SEI Webcast

Software Supply Chain Concerns for DevSecOps Programs

by Aaron Reffett and Richard Laughlin

Shane McGraw: Hello, and welcome to today's SEI webcast, software supply chain concerns for DevSecOp programs. My name is Shane McGraw, outreach team lead here at the Software Engineering Institute, and I'd like to thank you for attending. We want to make our discussion as interactive as possible, so we will address questions throughout today's talk and you can submit those questions in the YouTube chat area, and we will get to as many as we can. Our featured speakers today are Aaron Reffett and Richard Laughlin. Aaron is a senior member of the technical staff in the security information systems group within the CERT division here at SEI, where he develops and operates software systems for the analysis of cyber related data in support of DoD DHS programs. Richard is a software engineer also in the security automation team within the CERT division here at the SEI, and since joining the SEI, Richard has helped customers adopt modern software engineering practices, techniques and tools. Now I'd like to turn it over to Aaron Reffett. Aaron, good afternoon. All yours.

Aaron Reffett: Afternoon, Shane. Thank you very much. I'd like to thank you for welcoming us, and for everyone who wants to learn a little bit more about software supply chain risks to their DoD, org, industry DevSecOps programs. Take a second here to make our lawyers happy. So what are we going to talk about now? We're going to talk about who are we and why are Richard and I here talking about this? We'll introduce the software supply chain in a little bit. It is unique as compared to, say, some physical supply chains, traditional supply chains. I want to kind of take a look at some trends and recent incidents that have attacked or affected the software supply chain, to give you all an idea of what could happen in certain cases. And I want to talk a little bit about tail risk. Forewarning, it has a graph, it has math on it. Well, not really math. I feel-- I apologize a little bit for not telling anybody there's going to be a little math. Then I'm going to talk a little bit about the DoD DevSecOps reference architecture, more just to kind of frame the rest of this discussion. And then Richard will go into a little bit of a deep dive into some of the aspects, particular aspects of the software supply chain, as they affect various parts of that DevSecOps architecture. Then we'll wrap it up by giving some time to talk about what can programs do to protect themselves? And there'll be room for discussions and questions at the end, but feel free to drop questions into the YouTube chat, and we will address them as we go. So Shane introduced us. Richard and I are software engineers first. We happen to do security. Our background is architecting, writing, operating systems, either on behalf of our customers or internally within the SEI. So our perspective is from an engineer's perspective, and when we're not yammering on here in YouTube, talking to you guys about software supply chain risk, we are helping programs actually implement DevSecOps for their own use. Talked about them left to right, all aspects of it. So DoD DevSecOps is a relatively new thing. It's been around in industry for a number of years now, but the DoD, recently in the last few years particularly, has really taken it up and is attempting to implement it department-wide. And it's still being refined as we speak right now. The OSD, Office of the Secretary of Defense, has sponsored the DoD DevSecOps reference architecture and the DoD DevSecOps initiative, and version two of a series of

documents just came out, I believe a week or two ago, and came out on the Air Force chief software officer's website. Mr. Chaillan was at OSD before moving to the chief software officer's role in Air Force, where he continues to spearhead DevSecOps for the department. One aspect is, while DevSecOps has seen a lot of use within the department, it really hasn't been significantly used in highly contested environments, and I think that's what we're-- what I really want to get to in this is, what happens in an adversarial setting? What happens in a war-like type setting? And that's where some of the talk about tail risk is going to come in. And to really kind of explore, what are the worst case scenarios that could happen? One other aspect I want to raise is the increased use of open source software DevSecOps in order to move quickly-- in order to move capability to war fighter [ph?], it's changed to lean heavily on work that's already been done, and a lot of work has already been done in the open source space. But it's not always just the open source that a program uses itself. It can be the open source that it inherits from COTS software that it brings into the program. A lot of COTS software is based on open source. COTS service providers rely on a lot of open source software for various parts of their offering, and that opens up a unique aspect of the software supply chain, whereas before, using custom code or using closed source code, adversaries had to work a little bit harder to see what was inside. But with open source technology, that's not the case. They can simply just download it from a website, analyze it on their own, find vulnerabilities and seek to exploit them against programs that are none the wise that that reconnaissance was ongoing. As we get towards the end of this, we're going to talk a little bit about particular weak points within the DevSecOps software supply chain, just look at various aspects and say, okay, if you've analyzed your program for all of these perspectives, we have four areas to look at. What should you be looking for? And what are the worst case outcomes if some of those exploitations happen? Because you really, once you work that into your risk analysis, software supply chain issues are rare. They don't occur as often as other vulnerabilities, just regular vulnerabilities that you would see, day to day vulnerabilities that you might see in software that can get patched and fixed on an ongoing basis. They are a lot more rare, but when they occur, they are very high impact. So what is the software supply chain? The traditional supply chain focuses on physical rather than logical grids. But the software supply chain isn't just the dependencies that go into a particular piece of software, say the software that you use in leverage to your end products. It's everything that is software related that touches and affects your software. It's your dependencies. It's the tooling, particularly, it's the build tools. It's your compilers. It's your code analyzers. It's the code repositories that your code comes from. The orchestrators that you use to actually move your source code through the finished products. You have your operational tools on your day to day operations, your loggers, your monitoring systems. The systems that you rely on to tell you whether your system is functioning properly or not. What happens when an F [ph?] does fail. They're the people, the organizations and the processes that are used. So of course, we like to use tools to be able to automate building of our software and operations of our software, but there's going to be people in place. Their software systems implement processes and they're going to be implemented by organizations. And then in the end, there's the underlying infrastructure and the platforms. By

infrastructure, we're going to focus on cloud service providers, but there are traditional supply chains associated with physical. So you run your own-- you buy servers from a company like a Dell or an HP-- I'm not trying to name names-- but you get your hardware, you get your networking gear from somewhere. You get storage from somewhere. You put it together in your own data center. But you to a cloud service provider, they do all that for you. But then they throw a bunch of stuff on top of it. That's all software. So software-enabled infrastructure it's what's being given to you by cloud service providers. And importantly, an organization inherits the supply chain of all the other participants in its supply chain. So all of your upstream vendors and so forth, you're inheriting their process. So can you really trust those processes? How do you trust these processes? How do you know what they're doing? Those are the sorts of questions you need to ask yourself. And just to throw out here, the average GitHub repository-- I learned this a couple of days ago-- contains 203 open source dependencies. And so these are all direct and transitive dependencies. As an anecdote, the last medium sized system that I wrote out ended up being about 20,000, 30,000 lines of old code. I had about that number of dependencies, between two major languages, Python dependences and PEARL [ph?] dependencies, to give you an idea of how old the system was. So 203 dependencies really isn't outlandish. And at a project that size, that seems about right. Larger projects will have more, smaller projects will have less. I think the key thing is, the software supply chain doesn't just end when you receive the finished code, you've built your software. It is enduring and continuous, and part of doing DevSecOps. Everyone's seen the infinity loop in DevSecOps. It is enduring, it's continuous, and the goal of DevSecOps is near real time. That from the time that a developer commits code, to the time that that code is put into production, the goal of DevSecOps is to reduce stack to the minimum amount of time necessary. So trends in software supply chain attacks. So Atlantic Council [ph?] last year, last July, put out a very good report. I've linked it in this and the slide deck will be posted on the stream at the end. I encourage everyone to go take a look at it. There's a pretty high level look at insecurity in the software supply chain, focusing on the commercial space particularly. But they identify these five trends, and there's a lot of analog between the commercial space and DoD space, as it adopts DevSecOps, as we'll see here in a second. So the first is state actors. So software supply chain attacks tend to be-- they're rare, because they're hard. They're very difficult things to pull off, but state actors have the ability to do so, and when a state actor puts their mind to something, the effects tend to be very pronounced, particularly to the targeted individual, if not globally. The second item is abusing trust. So code signing is one of the big mitigations that are used to-- when a piece of code is published, it gets signed, and so the signature says, okay, I attest that I am publishing this piece of software. That's really all that means. But there is a litany of other items that get assumed along with that, then not only are they attesting to I am publishing thing, but they're attesting that it's good, and that it's free from vulnerabilities, malicious code and so forth. But there's nothing about a signature that actually guarantees that. Really, what you're doing is, you still have to rely on the publisher's standards and criteria. Do you trust them that their process has not been subverted? And we'll see a little bit later some of the attacks, that that's an assumption that shouldn't be made lightly. The third item

here is entitled breaking the chain. So hijacking updates. So I've managed to-- I have a piece of malicious code and I want to deliver it. Old school, I can just start emailing you. I can spear phish, I can do things like that. I can try to get you to click check, whatever. I can at brute force try to get you to take a positive step to install my malicious code. Whereas in these sorts of attacks, a much more sophisticated attack, is I'm just going to slide it into an update chain that you're already utilizing. For instance, automatic updates to operating systems, or pieces of software. If I can get an update into that and your system's configured to automatically take all updates and install all updates, I can deliver my payload to you without you seeing it, and again, we'll see, there's a very high profile attack that basically did this. Fourth item is poisoning the well. Open source software, there's a reason why I raised open source software and why it's so important. What's great about open source is, you have access to a global set of developers. What's bad about open source is, you have access to a global set of developers, and you're not quite sure necessarily who each of those individuals are, or who they're backed by in terms of what their motives are. Are they truly altruistic? Are they just trying to make good contributions? Or are they attempting to subvert the software? And since open source can just be downloaded by anybody, this is both a good thing and a bad thing. It's possible to analyze the software, identify a weak point or identify a point where a weakness can be put in, and then find a way to put to that in. And then, once you get it in that far to the left of the supply chain, it's just going to flow to the right very quickly and get adopted by a large number of users. The fifth item here is related to app stores, downloading trouble. So at first I thought, well, I don't know if this has an analog to my use case, because it's really targeted towards Apple app store, Google Play and so forth. But look at it from the perspective of DoD wants to create centralized repositories of software that are built for and trusted by the DoD. So they're effectively really building an app store, and that app store's going to be a gold star target, potentially for our adversaries, in order to slide either malicious payloads or whatnot, try to subvert those projects, because they know that DoD programs are using that code. So some recent examples of some supply chain incidents. I basically just did a Google search and there's some that are well publicized. Most of these are well publicized over the last five or six years. No real particular order, but I really wanted to show that there are attacks that have been made against all different types of the supply chain. I don't just want to show software vulnerabilities, or I don't want to show structure vulnerabilities. I kind of want to show, here's a wide variety of types. So AWS, about every year or so, has a well-publicized event. In 2017, S3 went down and took down a lot of web services in the US east region. US-east-1 is the most widely used region in the US. This goes down, it affects the internet. So this crashed the internet, so to speak, and it was a simple update. An operator at Amazon ran the wrong script, actually, ran the right script with the wrong influence. Took down more nodes than they intended and then it just cascaded. It was just simply a mistake. However, all he users of S3, all the users of AWS suffered the consequences of it through no fault of their own. Of course, with a lot of these issues, Amazon tells you, or Microsoft will tell you, or Google will tell you-- I don't mean to pick on Amazon here-- it's just that they're very transparent about their incidents, which is a good thing. But architecturally, you

could architect your way around some of these issues, but as we'll see a little bit later, it may not be possible to completely architect out of all of these issues. 2019, there was a power failure, and this actually resulted in data loss. It was EVS volumes went down, a less elastic block system or block storage. There were sorts of devices that aren't meant to be powered off and just be like, if you pulled the power off at the hard drive midcycle, you were probably going to lose data and that's what happened. And so this affected the systems that didn't have good backup plans, suffered data loss. 2020, or last year, there's a kinesis outage, again in US-east-1 that disrupted-- it wasn't just the users of Kinesis. One, these systems are highly integrated and it collapsed, and one subsystem will affect others. Also AWS, this was more of a global thing. So while those were attacking AWS-- not really attacking. They happened to AWS. None of those really should actually be attacks. They're just things that happened, and I think that's a good point to also look at. It's not-- I went for supply incidents, not attacks. Things can happen just because bad things happen randomly, and not necessarily because someone is attempting to make them happen. But as we'll show, that when motivated individuals want to make something happen, black swans tend to show up in multiples. So there is an AWS Route 53 BGP hijack. I don't remember the year on this. It was recent, last couple of years. This was to steal cryptocurrency. It was just a criminal act. But BGP hijacks aren't a new thing. They happen a lot of time just because of misconfiguration, but they are a big factor for, how do I reroute large swaths of the internet? And by hijacking BGP-- border gateway protocol, for those who are unaware of it-- that it's the core routing protocol of the internet, you can-- by hijacking BGP, you can reroute the entire internet or large chunks of it, globally or regionally. And this allows you to move traffic from one location through your location, so you can steal information from it or you could black hole it, denial of service type attacks. So this shows that this sort of attack is possible even against large providers of service. Third instance, here's the SolarWinds. I'd be remiss not to raise SolarWinds. Two things really happened here. One, the attack was against SolarWinds's build process. The build process was attacked before code signing, going back to tying this together, we always talk about earlier, is the malicious code was injected in compiler time. So there's a time between when the build server downloaded the source code, and the time that it was compiled. The process was stopped. The malicious code injected, or the malicious code running the process injected the code into SolarWinds's code, then let the compiler continue and then the rest of the process picked it up, so that the code got signed at the end. It was so unique that the malicious code never showed up in the code repository, so if you were just scanning the code repository, you weren't seeing code changes. And if you were just checking signatures for updates, again, you wouldn't see it, because the signature was-- the malicious codes were signed, unknowingly signed. And the second half of this is, these packages were delivered to customers via the update system. So the actor responsible for SolarWinds leveraged two of the trends, where they attacked a build system and got code in prior to code signing, and then hijacked-- effectively hijacked the delivery mechanism, the update mechanism, to have this code flow through to the users of SolarWinds's system, who happened to be operators of systems. So the attack was against SolarWinds's supply chain at development time. The victims' supply

chain was in their operational systems they used to operate the day to day systems. Last look, Pulse Secure had a preauthorization remote code execution vulnerability. This happens a lot. Microsoft has had issues in their net mess, secure message block and RDP systems. This is a normal vulnerability. This is the sort of thing that happens in really any piece of software that you might inherit, or any software that you build. There's a vulnerability in it. It gets discovered, it gets exploited. In this case, it's a VPN gateway, a very critical piece of software. I put this in here because these are the sorts of the things that people look at when we're thinking supply chain, but I just want to show there's really just one type of incident. But it is considered supply chain attack, even if it's just a vulnerability. So some software related ones, typo squatting and the Python repository. This is a fun one. Richard found this one. This is really simple. There are well known libraries that almost everybody uses, but there's really nothing in some of these repositories that prevents someone from publishing an artefact with a typo, like urllib instead of urllib3. Urllib3 is the real one. Urllib is the bad one. If a developer's saying, "I need urllib in there. Just type it in. I need--" they add it to their dependency-- they just add the wrong one. They are unknowingly downloading the wrong thing. There's really nothing there to tell them that they're not, because they are downloading the thing that they're asking for. They're just asking for the wrong thing. Two attacks against-- Node. Well, not attacks. Two instances that happened in Node systems. One that was an MP issue and one just at the Node system. Event-stream was a Node library. What happened was, a new developer-- again, this is open source software-- new developer comes in, puts an unused dependency in for another library that didn't end up getting used. So probably the new developer said, "Well, I'm thinking about maybe migrating a part of this to use this other library." That never happened. The dependency never got removed. That either got noticed, or unrelated, a vulnerability that put in that dependency and through transitive means, that now malicious or vulnerable flatmap-steam dependency got passed along to all the downstream users of the event-stream library. Left-pad also in Node library. This has happened when there was a snafu, an issue between the publisher of this particular library, the developer of this library, and the package manager. And what the developer did, they just removed their code from the package manager. Well, left-pad is 22 lines of code that was used transitively in thousands of projects, including the React web toolkit. It broke all of those builds and it broke all the deployments of users of these systems that were deploying their system live from NPM. So yeah, a 22 line project for a few hours pretty much broke Nodes, globally, away from the Node space. And the last issue, the last one here is ShadowHammer. This is an update system attack. So ASUS computers-- ASUS is a third party system integrator. They have an update system for their software and lo and behold, it got hijacked. And so malicious updates were sent down this pipe, or published at the ASUS and got sent down to users of ASUS's computers and they were downloading that assault. So those are some incidents. There's a question from the audience. Derek Rowe [ph?]. Examples of state actors who've done software supply chain attacks. I'm not here to attribute. I'm not an intel specialist, so we're not going to attribute blame or anything to some of these attacks. There is speculation about the attacks that I talked about, who may have perpetrated them, but I would leave answering that question up to the intel specialists who are

actually in law enforcement, whose job it is to figure that out. But if you would like, there's plenty of writeups about these. That information is available. So this is the part where I warned about math. So the tail risk. Why do I raise tail risk? Tail risk is a financial term and you'll see in the graph here, there's always tail and distributions of probabilities. And in a normal distribution, a bell curve, in a normal distribution, the most likely set of outcomes is around the mean, and then the likelihood of an event occurring decreases at some rate of distribution, the further you get from the mean. And a lot of risk management assumes a certain distribution. So you assume that, I think my risk is normally distributed. And so you think the bulk of your risk is kind of around that mean, so the average type of incident, what that average is. And the further you go out, there's less potential events. And so risk managers tend to cut off at a certain number-- at a certain point, at a certain number of standard deviations away and say, "Okay, I'm only going to consider three or four standard deviations." And what they do so is, we don't have the resources to account for everything, and so these low likelihood events, we're going to try to maybe put a blanket mitigation in place or something, but we're not going to directly attempt to deal with them. And that's all fine and dandy if the actual distribution of risk is as modeled. I hypothesize, and others hypothesize that the risk curve is more akin to a fat tail, and by fat tail, we mean that little thin, long tail at the end of the distributions are actually fatter than what you think they are. So the likelihood of those events occurring is greater than you're giving them credit for. So if you don't account for them, like low risk, or low supply chain attacks. If you attempt to maybe just mitigate them with a blanket, you might be missing some particular attacks that are much more likely to occur than they don't. And I think this is particularly important for DoD systems, mostly because DoD systems are very high value targets. And they're run by a state actor, the United States government. And the United States government's adversaries are other state actors, who have large resources to be able to bring to bear to attack these. So the sorts of attacks that are more likely are a supply chain type attack, and the types of actors that have the ability to successfully undertake one are one and the same. So I do believe that the likelihood of a bad supply chain event happening to a DoD system, or DoD program is greater than your typical commercial system. And I say, for the last thing, simple reasons. Don't assume black swans are random as opposed to orchestrated. Black swans are, by definition, unpredictable, but they are not random. Be prepared to face a black-- a flock of black swans, common with adversarial situations, particularly in a wartime-like setting, when a state actor is likely to pull all the stops out. That's the time when you're most likely to see these sorts of incidents.

Shane McGraw: Aaron, we have another question from the audience. We can actually talk about it, because it's actually an interesting topic, which is, should we be maintaining a private repository, like a copy of Py-Pi or something, in order to prevent some of these types of attacks? I'll let you go first and then I'll give my thoughts on this question too.

Aaron Reffett: That's a really good question. So absolutely. If you maintain your own repository, there's a couple of good things that you get from this. One, if a package disappears, or you depend on a particular version of a package, you will always have it. And you can watch your repository for changes to it. So you're not always constantly re-downloading it from the internet and 99 times out of 100, you get the same thing back, but that 100th time is, now I get a subverted version in. I download it once, I put it in my repository and I make sure it doesn't change. That's good. And it also allows you to vet packages, so yeah, that is a good-- that's a great place to start in order to start to reduce the number of these sorts of attacks. It definitely would have protected you against left-hand, absolutely. It would not have protected you against event-stream though, unless you actually were watching the code that goes into it closely, and you would have seen these changes and flagged them as anonymous.

Shane McGraw: I like what you're saying. I think it's a good approach. One thing to call out though, is that if you get into the business of maintaining your own copies of things, you have to make sure that you're committed to watching what's going on upstream, and that you're pulling in updates regularly, especially security updates, right? And there's this constant pull between, I want to have a curated set, versus, I want to be latest and greatest. And that's essentially the challenge with this kind of approach and why, especially if you have lots and lots of dependencies, it maybe for some projects become intractable. If you look at React, for example, a simple React project may have 10,000 JavaScript dependencies pulled in transitively. There's no program I'm aware of that can really support, you know, keeping an eye on 10,000 different packages, and so different ecosystems have different challenges. I think if you really need to be sure that the code you're looking at is good, you'd maybe stop doing it, unfortunately.

Aaron Reffett: Absolutely. A comment on the event-stream attack, Daniel Gonzalez [ph?] said, "The malicious actor from the event-stream attack was not exactly, quote, new, as they were given maintainer privileges by the repo owner." Good point. And I missed that "new"-- I wasn't implying that they are absolutely, completely, brand spanking new. But it does raise the issue that I was mentioning earlier about what happens with the ebb and flow of people who come into open source projects. What sort of vetting of maintainers versus owners versus developers? Who has the ability to commit code without review? Who has the ability to do whatnot can change, and it can change rather quickly. Projects can change ownership, they can change sponsorship, they can go dormant. These are all things that can adversely affect open source projects. So moving forward to the DoD reference architecture, I have a couple of slides here and then I'll hand it off to Richard to talk a little bit more deeply about this. So we mentioned that the DoD DevSecOps reference architecture and there's the DoD DevSecOps initiative. The initiative is building, or attempting to build, a reference architecture. And so for the purposes of this talk, we're breaking it down into kind of four aspects that the software supply chain touches. And so we have common infrastructure at the bottom. And so this is, mostly for this talk, cloud infrastructure, but it

could be physical, it could be hybrid. It could be a litany of things. But we're focused on software-enabled infrastructure at AWS, Azure, or _____. On top of that, we have common platforms. So a big one is container orchestration through Kubernetes. This could also be really any sort of other platform that you might build an application on. There are many from high level to low level platforms that you can just modify or add something to and push your application out, like WordPress is a platform for publishing content-based systems. That's an example of a platform. A lot of times, these are also one on top of another. And so that's a critical part of your supply chain. The third here is your software factory. This is typically what most people think of when focusing in on your software supply chain, and it's one particular aspect of it here, the 3 in black, the part where the public image repository is fetched [ph?] and those are upstream dependencies. But as you can see here, there's a lot more going on as part of this. So the software factory needs to be considered as a larger entity than just upstream dependencies that go into code. And then the most important part is for the mission system, them being built. Once the software has been built, the supply chain doesn't come to an end. You don't just take the thing being delivered, say, "Okay, great, I don't now really have to consider what's going on anymore," because now when you're operating this in a DevSecOps world, this is being operated on an ongoing fashion, utilizing tooling to assist the operators in doing so, and then those systems also have their own upgrade cycles. My mission system might rely on other web services, for instance, that are needed for it to function. And those are being acquired. That is part of your supply chain. If I reach a high AP item on your system, what happens when they go down? What happens if they're attacked and then bringing you back bad data, malicious data or so forth? So the next slide here is just a little deeper dive down into the software factory, really just to amplify the fact that there's a lot going on inside of just this one aspect, and I don't want to imply that this is just what goes on inside of it, or that the other parts aren't just as complex. I would say that infrastructure, cloud infrastructure, is just as complex, if not more so. But deep dives in any one of these areas could be a series of talks all on their own. But within just the software factory, you've got your security tools. Each of these you're relying on to give you good data back. You're relying on a software code analyzer to tell you, or a static code analyzer that tells you whether there's vulnerability in a piece of code or not. You're relying on a compiler to generate good code and not necessarily bad code and so forth. You have the orchestrators for your process. You have various tooling that is used just to enable all of these things, and I've mentioned the operational tooling on the operation side, logging, monitoring and so forth. These are all pieces of software with their own components, their own update cycles that are being relied upon to insure that your system works as intended. And all of this is getting deployed in the various environments. There's test integration, preproduction, production environments, so you're not just doing it once in production, you're doing it three other times, throughout the entire test and development cycle. There's a lot going on. And so with that, I will hand control over to Richard.

Richard Laughlin: Yep, I'm ready to receive it.

Aaron Reffett: Also a good time to take a look at...

Richard Laughlin: Yeah, there's some good discussion going on about--

Aaron Reffett: Good discussion that's going on. This is a good--

Richard Laughlin: Pinning versions versus, you know, taking whatever is upstream at the particular time, and some other problems with that.

Aaron Reffett: Yeah, question. Should we be pinning dependency versions for everything? Well, that would prevent you from taking updates and potentially taking malicious updates, but then also prevents you from taking security fixes. And so that's a-- there's no one size fits all for that. Whether you pin dependencies or not is-- has to be a positive choice made by your organization for good reasons, and the risks and benefits need to be weighed individually for those. And I would say that it's not a one size fits all. You might want to pin some but not others, and it really depends on what you feel are your biggest risks, to either taking updates or not taking them.

Richard Laughlin: Yeah, and I echo that as well. I think one of the biggest issues is the fact that these kinds of decision of, do I pin it or not pin it? are often made by individual developers in many organizations. So some of those decisions aren't really bubbled up to a level where someone who's thinking from a global risk perspective can actually make that decision of like, okay, for this particular project, I just want to pin it. It's like, for left-pad, for example. Left-pad is a function that pads a string on the left. It's very simple. It's something that if somebody looked at, they probably could pin that version in most cases, because it's so simple there's really not a whole lot going on there. But this gets back to that fundamental tug of like, I want those security fixes that are going to come out, and this whole system that we've constructed out of these packages, there's really no good way for someone to kind of consume that information without it being a complete firehose of information. And so we get back to sort of this basic level of risk management, and I'm going to talk about that a little bit here. So now I want to get down into the weeds of those four layers Aaron talked about and I was going to talk about some specific issues that those areas have, and then some general advice on ways people tend to approach solving those problems. I just want to mention that there's no silver bullets here, there's no magic fixes that are going to make your supply chain problems go away. In the end, it kind of gets back to what we've been discussing this whole time, which is, there's the tug and pull between owning everything and doing it all yourself, versus outsourcing some of it, or maybe even most of it, and trying to get someone else to do it. And fundamentally, when you're pushing stuff outside of your organization and having other people manage it, you are inheriting that supply chain risk. And so we're going to start out with cloud infrastructure, if I can get the slide to advance. There we go. So in the cloud, I mean, as Aaron had touched on earlier, this

could be physical, and we're kind of glossing over physical, because physical infrastructure is very similar to traditional sorts of supply chain problems. It's getting certain things in certain places from certain people. It's not to say those problems are easy, especially if you have, say, semiconductor shortages, which we have right now. But in the cloud, there's an additional layer that's been built on top of that basic compute, right, that basic networking and whatnot, and that software is maintained by some cloud provider, whomever they are, whichever one you pick. And that software has its own set of supply chain problems that hopefully, you're trusting the cloud infrastructure provider to manage. And it may not even be attacks, it might just be mistakes, as Aaron mentioned. So an earlier example in this webinar was S3 being taken down by accident and crashing tons and tons of clients. And so there's a few different ways that cloud infrastructure in particular is somewhat vulnerable to these kinds of attacks. So the first six of them I'd go after would be the fact that all the cloud provider-- you know, at a particular cloud provider, all of the users are using the same stack. So if I can develop an attack for that particular cloud provider, I kind of can attack many different targets. Or I could buy an attack that was meant for one target that's for sale on the black web or whatever, and just go and attack the US government with it. That's one of the issues that you might have in a cloud environment. You have to kind of think about, at least. The other aspect to think about is that cloud providers tend to be geo-distributed, so they tend to have lots of different installations around the world potentially. But they give you access through a single control plane and they generally are managed by a single control plane under the hood, you know, at the software level. And so, you know, somebody were to get access to that control plane, through any sort of attack, but especially some sort of malicious supply chain attack that no one sees coming, where they can potentially take over a large amount of systems very easily. And we'll see a little bit more of that later, where you have some system that has a large amount of control and it ends up being a juicy target that gets taken advantage of. And then the last point here is similar to some traditional kind of supply chain attack, in that it's injecting things into artefacts that you're going to make use of. So an AMI is an Amazon machine image. For other providers, they have similar-- like, they're basically snapshots of particular Linux _____ or even say, Windows installs, and if those things were to get infected with some sort of malicious software, then that could be leveraged against you, right? And it may be a mistake on your part. You might search for Ubuntu AMI and end up with some third party version of Ubuntu, as opposed to the actual correct one from Canonical. Or it could be that someone takes over the build process at your favorite AMI provider and injects malicious code in there. And you have different layers of that AMI. You could either see it in the kernel, you could see it as software that's on that AMI, or you could see it potentially in the hypervisor. A lot of this stuff would be transparent to you, right? It might happen before you were even aware of it, and we'll touch on that a little bit, the importance of monitoring. And then I've already touched on this once, but what happens if it just goes away? And this is just a callback to the earlier BGP hijacking issue that Aaron mentioned. All right, so these are just some general suggestions for some of these problems. The first one is hopefully obvious, but bears mentioning anyway, which is, don't assume that your cloud providers are invulnerable or that

just because you have an SLA with them that you don't have to worry about these issues. You should be thinking about what their processes look like if you can. Not all cloud providers are going to provide you with tons of details of what they do internally, especially if you're just a small time customer to them, or if they're just a small time shop. But most of them provide quite a bit of details in terms of how they manage their systems, like the processes that they have, to some level of detail. Now they keep a lot of the details close to chest, but you want to think about, if you were developing that system, what would be the choices you'd be making, and bear that in mind, right? They're not invulnerable. They have the same problems that we have building these systems. And you want to consider both the risk due to it just disappearing, or to someone coming in and attacking it, when you're doing your risk analysis. And so if your stuff is particularly critical, you might decide to have multiple cloud vendors, where you can actually deploy your application to, the whole stack from the bottom up. And that might be part of your disaster recovery plan, and hopefully you have a disaster recovery plan. And so if your application can go down for hours or days, then you might be able to tolerate the SLA of a particular cloud vendor, and just assume that they're going to do a good job and they're never going to make mistakes and it's going to be fine. Or you might decide, okay, now I'm going to actually have my software stack be able to be deployed in multiple different places, and you might also-- the other option would be to spread your infrastructure out across multiple cloud providers at once, though it starts to get more complicated as you start to do that. It's usually a tall order to try to-- I'm going to use three different cloud providers for my service, you know. It can turn into a mess pretty quickly if you're not able to manage that complexity. And, you know, you also can inherit risks from having that complexity too, so there's reasons why you wouldn't want to do that. And then monitoring and telemetry. So you definitely want to make sure of your cloud infrastructure's monitoring services. Most of them have pretty sophisticated monitoring tools, but you have to keep in the back of your mind that you don't want to be entirely dependent on just that cloud monitoring that you get from your cloud provider. It may disappear. If they're suffering a supply chain attack, or any other kind of attack, monitoring is going to be something that an adversary is going to go for immediately. They want to blind the operator. They want to make the system harder to understand what's going on. And so you want, at different layers of the stack-- and I'm going to call it out in a couple of slides here-- to have multiple layers of monitoring, so you have different views of the same system.

Aaron Reffett: We have a good comment here, a good question. How do you solve the supply chain in the cloud when you don't have access to or don't know which dependencies your provider has? I have thoughts. What are yours?

Richard Laughlin: Yeah. I'll give you my thoughts. I mean, this comes back to this sort of fundamental issue, right? Even in the case where you do have visibility into the software, as other people have noticed, there's often so many dependencies, you can't possibly look at them all. And so, you know, it really comes down to risk management, of thinking about what-- you know, is the risk of having a cloud

provider provide the service for me appropriate for what I'm trying to put on that particular cloud? Or is it the case that maybe I would prefer to have some _____ stuff where I have a lot more control over all of the features associated with it. Or, you know, in the case of say, GovCloud, this has kind of been handled to a certain extent, in that Amazon has gone through an extra set of steps to try to show that their system has certain characteristics, as far as security is concerned. So you might choose a cloud vendor that has certain certifications, or has certain audits done against them. All those things are trying to wrangle pieces of this problem, but in the end, there's always residual risk, right? There's always a section of the risk profile that you can't-- you have no control over, you have no ability to manage. And that's a section you essentially have to accept. But you want to be aware of what you're accepting, and that's really the key, is if you understand what your cloud provider has to do to provide you with service, you can understand what risks they are basically giving to you, what risks you're inheriting from them. Aaron, what are your thoughts?

Aaron Reffett: My thoughts. I think this is a good potential example of like, heterogeneity versus homogeneity within particular landscapes. You don't just have to go with one cloud service provider. You just have to-- you're here with a cloud service provider and a non-prep [ph?], or the hybrid based approach. Diversify your risk away here that forces your adversaries to have to take out multiple service providers as opposed to just a single one. Don't put all your eggs in one basket is a good mantra, I think, in this space. There is more than one way to solve this problem. You can't really fix your cloud service provider's software supply chain issues. The best you can do is you mitigate it the best that you can within your program.

Richard Laughlin: Right, yep. I agree. Someone else is following up with, how do you achieve zero trust in the cloud if you don't have the ability to necessarily-- you always have to boil down to trusting someone? My answer to that would be, zero trust is really more about identity. It's about setting up the ability for people who are interacting with your system to be able to prove who they are, even in a very robust way, rather than not-- you know, at some level, we have to trust someone. I mean, if you're using any piece of hardware, you know, you're trusting whoever designed that chip that there's not vulnerabilities in there, or that there's not malicious content in that chip. And so the only way to mitigate that would be, you're going to go out and make your own silicon. You're going to make your own chips. You're going to make your own hardware. And at a certain level, you have to stop. I mean, otherwise you'd get nothing done. And it really just boils down to, you know, what level of risk is appropriate. That's really what it comes down to, and I call up that later, of just that risk management, of doing-- sitting down and doing a threat analysis and doing these risk management techniques to try to actually manage your overall risk, and get it-- at least get an idea of what it is. I mean, some of the risks you can't really manage effectively, but knowing what they are makes you better prepared to respond to them if they were to come your way. All right, so now I'm going to talk about the platform, and platform is, in our depiction,

in our little diagram, it's a little bit abstract. It covers maybe potentially multiple different kinds of applications. But essentially, what it's meant to mean is, everything above the level of your cloud provider, whomever that might be, or your physical hardware, all the way up above the operating system or whatever, all the way up to your application. So the classic example of this in the DevSecOps space is something like Platform One or Kubernetes, which is going to try to provide you a lot of functionality, and then you're just going to plop your application on top. That's the selling point. One thing that Aaron kind of touched on briefly earlier was this idea of homogeneous versus having heterogeneous dependencies. And from the perspective of a large organization like the DoD, having say one Canonical platform is an interesting risk, because your adversaries who know you have that one particular platform will know how to attack it, right? They'll be able to dedicate all their time and all their money and all their resources to just that one platform. And so I think it'll be an interesting experience to see whether or not we want more heterogeneity across different areas, where we can have different stacks with different components in them, such that it requires our adversary to spend way more resources to attack all these different platforms. And the other piece is that all these different things, Kubernetes, Platform One, everything's based largely on open source, so our adversaries have full visibility into the components. It's not necessarily a bad thing. It does mean that we have more eyes on it, just in the general open source community as well. So there's higher potential that bugs get caught, but it does-- it carries with it some risks you have to think about, is it may be possible that your adversary is not just looking at the code, but they're also making commits, right? They're writing patches for the software that you're using in your stack, so you have to think about, what is the probability that they managed to sneak something through there, whether it be a design weakness or whether it be something of that sort? Those risks are there. And then the platform could also include other things as well. It doesn't have to Kubernetes, but that's popular these days. It could just be any software stack in the libraries associated. And any service you get from your cloud provider that's not strictly BMs [ph?] or Linux systems, you could view as a platform in this diagram that we've kind of constructed. So what do we do with the platform? Here we go. So general suggestions for dealing with platform risks is get into the weeds. So understand, if you're using Kubernetes, you really need to understand Kubernetes. If you're using a console, you really need to understand how it works, especially with something like Kubernetes where there's lots of different flavors and distributions. Those distros and those vendors, they're making opinionated choices for you on your behalf, and different ones are making choices that you might not necessarily want in your system. So you really need to dig into those details and understand what those choices are. So I'll give you a quick example. In Cube Admin-- which is one of the variants of Kubernetes that you'll find out there. It's an upstream project-- the master nodes are essentially just special worker nodes, and there's consequences for making that decision. Other vendors take all of the master level services and cram them into one executable and runs as one executable. And so there's tradeoffs for that. So understanding architecturally what that means is critical. And the second bullet to this is, when you're using an off the shelf platform, essentially, you want to understand what all of the optional components are. What are the

things you can rip out? And you want to strip it down as much as you can to just the critical pieces, because you want to limit the attack surface and reduce the number of supply chains that you're dealing with. Obviously if you need those components, then you need them, but if you don't, then you should remove them. And you can only do that if you understand your platform well. A lot of people think of the platform as this generic thing that they could just grab it off the shelf and use it. And when you're dealing with these kind of risks, you really can't. All right, and now we'll talk about the factory. I want to specifically talk about the factory inputs. The factory's made up of pieces of well of course, but those pieces fall, in my view, largely into the platform and the pieces of the platform. But you could also view them as inputs to the factory here as well, depending on how you want to cut the-- like, put the line between the two layers. And this is essentially the base images that are making up your containers, and I'm assuming containers here, because it's pretty common in DevSecOps. The distribution packages that are included in that. So whatever you get from Yum or APT or whatever you're using. And then any third party tools or libraries you pull into that beyond what you're getting from your distribution. So this is probably the area that has the most attention to this problem, and it's probably the most tractable of all them. But it's still, as we've discussed earlier in some of the discussion, it still can get a little bit hairy. So the first one is, wrangle these things down, right. It's the attack surface thing. Look at whatever you have and pare it down to the minimum that you can. So you're going to look for signed images in your base images. You want to make sure you're using-- anything that you can verify where the source is coming from is important for this. And then when it comes to base images, you kind of want to, for your particular program, you want to kind of consolidate down to a single package ecosystem, right. And this isn't always possible, because sometimes you want to get images from vendors, images from open source community. But paring it down to a specific package ecosystem gives you the benefit of, now you only have to track on particular Linux distro's mailing list, or what security updates do you need to get pushed out there and make sure, actually make it to production. And in the same mind, utilize scanning tools like Twist Lock, Amcor [ph?], whatever you prefer for containers, to scan them and look for these CVEs that might exist in the packages that you have. And then when you're actually building your software, carefully consider all the libraries. Make sure that you didn't do any typo squatting. Aaron talked a bit about that earlier, with picking the wrong name for the wrong thing. There's an argument for some programs may want to set a list of particular allowed packages, because they've gone through the effort of thinking about, is this worth it for us to use? The challenge of course is there's always a pushback to that, is it might make everything slower. It might slow down the development, and that's definitely a risk of doing that approach, of having that sort of heavy handed, here's the list of packages that are approved. But if you have the right process, it can be smooth, as long as you're sure that you can get those approvals in before it becomes a blocker for someone, then it doesn't really become an issue. And finally, and I'm going to go through here quick, because we're running low on time, there's the risk to your mission system from other applications. Things you're using to manage it, right? So SolarWinds, Pulse Secure, these are good examples of that. And any other applications that your service is using through an API or

something like that, there's risks that you're taking on by use of that external piece of software. And yeah, you just have to think about these and I'm going to talk about some of the mitigations for this on the next-- in the wrap up section. But basically, you want to treat these tools like you would any of the other tools in your stack, and you want to think about the processes that the developer that was building this tool has used to create it. You want to get as much detail from the vendor or whomever it is as you can. Often they keep that close to the chest, and that may increase the risk of using that particular piece of software, so you might opt to use something else. All right, so wrapping up, I'm going to leave just a tiny bit of time for questions, so I'm going to go quickly here. These are just general pieces of advice to follow. So take an inventory of all the software, the tools, the vendors, everything you have. Build a graph essentially of the supply chain that you have. Get a full global picture of what that looks like. And then you can use that graph when you're doing risk analysis. So if you're looking at the kind of system you want to build and how you want to implement it, you can think about whether that supply chain is good enough for that particular risk for a file that you have. And select areas of that supply chain that you want to take control of, take ownership of and take care of that. And then when you get to production, it really transitions over to monitoring and patching, and keeping an eye on the pulse of what's happening to all the pieces of software you've included, CVEs, chatter on forums about various things. It really requires hyper attention to all these details when you're using something that's coming from outside. Yeah, I think I'll leave it there so we have just a tiny bit of time for any additional questions.

Aaron Reffett: Yeah, I think we have a couple of questions that have come in a long the way. Here's a good one. "The DoD is using private funds, so GovCloud or sensitive-- or several _____ of workloads, also using appropriate public cloud services where appropriate." So yes, they do use a hybrid approach of the two. The DoD has a unique budget that most can only dream of. That is also true. GovCloud is not completely private. It's quasi private. The tenants allowed within it are government programs or other allowed entities, but they are internet accessible at the classified level too, those workloads are all internet accessible. And they use publicly routable IPs, so it's possible to access it from the global internet. So we don't consider it to be completely private. They do have private private clouds, particularly at the classified levels, that are completely private and routed and not internet accessible. However, they use a lot of the same technologies, and so an adversary could recon on a public cloud, learn how to exploit it, and then use another vector, one of these other supply chain vectors, to get the workloads over into-- or the malicious payloads over into the either air gapped [ph?] or less directly accessible areas. It's not really as much of a mitigation as it would appear to be on the surface.

Richard Laughlin: Yeah, sounds good. I think that's all the time we have, unfortunately.

Shane McGraw: Yeah, Aaron and Richard, great discussion today. We thank you both for sharing your expertise.

Richard Laughlin: Yeah, absolutely. Anytime.

Aaron Reffett: Thank you very much. If folks found this useful and would like some more deeper dives, particularly in solutions, we understand that just laying out the landscape of the issues, it's a large undertaking, and being able to dig down into each of these areas and be able to create some actionable solutions, that's what we're working towards with our sponsors. And as we develop some of these best practices, we'll be bringing them out to the greater good.

Shane McGraw: That's terrific. We'd like to thank everyone for attending today. Upon exiting, please hit the like button below your video window and you can share the archive as well. Also you can subscribe to our YouTube channel by clicking on the SEI seal in the lower right corner of your video window. Lastly, join us for your next livestream, which will be tomorrow morning at 10:30 AM Eastern, and it'll be on LinkedIn Live, and our topic will be cybersecurity, past, present and future, with SEI CTO on long staff. The technical director of our risk and relayings [ph?] team, Matt McCovey [ph?]. And you can watch that livestream on the SEI company page on LinkedIn. Any questions from today, please send to info@sei.cmu.edu. Have a great day, everyone. Thank you.