to do an execution flow and data flow, right? The fundamental of static analysis is you're trying to figure out is data flowing from an untrusted source, like a read from a form field, through the code to a security sensitive sync, like a database call, without being sanitized in the process? Most static analysis rules fall into that pattern. It's called taint analysis or taint flow analysis. It's like it's tainted information flowing through the application, un-sanitized and gets all the way to a secure sync.

Over there, most – As you can imagine, the different paths that you can take through the code, they would explode very, very quickly. You could go through many, many different paths and permutations that data might take. And so typically, static analysis has gone through this sort of pruning process. So it would go through and would analyze in which reverses and would build up a graph in-memory. And then when it hits certain limits, it would prune the branches that are not as relevant or that that you thought was more deemed. And some of them even go as far as putting a memory limit, where if you give it more memory, it would be more accurate in that path. It's also very slow.

The deep code engine takes a big data approach. That's something totally different. It takes the whole code. And it translates that into data log format, a key value pair type model that has a whole bunch of – I'm oversimplifying over here. But conceptually, has a whole bunch of like this variable get copied to this variable, right? Or this function was called with this parameter. Like just these key value pairs. And the representative is this flat data log, sort of like big data style, key value list. And then there's iterative work on top of it using – Increasingly, that generates new nodes that are increasingly smart. And that's a far more scalable approach.

And what it does is it creates summaries of different files. So it would analyze a file. It would create that data log. It would analyze. It would create summaries of different files. It would do that for other files. And then as it analyzes the full set of data, it would generate insights that are cross file, and understand flows increasingly again, and again. The systems can get bigger and bigger and bigger.

And the same model also allows us to do the same type of analysis to the entirety. It's a bit hard to sort of describe the data set. But think about it as like all the code in GitHub and all of its history, and use that as a repository to apply machine learning and identify code patterns. So you can give it something like here are like 10 sources. Then reads from form fields. Here are 10 syncs, like database calls and the likes. Go fetch. Go find me others. Go find the others that seemed the same. If you're a good software engineer, typically you can throw – They can throw you into a piece of code. You don't need to know the API's. You don't really need to even sometimes know the language to be able to determine that, yeah, I can kind of get that this is data that's flowing from this form field into this database, right? And identify that. It's because you see the patterns. So this machine learning engine kind of does that. It encodes this vast amount of code. It runs this machine learning. It identifies these patterns. Codifies them in rules, and then applies those on to that data log format to identify mistakes.

So I think, over there, there's actually brilliants, that a lot of it is about memory management, and is about scale and speed. But as I mentioned before, it's just fundamentally different, and just opens up this world of possibilities and these abilities that we're just not practical in the previous way that it was done.

[00:23:34] JM: And do you, in building all those data structures to manage the code analysis, do you actually hit memory footprint issues? Like memory consumption issues? Or does it remain like manageable? What kind of infrastructure are you having to spin up to manage these scans? And I guess, also, since you mentioned the IDE, I'd be curious here, because the IDE, I would imagine, is more of a local procedure. So I'd love to know more about that.

[00:24:06] GP: Yeah. I think, generally, our products, all in all, are not terribly demanding on the client side, as far as memory goes. It was more speed, even with the dependency tree explosion. Some of those things, it was more around navigating the graphs, but they could get

pretty big now with the different flat modes. So there are edge cases in which you might need a little bit more memory, especially if you're scanning a Java, say a Scala system or something like that for the dependency analysis. The code analysis is actually even less demanding.

On the back office – And definitely, the IDE integration isn't that demanding. Yes, it runs locally. It calculates the graphs locally. And some of the analysis is done locally. Some of it is still done on servers. But the systems on the server side, the ones in real time, are also not terribly beastly. But the big machines are really the ones that do that sort of machine learning setup. So those machines do need to be pretty sizable, because they're holding – No matter how efficient it is, if you're holding this incredible repository of data that is – I don't know the exact number, but I think it's millions of projects, and sort of historical context for them and all of that. That's a lot of memory. And run machine learning on it. So you run some pretty heavy computations on them. So they all need to be held in-memory. So over there, we do use some of the cloud stop machines to consume it. But on the client side, we're not that memory intensive.

[00:25:31] JM: What's the biggest engineering problem that you've had to solve in the last year.

[00:25:36] GP: So probably making our system extensible. And so Snyk has been built up in a micro service environment, but with a few services that weren't that micro, and grew up with an API-first analogy. But some aspects of the API were not as aligned maybe with some others or some were more thought through than others. And around the end of 2020, we really kind of made it a product goal to make the platform extensible. The platform was API enabled. So you could invoke most actions in Snyk through the API, and you can consume data from Snyk. But it was very hard to interject into the middle of something. If you wanted to integrate Snyk into some, platform that we didn't support, then you sort of were on your own. You could do it, but you had to do everything yourself. You couldn't just also integrate bringing in the information from that system, but then still work within the Snyk workflows, similarly around authentication, mechanisms, and all sorts of hooks.

And we knew that, over time, we want the ecosystem to be able to integrate themselves into Snyk. Customers had all sorts of custom setups that they wanted to plug in. And we see Snyk as a platform that provides this sort of depth of security and application analysis. And we want people to build on it. And we wanted to make it extensible. And we internally had the need for

certain amount of speed. As we had more teams, we needed to become more platform for our own needs as well.

And so we actually got a bit of a inorganic help over here. First, there was a few steps of this journey. First, we knew about this as a technical need internally. But I think defining this product vision of Synk as a platform helped us kind of rally around it and invest in it. And that happened, as I said, towards the end of 2020. And then we actually hired this amazing team from a company called Manifold, who have previously built **[inaudible 00:27:45]**. They built a platform as a service. They were in Heroku. They had also great backgrounds. An incredible team that's quite lead leading a lot right now at Snyk. They infused this sort of wealth of distributed system engineering and platform. They were building a marketplace as a service product. So they introduced this sort of a very, very deep knowledge overhead. And then also helped us – With their help and with focus on the existing team, we hired more and kind of devoted more energy to defining an API V3. Sort of like a new version of the API into evolving our deployment methodologies to what we've called predictable Snyk. It's still a work in progress. To make more and more, and the entirety of the infrastructure be repeatable and automated.

Where, again, much of it was, but enough of it wasn't, that it wasn't easy. And then extracts all sorts of capabilities into the mesh that is in between. So this exercise included, in parts, creating a mesh networks. We use glue in there to put all sorts of logic in the edges. So the different services don't have to handle it. Creating more repeatability around the infrastructure. And normalizing and streamlining the API, which is a big effort, because it includes people, and a lot of different functionality, and streamlining it, and aligning it, and choosing all sorts of platforms in the process. And then going through and working with the different teams to actually apply this.

So we've made huge strides. We launched the Snyk Apps platform, which is like GitHub Apps or Slack apps. It's like a webhook-based integration layer and authorization layer in Q4. And there's still pieces of it that are not done. But I think we have like this great foundation. It's probably the biggest engineering kind of a big architecture challenge that we had in the company this year.

[00:29:37] JM: You mentioned Gglue. Is that the solo IO proxy system?

[00:29:42] GP: Yeah. We use their API gateway. I'm probably not the best person to sort of dive into the details of how exactly it's implemented. But we have the capable team. They've done the assessment of it. They've played with it. Worked with them very closely. And we're quite happy with what we're running.

[00:29:59] JM: And just to be clear, that was for networking for what part of the application stack?

[00:30:04] GP: Well, a little bit sort of simplified here. But it acts as a mix of a service mesh and an API gateway. So it allows us to – As I mentioned, we created this sort of API V3. So it allowed us to create some connectivity. Almost acts as like an API middleware. Run some changes and things like that in the nodes. So the different services can move to the new realm, if you will, or the new authorization models without necessarily needing to write as much code. And then the second thing it allows us to do is indeed introduce all sorts of authorization checks and such in between the services. So, again, they don't need to be applied in line. I might be slightly misrepresenting over here, because I'm not as close to the implementation. But logically, that's what we set out to do.

[00:30:46] JM: I like to talk a little bit about container security. So if I'm deploying a vast quantity of containers, and some of those containers are being pushed to on a regular basis, and some of them are just sitting there, just as long-lived services that don't really get updates very often. Can you give me an overview for what's the best practices or state of the art for how to decide when and how to scan those containers?

[00:31:25] GP: For sure. Maybe let's talk about the security risk. And then we'll talk about the practices. So the security risks are fairly similar to those of operating system of sort of securing a VM or securing a machine. The difference is that now developers need to do it versus IT. So the security risks are primarily you have vulnerable binaries. Or you have binaries on the system that might have been vulnerable when you deployed it. Or maybe, over time, new vulnerabilities have been discovered in some kernel that you're using or some lib OpenSSL, or something that's a bit less well-known. And you need to know that that has happened, and you need to update them. So we'll come back to the remediation or processes in the SEC. But that's the primary, the most common risk. Unpatched servers just in container form.

And then subsequently, you can have other specific flaws, like misconfiguration of things that are installed or misconfiguration of the operating system itself. Or you can just be overly permissive. So permissions plays a role in the system, which is not necessarily a vulnerability in its own right. It just might be insecure. So if you're running everything as root on this container, then generally any mistake and a security mistake that happens is amplified. Running it as root is not a vulnerability per se. If the system is very secure, nobody will take advantage of it. But if it is, then it's just a bad defense in-depth move to not do it.

At the core, it's really these three things, vulnerable binaries, misconfigurations and permissions are the three security mistakes it can happen. I think the primary difference is really the fact that containers are applied as part of the build system, part of software as opposed to VMs. So in most advanced enough organizations, there's a whole element of server management of IT fleet management that tracks which servers are patched, which ones are updated, and which ones are not. And IT security has evolved through the years to get better and better at it. Still, wrangling a lot of these servers is a problem. And it still accounts for a lot of breaches. But it's not because there aren't solutions. Those solutions are centrally managed. And they are replaced.

Containers, on the other hand, are immutable. There's no point patching them as they're deployed. Even if you did deploy your server management on all of your containers and then you went ahead and you patch them, those containers are immutable. If you're going to restart the system, the original vulnerable container image is going to be the one pulled and running in the first place.

So you have to realize where is the actual core of the problem and fix it there. There are sort of three areas that you need to worry about. I'll start from the right and go left, and then kind of evolve from there. On the right is you can check what is actually deployed. Containers offer great API's, especially if you're on Kubernetes. Then you actually have the opportunity to ask what is on my Kubernetes setup. Snyk does it. Other solutions do it, which is plugged into the Kubernetes container. Find out what sidecar. So there's a variety of means to sort of inquire the query. What is deployed or query the cluster from within. And you find out which container images are in there and secure them, remember. And then you need to test those images to see

if they are secure, which is a scan that Snyk and others allow you to do. It's a scan of an image file and would tell you what's vulnerable inside.

Remember that not every – This is important because it's the actual thing that is deployed. So it would find those examples that you mentioned, for instance, of something you've deployed a while ago and just went stale and grew stale and vulnerable. But also because not everything in your Kubernetes would necessarily come from your registries. You might be using something you pulled in then from a public Docker Hub or things like that, and those might also grow vulnerable. So you have to think about that.

Before that, you can scan your registry, and you can see all the things that are published. Are they vulnerable? Fundamentally, when you find out something that's vulnerable in Kubernetes, you need to update an image, publish it to the registry, and then you're going to pull it from the registry. So the registry is a great place to be constantly on top of at least the latest. So you have to tag your registry images that are actually in production, because there's going to be a lot of old registries. Sometimes you can actually literally deny access to all the delete old images from the registry. But at the very least, you want to tag the ones that are production worthy, or the ones that are about to be deployed. And you can scan them directly from the registry. That's probably the most common way that people scan it, because it's so readily available. Registries have standardized API's. You can just basically point it, give it the right credentials and scan it. And so those are very easy scans. And again, they can be monitored over time, because the same kosher image today will become vulnerable tomorrow, because a new vulnerability is discovered in a binary it uses.

And then we get back to sort of the core, which is really around the Docker file and the creation of it. So containers, unlike VMs, get defined. Oftentimes, in software, they have a base image, and they have the source code, if you will, of the Docker file, and then you run a build. There are sort of two practices to consider here. One, good base image management. The vast majority of content on a container image comes from the base image. And with it, the vast majority of vulnerabilities come from the base image. And so choosing the right base image as slim a base image as you can get. Now take the smallest base image, the scratch base image, the less you have in there, the less chance you have of having vulnerabilities. But also, just constantly updating it. And similarly, a lot of organizations create golden base images that are centrally

managed, and that the different teams just need to make sure that they consume. And so invest in it.

You should also care about the things that might be vulnerable that you're installing in the Docker file. But just statistically speaking, vast majority of your risk comes from your base image, not from the Docker file. And it's important that whatever solution you're using to secure separates tells you if it was in the base image, or if it was in the core, and that it supports base image as well.

The second, and trickier one, is repeatable builds. So you can build Docker files from a specific hash, so that when you run the build, you get the same thing again, and again, and again, and again, and that gives you a reproducibility. And that's great. But it's also a pain to maintain in many cases. And so the reality is that most people use latest tags or use other major tags. And it means that every time you build, you might be pulling in a different version.

I wouldn't go as far as saying that that is good or bad. People have more firm opinions on this than I do. But it's just something to remember. Very, very, very often, when you find out that a image in the registry is vulnerable, and you run a build again without changing any source code, it would fix the problem, because you get the new image. Just, again, you need to understand in the remediation what is the action that is needed. Do you just need to rerun? Do you need to actually modify the base image? Or is the vulnerability in your Docker file. And it's probably in that order of probability if you're using one of the common practices of a latest.

What I would say probably right is, in many applications, I'd say most applications using these non-immutable or these mutable like latest tags and the likes is probably good enough and gives him more convenience. Like it or not, that's what most people do. In security sensitive systems, or in ones in which reproducible builds matter more, then people use specific hashes and chars. And I think ensuring that you maintain your own golden images that you know exactly what's on them and as slim as possible, etc. And securing them is a good way to use those, like use those as the base images.

And then, again, those are the edge cases of it might be vulnerabilities in your own, especially like misconfigurations and things like that. Oftentimes, it might happen in your own profile code.

But it typically just installed fewer things in them. So that's, I guess, a natural container security in a nutshell, or securing your images in a nutshell.

Container security has a whole element of runtime security, which in my view is really endpoint security. Container is just you care about runtime security for your VMs and your containers. And there are a few changes, a few differences. Like a compromised container, you can just reboot and it would come up clean, versus a compromised VM, which if you reboot, it wouldn't. But I think most of the time, you just want the same endpoint security solution for your runtime. Fairly comprehensive answer here, right? Yeah.

[00:39:55] JM: Yeah, great answer. We think about container security, are there also elements of securing Kubernetes or securing the Kubernetes runtime? Like securing if it's running on Google Kubernetes engine or Amazon container engine? Or do you just kind of defer security of those platforms to the platform provider?

[00:40:25] GP: So two levels here. Again, a Kubernetes configuration absolutely allows you to shoot yourself in the foot. Absolutely should be inspected for security mistakes. Tends to not be the application developers, but rather the platform developers that deal with that, the DevOps teams. And I would classify that a little bit less in the container security bucket and more in the infrastructure as code bucket. Because really, you're deploying Kubernetes as infrastructure. There are many security mistakes that you can make, and you absolutely should inspect it, and secure those.

There's a whole world that used to be a thing around sandbox escaping and things like that. Can you break out from the container image itself? From the pods to the host of the container engine? There are edge cases there are things like that. But for the most part, those were a period of time in which container engines were new. And they had a whole bunch of security mistakes that just needed to mature. I think, today, those engines are really quite secure. And that is typically not something you should worry about, especially if you are running on one of the cloud providers managed Kubernetes engine. So you don't really need to worry about someone breaking out of it.

Again, there are sort of edge cases for very sensitive systems. You probably still want to assess it to see what memory is shared and things like that. And sometimes you want specific security policies that ensure that certain types of workloads only run on their own dedicated machines and things like that. But it's definitely low on – Should be low on your priority list as a developer.

[00:41:53] JM: As an example of a cute scenario that a security company might have to deal with. Can you explain what happened during the Log4j vulnerability from the perspective of Snyk? Like what you had to do to adjust or to remedy the Snyk user base?

[00:42:15] GP: To remedy this neat user base around Log4j? or like about –

[00:42:20] JM: Yeah, yea. So how did you have to – From the perspective of the company, how did you have to respond to the Log4j vulnerability?

[00:42:27] GP: Yeah, Log4Shell. So Log4Shell was a pretty amazing point in time. It's probably the most severe combination that I've seen of prevalence of a component. So how widely used it is? Log4j is massively widely used, practically in every JVM-based system. And severity, the vulnerability of Log4Shell is not only is it – Like the fact that it's a remote command execution is bad. It's always bad. But I think what's especially unique about the severity of it is that if you use Log4j for the intended purpose, which is logging user inputs, it's very common. Like you're logging, what has happened? Hey, this input was provided. And logging it. Then you are vulnerable to this remote command execution, the worst type of vulnerability.

And so I think that combination made it truly explosive. And nobody really missed that fact. So for us, as a company, we immediately – So we learned about it very, very immediately, I think within hours of the vulnerability. Kind of having some noise in the air. We've had it in our database. We've rolled it out to our users, our tests. Started accumulating it. We do have a period of time in which a cadence in which the projects get retested. So we did some amount of acceleration of that to try and get people who were already using Log4j to be notified about it sooner. And really worked kind of through the weekend.

This was Thursday night, the vulnerability was disclosed. Or Thursday night, I guess, kind of European time. Or, I think, definitely sort of through Friday, Saturday, Sunday was a ton, a ton, a

ton of work. Log4j is one of those libraries that is used, can be consumed in many ways. And so what we ended up doing is providing – So one obvious way is you might explicitly use Log4j as part of your **[inaudible 00:44:15]** or one of your libraries. So those were the easiest to detect. And people would have already had them in their bombs. Another would be to install it as an operating system dependency. And so we made sure like our container product generally caught that and alerted on it. There were edge cases where it didn't. And so we like quickly added some capabilities to ensure that those scans are there.

And then it can also just be something some binary, some piece of code that you – Or binary that you just put in your system. And actually, when we deal with C, C++ and such, we have what's called unmanaged code detection, which uses partial fingerprint matching to identify those. And we even pack it. So for simplicity, for users, we created a single command called Snyk Log4Shell, which tries to identify the best way possible for people scanning in on their system.

But what has happened really, is over the course of these few days – So most of the time, what customers fuss about is they fuss about, "I've got like a zillion vulnerabilities. I want to know about them. I want to know about them in a timely fashion." But the thing that's top of mind is, which one should I act on?" This was a case in which people wanted to find every single instance of Log4j on their systems. And so a lot of the collaboration with customers was around that.

We also have a partnership with Docker. And so we worked. Had a couple of hiccups at the beginning. And then we kind of worked through those to ensure the Docker and it scans using the Snyk engine identifies Log4j. And so there was a ton of that type of work.

Look, it was amazing to see the security industry rally, I don't think anybody in security slept kind of those few days over it. There was a lot of work. A ton of value provided. The following week saw two primary iterations of people finding flaws in the version that was released. And then again, flaws. None were as severe as the originally disclosed vulnerability. But there were still ones that you had to go to for something one, for something two, sort of do something. So we were on top of those and identifying customers and working with them.

I think it's really interesting. I think it has a lot of repercussions right now. In the open source scene, we're seeing a lot more demand from customers caring about the timeliness of our database. Caring about the real time notifications, which we've sort of always invested in, but nobody cared, I guess. Or like people cared, but they cared about it less. Now suddenly, I think it's appreciated. So a bit of an opportunity to shine.

If I just sort of do a bit of a shout out, I think maybe the piece I loved the most about it from a Snyk perspective is just the startup rally. We're 1000 people now. And so we're like a lot of people. And there was just this great rallying across the customer side and the tech side and DevRel for sort of education and some marketing. There was not a single mention about, “Hey, let's use this to make money,” type opportunity. Everybody was there to say, “How do we help customers? How do we help users? How do we help the open source community get over this?” Not a single complaint about working on the weekend and all of that. It wasn't some boss that demands it. It's people appreciating the severity of the moment, and the gravity and the importance of it. So for me, that made me super proud, to sort of see the team rally.

I'm overhead to these days. I helped where I can. I crafted some messages. But the work is not done by me. But I felt really, really great about it. I think we'll see a lot of repercussions in terms of the severity of it. The White House has convened sort of a session for a variety of vendors at the beginning, which we sort of contributed to think about a week or two ago, mid-January, talking about open source security. We've already been sponsoring the openness of the OpenSSF, the Open Security Source Foundation. And I'm on the board there. And we work on a variety of things. A lot of those were already in the works. But Log4j is really kind of woken a lot of people up to the gravity of the risk involved in this problem of very, very widely used components without proper visibility or investment in ensuring that they're secure.

[00:48:11] JM: Well, congrats on the speedy response. Guy, thanks so much for coming back on the show. I really appreciate it.

[00:48:16] GP: Always fun here. Happy to be on. And thanks for bringing me on.

[END]