Securing Docker Containers: Techniques, Challenges, and Tools

Featuring Sasank Venkata Vishnubhatla and Maxwell Trdina as Interviewed by Tim Chick

Welcome to the SEI Podcast Series, a production of the Carnegie Mellon University Software Engineering Institute. The SEI is a federally funded research and development center sponsored by the U.S. Department of Defense. A transcript of today's podcast is posted on the SEI website at sei.cmu.edu/podcasts.

Tim Chick: Welcome to the SEI Podcast Series. My name is Tim Chick. I am the technical manager of the Applied Systems Group here at the SEI. I am pleased to introduce to you two of my colleagues, Sasank Vishnubhatla and Maxwell Trdina. Welcome. We always start our podcast series [by having you tell] us a little bit about yourselves. We will go alphabetical. Max, we will start with you. Can you tell us why you came to the SEI and some of your interests.

Maxwell Trdina: Yes, I have been at the SEI now for about a year and a half. Before this, I was working more in the manufacturing sphere, doing quality control software. I think what brought me to the SEI is I like the SEI's focus on security as well as DevOps best practices. Not only doing them, but also advancing the state of where they are and also how to make that accessible to other people and how to make it easy for other people to implement themselves.

Tim: Right. I mean, one key that I love about being here is that applied

research aspect. You get to do new things. You get to try different aspects. But then also it is not fundamental research where I am trying to boil the ocean. It is applied research where I work with customers and really get to help them solve their hard problems, which I think is also what led to this blog that we are going to talk about.

Maxwell: Yes, definitely.

Tim: Sasank, can you tell us a little about yourself and what brought you here?

Sasank Vishnubhatla: Yes, so like Max, I am also a software engineer here. A little greener; I came here about just a little over a year ago. I have some experience being a systems engineer and also doing some software development in the government contracting sphere. I was familiar with the SEI prior to actually working here. As a CMU alumnus, it is nice to be back in a very familiar environment. One of the big things that you touched on that SEI does is a lot of applied research, specifically in the cybersecurity field. That has been something I have been doing since I was in high school, doing research in cryptography back then and continuing that in college. Being able to do that here with the work that I work on, some rapid prototyping and MBSE [model-based systems engineering] work that we do, being able to continue that research is something that I am really excited about and really glad to do here at SEI.

Tim: Great. Today, we are going to talk about Docker containers, the vulnerabilities and the issues that are around them. For those people that are earlier in their careers—this is an introduction to these concepts—why would someone use a Docker container? What is its major use case? What are some of the vulnerability concerns?

Maxwell: This is something that I definitely am interested to talk about. I do not know if you had this experience or not, but when I was in college, we didn't talk about containerization. We didn't talk about Docker. It was entirely new to me.

Tim: It was not that long ago.

Maxwell: Yes. It is something that I sort of learned as I went along. Even in my first job out of college and whatnot, we were doing things like running monolithic applications, more or less everything all baked into one big program.

Tim: The whole concept of microservices.

Maxwell: Yes, microservices were not really a concept. That was really something I was not familiar with until somewhat recently. I think that once I actually was exposed to it and learned about it, and then I realized how much of the world actually operates that way and how pervasive it actually is, it was pretty neat. I think it is definitely worthwhile to talk about that. I do not know, maybe someone else has the same experience. But yes, containers are really neat. They allow you to run individual applications in a very isolated, very controlled, very repeatable way that you do not get if you just run them on your system. That gives you a lot of advantages. For example, say you are hosting a website. You have your web server service, and you maybe have some back-end services, like a database and what have you. What else? The list is endless. If you were deploying this the old school way, you would install them all on the same machine. Probably. Maybe not. But they would have all sorts of dependencies that would have to be resolved. Sometimes they might have conflicting dependencies. How do you fix that? Docker solves all those problems by just giving everything its own little area where it can run the things it needs and nothing else and only talk to things as needed over the specialized network it creates. It solves a lot of those older monolithic program problems.

Tim: For people who are old like me, what you experience really traditionally would be someone comes back and says, *It does not work*. Or, *There is this defect, this bug*. And I say, *Well, it works on my machine*, Or, *It works in my staging environment*. You had this disparity between environments. With Docker, the whole imaging construct really eliminates that.

Maxwell: Yes. Not only do you get the benefit of not installing on your machine, but you get an identical environment every time. You can hand them out to anyone, and they will work every time.

Sasank: That is a huge plus, it is easy distributability. You can hand it out to everyone, like you said, and everyone is able to actually say, *Yes, this is the same copy. We all have the same versions of these containers that we are running, and we all should expect the same behavior then*. This type of paradigm is huge for large cloud computing companies for cloud services. You think of your favorite online web-hosting service, your favorite document editors online, all of these services, these are all really just some sort of application running in a Docker container on some sort of server somewhere. That is a lot of isolation away from another type of service that

could interact with it. They can have these interactions, but by isolating them, you are able to monitor them better. You are able to see is everything working properly. You are not mixing lanes and getting potentially confused. This also allows you to, in isolation, track vulnerabilities. You are able to look at a snapshot of your service running and say, *Okay, I want to look at this picture almost and look and see are there any defects. Is there anything wrong in that moment?* That lets you get a vulnerability scan in the moment, which is huge for assessing the security of your system. Containerization, the fact that it isolates away your running services and a lot of things from your actual system, which you the user might be changing, that allows for a lot of replication and replicability across young developers who are able to say, *Ok, I can remove a lot of these potential red herrings in my development and just focus on this I know is to be true, so I can work on it then*.

Tim: There are two things. One is scalability. One cool thing about traditional virtual machines—or even hardware—you have to have dedicated memory, dedicated CPUs. Where this allows you to scale it up and scale that down and actually maximize the use of your CPUs and your hardware, or even minimize your costs in the cloud. You do not have to over-provision your containers. When you are paying per CPU, or you are paying based on the amount of memory you use, there are huge cost savings, even in the cloud environment as well.

Maxwell: Applications like Docker that allow you to spin up these containers and build them from base definitions or recipes like from Dockerfiles, for example. The fact that you can control kind of the data flow and say, *I only want on this Docker network data to flow this fast*, that is a cost-saving methodology that a lot of people and a lot of developers will use to ensure that, like you said, your environment, your platform is scalable in that sense. I think you have some experience actually working a little bit with scalability of this.

Sasank: Yes, I was going to add on to that. You throw in something like Kubernetes into the mix here, and suddenly you give yourself a lot of control over resource management, fault tolerance, and things like that. It is just a little bit outside the scope of our discussion here, but it is just additional icing on top of the cake.

Tim: That is the point, it's an introduction. We hit the economics and then the technical, why it benefits the developer. It makes your life easier. You do not have to focus on everything. You just focus on the resources your application needs to run effectively. From an engineering perspective, it simplifies your

area of concern. But with that said, with the economics being there and the engineering areas of concern being there, the next step is security, which is what our topic is. What have you seen that really has made you question or really become concerned about the security practices of building and sustaining these containers?

Maxwell: Yes. On one hand, Docker as an entire entity, as a piece of software, it can also have its own vulnerabilities. We are not really focused on that aspect of it, but I just figured it would be worth mentioning that this stuff is everywhere, but…

Sasank: Docker is just software like everything else.

Maxwell: Yes, it is just like anything else. Yes. What we are really more focused on is, so you pull an image off of Docker Hub, in my example earlier, you need an nginx server and you need MySQL or something like that, Redis, for example. I do not know, whatever you may need. When you have containers like that, you can generally reasonably assume that they are probably pretty secure right off the bat. It is not a terrible assumption to make. But sometimes you are curious or sometimes, say, maybe you are pulling a container that is maybe a little bit smaller, maybe a little bit less common, a little bit less used, maybe a little bit more specialized for your application. Now, you are more beholden to the decisions that individual software developers have made in the process of not only developing the application that you are using, but also how did they make it into a container image that you can deploy. Both of those things are areas where vulnerabilities can be introduced.

Tim: But even when you take a trusted source, it is really trust but verify. You should not just blindly trust because they say they have this process, and they are providing secure Docker images. Because the reality is, you are the one assuming the risk, as an organization or as a company. Even as an individual, if you spin it up to run your application, it is really buyer beware, right? The burden is on you. It is not on the person who produced that container and made it available to you.

Tim: I am assuming when you say that, you are meaning it scans it and says, *Okay, you are using this library or this application of this software, and based on the versions you are running, there are these Common Vulnerabilities and Exposures (CVEs)*.

Sasank: A CVE, what it tells you is the information about a vulnerability, when

it was found, if there are any bug fixes, the affected versions. It is a great identifier for the vulnerability. What these scans will do is help you identify what vulnerabilities exist in that image, and this is beyond just the code of the software you are using in the image. This is also the information about the operating system that the image provides. Docker images are very similar to virtual machines, but they operate in a much slimmer, sleeker, faster context. The way I like to think about it is your virtual machine almost virtualizes your full computer architecture. Your Docker container, whenever it executes, it does not need to virtualize the entire computer architecture. It is only a small part of it, so it can actually be much more lightweight and, therefore, scalable. There are still parts of the OS that are required, which the container image will provide, and that is where a lot of vulnerabilities can actually be seen. I think you actually can speak to one in particular that is pretty bad, a vulnerability in Docker containers.

Maxwell: Yes, I was just looking into this recently. Apparently, there was a CVE in 2019 where an attacker could overwrite a runC file and, therefore, break out of a container on a machine. That is just one example.

Tim: Basically, break out of the container. And now, they are on the underlying system.

Sasank: Let's just make a little fictitious scenario where this could come in. Let's just say I am providing software to a customer and I have this vulnerability. Some third-party could gain access to my system then through my container, which is exposed to the web. Through that connection, they could be able to gain full privileged access to my host system that is hosting all of my services, that probably has connections to all my databases, that has potential user information, that has all the credentials of the host system which is running these services. That is a huge potential data breach. Locking down not only the Docker software but the container, the actual container software that is running, doing your due diligence on the Dockerfile is going to be huge, and that is part of hardening your image as well.

Tim: One example I think you say in your blog is, manage the privileges of your container. If you are running Linux, do not give them root access.

Sasank: Yes, whenever you are running the Docker container, do not run as root. One of the biggest kind of mantras in the cybersecurity field is the idea of least privilege. If your Docker container does not need to run in root, then do not. It is as simple as that. There is not a need for it to be able to access everything. Do not give it access. Only give it access to what it needs. It is kind

of like a child almost. Whenever you have a baby or you have a small child, you baby-proof your house. Your operating system, your host system, that is your house. These services are the babies in this case. You have got to baby-proof your house, make sure the baby cannot accidentally get a knife and hurt something.

Tim: Max, based on what Sasank says, is there another technique beyond the root admin privilege aspect?

Maxwell: Yes, what Sasank said to specify your service user is a great thing to do because you then can prevent unauthorized access, even if something or someone does break into your container.

Tim: Or break out of their container.

Maxwell: Yes. But alongside of that, there are other things you can do in your Dockerfile that can also prevent other types of exploits. For example—an easy one, and Docker will tell you this on their website, too—is that you do not really want to use the ADD instruction. Anytime you can avoid it, instead just use a COPY instruction. They have very similar functionality in that you could both use them to move file from here to over here in your local build environment. But the ADD instruction, it can do things like download a file from internet. You can just give it a URL and download it. You can also use it to unpack a tarball. If you have some unknown tar archive file that you need unpacked, it will do that for you too. Those are not inherently bad things, but they are certainly exploitable. Say you need to download some sort of artifact from somebody's GitHub page, and say you are getting that from a variable defined somewhere in an environment file. Say that gets changed. Maybe it is someone's automated build process somehow and that gets modified. Now, you can use that as an entry point to insert whatever you like.

Tim: Right. If someone understands about the architecture, they can go, *Okay, well, I cannot attack you directly, but I can attack this thing that you are copying into or you are adding into your system from some maybe not as trusted source.*

Maxwell: Yes, and another thing that is maybe a little bit on the qualitative side is just look at what base images your container is using. Especially in the case where maybe you are adapting an existing Dockerfile, say, *Hey, this project builds a Docker image. I found this GitHub repo that builds a container image based on some open source project that I am using* or something like

that. You want to look at their homework. You want to see what are they doing. What are their base images? If your goal is to harden their product, you say, *Hey, that base image, I have never heard of that before*. Or, I go and see where it comes from and, *Well, maybe it does not really get maintained that frequently* or something like that. You want to make sure that you are substituting that kind of stuff out wherever you can. If your base image is just an operating system, say you are using AlmaLinux or something like that. If they are using something else that is maybe a little bit less common or less supported, then maybe you consider switching that out for something a little bit more mainstream, something that has a security posture, or something that just is more secure.

Tim: We are going to talk about security. We have dived right into some techniques we could use. I want to just take a step back, a little more abstract and say, *Okay, I want to set up a secure container. I am an engineer. I am a developer. This is what I have been asked to do*. What is that process, and why you should you go about doing that? You can start to explain what a Dockerfile, an image, and a container are, what their differences are, and then lead that into the security concerns as we walk through them.

Sasank: Yes, definitely, and I can start us off here. You actually described the order perfectly of Dockerfile into image into container. I like to think about this as almost like as an evolution standpoint. You have your very first source, which is your Dockerfile. This is the recipe. This is grandma's instructions for how you have to cook your image. That cooking is your base image, what you want it to come from, whether it is going to be your batter or whether it is going to be a piece of bread, whatever you want it to be, that is your base OS, the base of it. Then you have all of the instructions that bring in your source code of the software you want to execute inside of it. As Max told us earlier, not using ADD or trying to remove it as much as possible, instead using COPY, which is much more directive in nature. All of that happens in the Dockerfile, which, as I like to think of it, is the recipe for the image. The image is the whole meal, but you are not eating it yet. You are not allowed to do anything with it yet. It is just kind of there.

Tim: It is pretty to look at.

Sasank: This is what you can distribute to everybody. Kind of a funny way of thinking about it, if you know how to make a cheeseburger, this is your cheeseburger wrapped up in a wrapping that you could go sell. This would be the start of McDonald's in essence. Whenever you get your meal, whenever you get your burger and you are about to eat it, you have become

the container now for the meal. That is exactly what a container is. You are actually using it. You are using the service that the meal has provided you, you are eating that wonderful burger that you got. And so, from a Dockerfile, this is what you develop, your source code. Most of developers will be looking at this initially and saying, *This defines to me the step process that my container will be going through and actively running*. The image is the middle ground that says, *Here is the step-by-step process separated into layers of what I am going to do*. You have security scanning software like Grype and Trivy, previously mentioned, that are able to do analysis on all three of these states, the source state, the Dockerfile. Look at it and say, *Your base image has this many vulnerabilities. If you are using this piece of software, understand it has this vulnerability*. Looking at your image, these pieces of software can scan through each layer and help you create a vulnerability digest. This would be what you would create as your baseline for your software from a security standpoint. Whenever you have created your image using Docker, you have built your image, you would get a vulnerability digest of that image, and that is when you would say is, *Here are all the vulnerabilities I have. I am done developing. I have now built something that could be usable. I want to see how usable and vulnerable it is*. In implementation, whenever something is running, you can take a real-time snapshot of the container and do vulnerability analysis of that. That is a repeatable process that you…

Tim: What I am hearing is really, they are stacking this dynamic testing. The container, since it is instantiated, it is actually running, that would be dynamic, right?

Sasank: Correct.

Tim: And the other two would be more static.

Sasank: Yes, exactly.

Tim: The other part is from a cybersecurity perspective to think about for audiences, is that the most knowledge you will have will be a Dockerfile, right?

Sasank: Yes.

Tim: Because it is the most prescriptive. The recipe gives you all the details of what makes it up.

Sasank: It is a ground source of truth, which you want to be able to base all

of your security scans and all of your knowledge from. If you are going to be using an open source container, what you should try to do and what is highly recommended is if you are going to be using a running container, find its image definition or its Dockerfile. Do your homework, study that. Ensure it is not going to have insecure software loaded. It is not using ADD as Max has said. Then when you look at the image, you can look and see if the image is executed in the proper format. Are we giving it a service user, or are we accidentally executing it, giving it root privileges? At each point in this process, you can look and say, there is a security mindfulness task I can look at and say, *Am I doing my due diligence on my Dockerfile? Am I executing my image in the proper conditions?* Whenever I am looking at my running container, my security mindfulness task there is getting the security scan, is actually running a live scan and saying, *How am I looking in real-time?* And feeding that back, all that information back to my development, to my Dockerfile and saying back in the Dockerfile, *In the running environment, I have these problems. Let's address them in our development environment and keep this process iterating so that we can drive down the amount of vulnerabilities that exist in your code base.*

Tim: Max, based on what Sasank said in terms of the overall process, is there any step that you think is key to success in really ensuring the security or gotchas that one should try to avoid?

Maxwell: I think that, on one hand, I also just want to say, I do not want to make it sound like, *Thou shalt not use the ADD instruction. It will never be used ever*. There are definitely good use cases for that. I wanted to say that. I guess to answer your question about a most important step, I do not think that there is necessarily a most important step. I think it is just—and this could dive off into a discussion about just best practices at large, which maybe is a little bit too much—it is just a matter of applying best practices. There are many things that are beneficial from a security perspective that are also beneficial for other reasons, like size. For example, if you are building a container image, if you need to build, if you need to use GCC to make a bunch of code that will run and perform some task for your ultimate end goal, end container, end service, end piece of software, you do not need make [the make command] in your final image. One of the things you can do is you can say*, First, we are going to make this, and we are going to get some result from that perhaps. Then you are going to take that, take some of that result and use it somewhere else…*

Tim: Based on your example, I need to run a C compiler, right? I run my compiler. I copy in my compiler, run it, build my executable, but then I delete

that compiler.

Maxwell: Yes. What you can do is, you do not have to delete it. What you can just do is you can say*, Hey, I have taken the result from my compiler, my compiled code, and you can just use the COPY instruction to copy that. And you can put it in a later build stage.* That way, you can take your built code that will do the task you need it to do without taking any of the dependencies required to actually get it to the point that it is at now, which, if they have no relevance to your end product, they shouldn't be there. Because it is just added size, added space, maybe not that much, but everything adds up. And also, it is just another vector for vulnerability at the end of the day. If you find out that this version of this Python script, this Python function I used has some issue with it or something like that, or is abusable and you do not need it for your end product, then it should not be there. You can use multi-step build process or multi-stage.

Tim: From just a fundamental cybersecurity perspective, you are minimizing your attack surface or the vectors of it.

Maxwell: Yes.

Tim: That is a key to minimize your cyber risk technique. I think you were saying these are just good engineering practices.

Maxwell: Yes.

Tim: That is just one of them, and it applies to Docker containers.

Maxwell: Yes. And from a security perspective as well as just like a best practice for size or something like that. I was sitting here trying to think of how I could bake this into the food analogy. There is definitely something there, but I do not have it. I do not have it at the moment.

Tim: Not everything has to be an analogy.

Sasank: But what you are saying, Max, I think is extraordinarily true of Dockerized solutions or containerized solutions is engineering best practices that normally you would use in your more waterfall products, these products that do not require this microservice architecture, those best practices still apply. Your projects are now microservices, and things are just now at a smaller scale, but you should still do those best practices. That is something that containers allow you to do: you are doing it at a smaller scale now, but it

is still incredibly important.

Tim: What I am hearing is, that Dockerfile image container, I should never just download a container off the internet and use it.

Sasank: Correct.

Tim: Because even if I run some tools, the tools can look into that, [but] that insight is limited compared to the more detailed script and recipe.

Maxwell: This is actually a good segue to talking about what types of things…What are the pitfalls, the risks? Can you do it wrong? From that perspective of should I never pull an image I have not vetted, should I never do that? No. At the end of the day, we all do it. I say I need something. I am going to just pull it from Docker Hub. It would be a bit of an over-application of caution to lock everything down as much as possible. I think it is also important to always talk about how security is a balance between usability and security. If you build a house with no doors in it, you can't break in, but also, you have no doors.

Sasank: You can't get out.

Maxwell: You can't get in or out.

Sasank: It is not really a usable house then. You are absolutely right. You can over-harden something in a sense where your software is no longer usable. Whatever you are trying to do inside of your container, you have locked down the container that nothing can get out and nothing can get in. Your software, in essence, is running in pure isolation. If that is not your end goal, but you accidentally got there, that is okay. You have just made a couple over-hardening steps in your process. I like to think these things through a requirements process. Each iteration you go in your hardening process, you are refining and redefining your hardening requirements of all the CVEs you need to remove. But if you over-define them, if you say, *I need to remove all of these things, I need to make sure it is so secure*, and over-generalize your security, you get a house with no doors. And unfortunately, you cannot use it.

Tim: That really comes down to your risk posture or aperture, right? What is the use case of this application or this container? Based on that use case and the severity of the ramifications, if someone were to break into it or become unstable, someone would be able to manipulate it, or violate it in some way,

what are the implications? What is the risk that I am exposing myself to? You need to balance that. I would expect someone who is building a container, especially in the cyber-physical world that might actually be able to do physical harm to someone, hardening is much more [important]. I am going to lock it down farther, or I am going to do a layered approach and put other controls in place to make sure it is secure that it only does what it is supposed to do, and it can cause no harm.

Sasank: Yes, and those types of solutions actually do exist. Thinking about it from like a watchdog perspective, where you could have a container service that is trusted and verified, that container service, its entire job is to monitor the other running services. You have these different ways of monitoring your attack vectors and saying, *Okay, I have to have a container running. It has to be working. Do I have something that can monitor it?* And say, *Okay, only this network traffic can go in or only this system traffic can go in, or it can only do these certain operations, and it works in a locked down type of manner.* All of these are types of hardening approaches. It just depends on, as you said, your security approach or your risk approach and then also the defined needs of what the hardened container needs to do. If it can be relatively open and just needs to lock down communication, that can be done through the Dockerfile. If it needs to have specific OS-level security things, that is done through the base images and ensuring that whatever you are building your service on top of from a Docker perspective or from an image perspective, that your underlying dependencies are secure. As Max had said, some of them are not insecure. That is okay if you have to use them. Do your due diligence. Work around that. And then also, there is nothing wrong with you giving back to the open source community and working on it yourself and saying, *I am going to help create a hardened version of this use tool*. Let's just say you use a really nice tool that is out there, but it is really insecure. There is nothing wrong with you yourself going and supporting open source development and providing a secured version back, a hardened image of that software running back to the community. You are not only providing value for yourself if you are going to use it, but you are helping others improve their security posture and overall increasing the general security of the population, cybersecurity of the population. That is a net benefit, I believe, that we should be mindful of as well.

Tim: Which is the whole purpose of open source, right?

Sasank: Exactly.

Tim: Yes. The many contributing to the betterment of mankind, right,

because open source, people are just volunteering their time. Right? What is next? We talked about this applied research, things excite you. What can we look for and see in a future podcast?

Maxwell: Yes, I am currently working on a different project where I am working on deploying hardened container images in a Kubernetes cluster. I am actually using them to stand up services and whatnot. It seems like it would perhaps be the logical next step going from making hardened container images just to using them for something. That is something I am working on, and I am sure I will have commentary and things to share about that.

Tim: Yes. Looking forward to it. Sasank?

Sasank: Yes, so looking at the benefits of using hardened Docker images. Some work I am doing is changing the environment of how you use these hardened images. Having them run in an air gap environment, and then also looking at building these hardened images and running these hardened containers in automated settings so that if you have easily mitigatable vulnerabilities that can be automatically mitigated, actually doing that in a CI/CD pipeline or continuous integration, continuous development, or deployment pipeline since that gets more secured software to the end user faster. You are able to get that at an iterative pace instead of at a much longer per release pace. So [I will be] looking at ways to speed up getting hardened images out there for people.

Tim: Sounds very interesting. Thank you for taking the time to speak with us today. For those not in the podcast, in our transcripts, we will provide links to any material that we talked about today. Also remember that podcasts are available in SoundCloud, Spotify, Apple Podcasts, as well as the [SEI's] YouTube channel. Thank you for joining us, and I look forward to a future podcast.

Thanks for joining us, this episode is available where you download podcasts. Including SoundCloud, TuneIn radio, and Apple podcasts. It is also available on the SEI website at sei.cmu.edu/podcasts and the SEI's YouTube channel. This copyrighted work is made available through the Software Engineering Institute, a federally funded research and development center sponsored by the U.S. Department of Defense. For more information about the SEI and this work, please visit www.sei.cmu.edu. As always, if you have any questions, please don't hesitate to e-mail us at info@sei.cmu.edu. Thank you.