Carnegie Mellon University
Software Engineering Institute

SEI Podcasts

Conversations in Artificial Intelligence, Cybersecurity, and Software Engineering

Understanding Container Reproducibility Challenges: Stopping the Next Solar Winds

Featuring Kevin Pitstick and Lihan Zhan with Grace Lewis

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.

Grace Lewis: Welcome to the SEI Podcast Series. My name is Dr. Grace Lewis, and I am a principal researcher at the Software Engineering Institute and the lead for the SEI's Tactical and AI-Enabled Systems Initiative. Containers have become an expected part of software development pipelines. However, although container builds should be identical, container images often differ because of external dependencies therefore, decreasing trust in containers and possibly hiding malware insertion. A recent SEI tool helps developers identify those differences.

Today, I am joined by two of my colleagues, Kevin Pitstick and Lihan Zhan both software engineers in the SEI's Software Solutions Division. We are here to discuss a project that they are involved with that aims to improve container reproducibility. They have created an open-source tool called Vessel that analyzes container builds to identify their differences, spot potential problems, and make containers more trustworthy. Welcome.

Kevin Pitstick: Thank you.

Lihan Zhan: Thank you.

Grace: Before we get into container reproducibility, can you tell us a little bit about yourself and the work that you do here at the SEI. Let's start with Kevin.

Kevin: All right. I am Kevin Pitstick. I have been working in software in defense for about 15 years. The last 10 years, I have been at the SEI. I am actually coming up on my 10-year anniversary here. My expertise and interest areas, they span a lot of things: edge computing, microservice architectures, containerization, related to what we will be talking about today, and also communication protocols. When I first started, it was very heavy in kind of the edge computing, R&D [research and development] prototyping. I also did some projects in the realm of video summarization, AI/ML for the edge. I have done some architecture guidance, design reviews. Then recently, and what we'll be talking about today, I have been working on automated tools to help with best practices for containers.

Grace: That sounds great. What about you, Lihan?

Lihan: My name is Lihan Zhan. I have extensive experience in public software repository mining and containerization. One of the projects that I worked on at the SEI is analyzing the impact of using software design patterns in a large codebase. Another one that I worked on recently was drift monitoring of machine learning models to enhance the assurance of models. Recently I have been working on Vessel, which is an automated tool for increasing container reproducibility.

Grace: That is great. That is a lot of interesting stuff. And as usual, it is always a pleasure to work with both of you. Kevin, let me start with a question for you. What is the problem that your work addresses? What challenges are you seeing in container reproducibility these days.

Kevin: All right. First, I would like to start with a definition of what we call reproducible builds, which is the topic for the project. Basically, the idea with reproducible builds is that when software is built, there is some sort of input. If a build is reproducible, you want to see the same output on the other end. In the context of container images, usually the input is something like the Docker file. There is the build context, all the files in let's say your Git repository, potentially build arguments, or environment variables that get passed in. The problem really lies with that most software that is built today is just not reproducible. There are a variety of factors. We will talk about a few of them: things like timestamps, random numbers, randomly generated UUIDs [universally unique identifiers], external dependencies that are always changing from package managers. Realistically, even though containers are designed to be a self-isolated piece of functionality that you can move from place to place, every time you build it, if you look at the checksum of the container image, it is going to be different. This poses a problem because you build a container, and you can't really verify, when you build it again, if those differences are just trivial noise or if something malicious has made its way into that container.

Grace: Right. Let me try to put your work into context. According to your Research Review presentation, and we will have a link to that presentation in the transcript, the 2020 SolarWinds hack involved malware targeting the software build process. Can you explain how reproducible builds would have prevented this attack?

Kevin: Yes. That was a pretty big deal when that happened. Basically, the gist is that hackers, they breached the SolarWinds internal network. What they did is they got some malware called SUNSPOT installed on the build server that was building one of their products. The whole goal of this malware was to watch for build commands to be sent. When it would see a build command being sent, then it would change some source code files that were being built. That would insert whatever malicious attack into the application as necessary. Then, as soon as that build was done, it would remove those changes from the source code. It really was only visible while the build was actually happening. A lot of times, when people are looking at verifying their build process, you look in and make sure that your source code is very secure. You make sure that changes aren't happening there. It is also very popular to put signatures on your final executable binary to verify that that is not tampered with. But in this particular attack, neither of those things actually would have helped detect this. It really wasn't until they saw some unusual behavior happening from the application after it had already been built and distributed across the world, that they realized that something odd had happened. This is a case where, with a reproducible build, if you imagine a situation where you had two different independent organizations, let's say within SolarWinds, one of them was actually infected with that malware. The other one is isolated, separate, not infected with the malware. If there was a reproducible build, and they built their software two times and then were able to compare them and look for differences, this would have been a big red flag that would have shown up. Yes, that is basically what happened, and what we are hoping to prevent going forward.

Grace: Lihan, you are one of the main developers on the tool that implements the process that you guys created; the tool is called Vessel. When you started working on Vessel, were there other tools that already did what Vessel does? Did you build on those tools? What are things that you had to create because they didn't exist, for example?

Lihan: Sure. When we started working on Vessel, there was no tool for comparing container images. We did some research, and we discovered a reproducible build project called diffoscope. Diffoscope is capable of comparing files, archives, as well as directories, but it is not built for containers specifically. So, we ended up using it as part of our Vessel tool to compare and finalize an image file system. We also developed a customized parser to convert Diffoscope output back to Vessel tool format. And another tool that is worth mentioning is hadolint, which is a linter for Docker files specifically. Right now, we are using it to identify and find reproducible issues before the containers are even built. In our recent research, we discovered a tool called diffoci, which is a newly released tool on GitHub, which is capable of comparing container images. However, that tool is only capable of comparing images at the metadata level as well as the manifest level. Vessel focuses on comparing the container images in the finalized file system level. So, tare in the current tool, which, still makes Vessel one of its kind as far as in the current industry.

Grace: Right. It seems like a lot more comprehensive I should say. Can you walk us through a scenario of how someone would use Vessel? From what I understand you could use Vessel to get a comparison of container builds, and then you could also customize it so that results are relevant to an organization's containers. Is that correct?

Lihan: That is correct. To walk you through the different steps of how a user will typically use Vessel, for instance, a user will try to compare two container images built from the same Docker file. Our dev tool comes with a pre-built set of flags to capture trivial reproducibility issues such as timestamp. Then, after the first try, the user will be able to see a lot of other unknown issues. Most of them are project specifically related. For example, if the user is trying to build a Java application in his Docker file, then he will have a lot of Java specific reproducibility issues. From that point, the user has enough freedom to, say, add their own customized flags to capture those Java-related issues based on file path and also in the file type or regex format. Once they, he does that, because our Vessel tool is very easy to be containerized and embedded into the user pipeline, he can easily embed Vessel into his CI [continuous integration] pipeline. After every container build, he will be able to identify any number of reproducibility issues from there, a very automated process. Also, another thing that I would like to mention is originally the intended use for Vessel was verifying the reproducibility of a Docker file. But it can also be used, to capture the amount of differences between two, releases of a container build. For instance, if you make a small change between your two releases, but you are seeing a large number of reproducibility issues, then you know something went wrong, right? And vice versa, If you make a lot of changes, but you are only seeing a small amount of reproducibility issues, then there will still be time for you to start thinking what went wrong and then allows you to trace back to the development life cycle and identify the issues.

Grace: That is very useful because what it means is that you have designed the tool such that it can fit into your own software development pipeline. That is great. OK, so, given how comprehensive and how thorough the work has been, Kevin, it is not surprising it has taken two years to get to where you are today. Can you break down what your approach has been and your timeline?

Kevin: Sure. Of course, yes. At the very start, after we had proposed the project and got funding, we took a couple of weeks I would say to really delve into the existing work in reproducibility a little bit deeper to try to understand what are these reproducibility issues and failures that would apply to containers. Some of the sources we looked at, there is the reproducible builds project, which we talked about. They have a lot of great tools, diffoscope and a number of other ones as well. There are existing tools out there that do Docker file analysis or Docker file linting. Lihan mentioned, hadolint was one of them. Then there is a whole slew of papers [see here, here, here, and here] that are looking at the Docker file ecosystem, whether on GitHub or elsewhere, and doing analysis of what kinds of just issues in general that they see, how often Docker files build or fail to build. One of the things we saw there is that if you just take all the Docker files off of GitHub and try to build all of them, a large portion of them just fail to build. Then there is of course trust the specification documentation. The Open Container Initiative [OCI] is the container specification that governs how the Docker containers are put together. So, we looked through that as well as the Docker file specification to try to really get a grasp on the lay of the land. Then after that, we started to look at prototyping this diff tool, a very basic version that would allow us to do the comparison of files within the container and also be able to manipulate container images. Pull them, unpack them. Really tear them apart so that you could compare two containers side by side. So, we created the prototype Vessel diff tool. Then after that we started to build some examples of different Docker files that were really doing isolated reproducibility issues. From that initial list of issues and failures, we were seeing, from literature, we said, *OK, we are going to try to write a Docker file that is 100 percent reproducible except for this one single issue, and then we could see how our diff tool performs on it and could account for that.* After that then we said*, OK, now we need to go out and go to GitHub. Let's get the real deal. Let's look at what real container images and Docker files are out in the wild and see what kinds of issues we can see with that*.

This was an interesting process because there are a lot of toy projects. There are a lot of things out there that aren't the best example. We definitely went down the rabbit hole of trying to create a good representative set of open-source Docker files. So, filtering on things like the number of stars that the project has, making sure that the project was last modified in the last year or two, and looking at number of commit hashes. If it is just a single commit, and it was just thrown up there, it is probably not actively being used. We created this set, and also the other cases, in order for us to do our testing. We have to be able to build the container image. We would gather a whole bunch of Docker files and identify repositories. Then we would go through and build and, somewhere [between] 20 and 30 percent of them would actually build. That was sort of representative of some of the other literature we would see.

After that we have, a prototype tool. We have a representative data set. We looked at improving our diff tool identification so now we had a whole bunch of failures, within these builds that we could look at. So, we started to improve the flags that our diff tool was getting out of these, with the reports. Then really from there, it was an iteration process of going back and forth between the data set, evolving our model of reproducibility, adding new fields, and that kind of brings us to today, where our main focus right now is trying to optimize that diff process. One of the things we ran into is the more representative the Docker containers we are analyzing, we would see very long diff times. Part of it is, the more representative container images, they just have so many files and sometimes you get a large number of differences. One thing that diffoscope buys us is it does a very in-depth analysis of any change it sees, but that also can take a while. We are looking at ways to speed up the process or potentially circumvent certain comparisons that might be less relevant, perhaps like create a fast mode.

Grace: Lihan, I mean, after hearing about the process, after hearing about what container reproducibility is and why it is so important, I think the audience is dying to see where they can find this Vessel tool. Where can people find the tool today and how can they download it?

Lihan: Sure. Today you can find the Vessel repo under CMU SEI in GitHub. Releases come with pre-built Docker images with all required dependencies prepackaged, so it is very easy to install and run it and test things out. During the process, if you ever encounter any issues or found a bug, feel free to open the issue on GitHub, and we will get to it as soon as we can.

Grace: Well, that's great because you are going to be getting direct feedback from users. Kevin, let's say that Vessel sees wide adoption because it is a super awesome tool. How would this change things for software developers. Why is container reproducibility so important, especially for organizations like the Department of Defense?

Kevin: I sort of mentioned, there are not really too many tools out there that would help developers with this reproducibility. If you were a software developer today, even if you wanted to make your containers reproducible, you would be kind of in the Wild West of trying to track down all these differences. Our hope for the software developer people is that they have an internal or conceptual model of all the things that they could be, I don't want to say doing wrong, but things that they could be introducing, changes in their container images so they are not reproducible, but then also giving them the tools that they can investigate those issues and potentially fix those. That is what we are hoping to change for the software developers. Now, as far as you know, if you have, your organization is trying to adhere to certain security guidelines. There is at least one framework, SLSA, that, in order to get to level 4 certification for that, you either have to have a reproducible build, or you need to describe why your build is not reproducible. This really gives developers a tool to help achieve those higher levels of certification. Yes, going back to the question of, why is this important? I like to think about it in terms of the conceptual model of zero trust, where there's a lot that zero trust encompasses. But the gist that I like to think about is that you treat every step of the supply chain process as being untrusted. We had talked about the SolarWinds attack earlier, even if your source code you think is trusted, and even if you think the output binary is trusted and can't be tampered with, there is still that build process. And so you need to treat that as something that is untrusted. If you look at the DoD, obviously a very security-focused organization, it is not the only organization. There are a lot of other organizations that do safety critical work. But really, if they are striving for reproducibility and maybe not every project within the DoD needs that full level of reproducibility. But for those who need it, we want it to ease their ability to get there so that they can really secure that supply chain and every step of the supply chain. Hopefully those organizations who maybe haven't thought about it or maybe haven't decided that the level of effort required to do it is worth it, it might ease that barrier for them.

Grace: Great. Wow. OK. That is all great stuff. I can't let you leave without telling me what you guys are working on next. So next time we bring in for a podcast, where are you going to talk about?

Kevin: All right, so, talking on the Vessel tool specifically, we talked a little bit about some of the optimization phase for the diff tool we are working on now. There is another aspect that isn't released yet. Hopefully it will be coming in the future where we are looking at Docker file linting and repair as a way to look at addressing the reproducibility problem more on, let's say, the front end of the build process. So, before the build even happens, there are issues that we can just analyze the Docker file and build context to see what might go wrong. Yes, it is really kind of taking that holistic, multi-pronged approach where we are analyzing before the build process and after the build process. We think that those two could work in concert together very well to get a full view of the reproducibility of the container. I guess one thing I wanted to say as we talk about going forward, a lot of times when you talk about the definition of reproducibility, people focus on bitwise reproducibility, where when you create…In this case it would be if you have one container and you say you do a checksum of everything in the container and you do another one, you would expect those to match. I think that maybe should not always be the goal, and that instead we should be striving more toward, I guess, 100 percent explainability so that we know that differences will occur, such as timestamps. Time is always moving. There have been some approaches to fake the time, so that those timestamps do match, but then that could potentially lead to other problems where you are reporting that your container was built 20, 30 years ago. In actuality, a lot of those timestamps, you would expect those to change. It really comes down to if you can explain every difference within your container to a reasonable degree, then it becomes a task of risk acceptance and you say, *Yes, these are* *close enough, and we have identified every potential issue that, where we are happy moving forward*.

Grace: Well, that's great.

Lihan: On my end, I am also planning to focus on the engineering of the backend of the tool. As Kevin mentioned earlier, I'm going to attempt to make the diff tool faster, potentially add a fast mode to make the tool, more usable in a CI pipeline. Also, we are going to test the effectiveness of our repair process and try to reduce a number of reproducibility issues prior to the build process of containers.

Grace: Sounds great. Thank you, Lihan. Thank you, Kevin, for your time and for being here with us. For our audience, we will include links in the transcripts to resources mentioned during this podcast, including the blog post on this topic. Finally, a reminder to our audience that our podcasts are available everywhere you find podcasts including SoundCloud, Spotify, and Apple Podcasts, and the SEI's YouTube channel. If you like what you see and hear today, give us a thumbs up. Thanks again for joining us.