Carnegie Mellon University
Software Engineering Institute

SEI Podcasts

Conversations in Artificial Intelligence, Cybersecurity, and Software Engineering

Updating Risk Assessment in the CERT Secure Coding Standard

featuring David Svoboda and Joe Sible as Interviewed by Robert Schiela

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.

Robert Schiela: Welcome to the SEI Podcast Series. Today, we are talking with David Svoboda and Joseph Sible, who are both engineers in CERT's Applied Systems Group. Dave and Joe are our primary developers and maintainers of the SEI CERT Secure Coding Standard. Today we are going to hear about proposed changes to this standard, specifically in the area of risk assessment. And I am Bob Schiela, the deputy technical director of Cybersecurity Foundations in the CERT Division here at the Software Engineering Institute.

Welcome, Dave and Joe.

David: Hi, Bob. It is good to be here.

Robert: It is great to have you. Dave and Joe, you have both been guests on previous podcasts in the area exploring software security, for example, in the Rust programming language. For our viewers and our listeners who have not seen or heard those podcasts, I would like you to provide a little background information about who you are, and how did you get into secure coding research.

David: Well, Bob, I started with CERT about 15 years ago. When I started, I had experience as a research programmer working with C, C++, and Java. I had a pretty good sense of the capabilities, the good points, and the flaws of these three languages. Adding more languages like Rust, I come to it from a language lawyer perspective, more of a Stradivarius perspective of *How is this language put together* rather than a Paganini language, *How do you play it?*

Robert: Right, and I know you have originally a degree in computer science. You are coming from the computer science perspective.

David: Right, that too.

Robert: Right. How about you Joe?

Joseph Sible: I got a little bit of a later start than Dave did. I've been with CERT for coming up on eight years. It was around 2020 that I started maintaining the coding standards in earnest. My main focus has been on the C language with a little bit on the C++ standard.

Robert: Recently, there has been a lot of attention in C and C++. The memory safety of those languages and other languages that are more memory safe. Let's give a brief overview of the secure coding standards, remediation costs, and the proposed changes, how the metric was initially measured, and why now is it outdated?

Joseph: Remediation cost is measuring how much manual work is required to find and fix violations of some rule. It is right now a number between 1 and 3, and it comes out of two different categories of whether the rule is automatically detectable or not, or whether it is automatically repairable or not. Before I go there, I am going to explain the concepts of detectable and repairable, the way we use them. Detectable isn't just *There is some tool that says it can detect this thing.* It's *It can be reliably detected with a low false positive or false negative*, right? Repairable only qualifies if it can be fixed locally. Say not changing a function's definition in all of its call sites, just its body. And that it doesn't have the risk of breaking properly working code. In practice, that means unless it was undefined behavior before, the existing behavior has to be preserved.

Robert: OK. So the standards as a whole are a good resource for learning about secure coding and how to secure code. With respect to the remediation cost, one of the values of the coding standard is that it also helps with priorities of finding and fixing weaknesses and flaws in code. I think what we are talking about with detection is how tools or what methods might be able to detect. Then, similarly, as the technology is advanced, how tools and methods can fix or automatically fix some of the flaws. It is helpful with regard to picking and choosing and prioritizing which weaknesses and flaws you might address. Is that correct?

David: That is right, Bob. Specifically, well, the risk assessment basically produces a composite number that we call priority. The remediation cost factors into the priority with the argument that rules with a higher priority should be fixed and addressed before rules with lower priorities. A static analysis alert that is automatically detectable will have a higher resulting priority than a rule that is not automatically detectable. Likewise, an alert that is automatically repairable should be attended to before a rule that is not automatically repairable.

Robert: For those that use static analysis tools, they probably know there is a common problem of too many alerts and findings for an organization or a team to actually address. So prioritization becomes important for helping to decide where to put your attention to fix those. That is where the risk assessment as a whole can help, and remediation costs coming back down to part of that. I was hoping you could walk us through your approach and efforts to updating the standard for the remediation cost.

Joseph: The change we are basically making is, as more and more tools are capable of doing automated repair, we don't want to gate repair behind detection anymore and having an effect on the score. Today, if an alert's not automatically detectable, it automatically gets the highest value of 3, whether or not it is automatically repairable. But now that there are repairs that even if the detection was a false positive, so we don't want to call it automatically detectable, it is still worth doing the repair because it doesn't hurt anything if it is a false positive. If it is a true positive, it will fix the problem.

David: Right. To add to that, we discovered this during a two-year project that we put together called Redemption. Initially it stood for a redemption of false positives because the argument was that we would repair alerts even if they were false positives, with the argument being that repairing a false positive is harmless to the code, for example, a null pointer. If you have a null pointer in line 10, you could spend hours trying to figure out if that is a true positive or a false positive, or you could just simply add a null check on line 10 and fix the problem, which takes five minutes. It will take longer to test, but the actual coding repair can be done automatically. It is easier to repair the alert than it is to try and determine if it is a true positive or not.

Robert: Right. So that gets to how different the method and measuring the findability of a weakness is from its ability to fixing a particular weakness.

David: Exactly.

Robert: At the same time though, you also are rolling out or considering these changes, and one thing I don't think we mentioned much is whenever you are transitioning for a change, you need to somehow support legacy or minimize how it might impact previous metrics or uses of the metric and have some way to convert to the new metric. Can you talk a little bit about that change of the new way and how that might impact people using in the past?

David: Sure. As engineers, we like building systems. We like making things better. At first, we wanted to rush ahead and make this change, but this change of updating remediation costs to about 100 CERT C rules—that is the total number we have got—making those changes might break some tools that depend on it. For example, the Parasoft tool provides nice nifty reports about your compliance, how your code base complies with the CERT C standard, and it heavily integrates our risk assessment, remediation costs, and everything else. If we change how the remediation cost is computed, that requires some work on the part of the Parasoft developers. So we don't want to make this change without at least alerting them that, *Hey, we would like to make this change. What costs does this impose on you?* We want to make sure that the change is handled smoothly by Parasoft and by everyone else who are depending on our coding standards.

Robert: Right. Because there are many different types of users that use the coding standards, whether it is the coders or tool developers that are trying to map to the secure coding standard. So we need to consider all of those different roles and perspectives.

David: Changing the remediation cost will also change the priorities because the priorities are computed. That means some rules go up in priority, others go down, and this will affect most people who use our coding standards for auditing code. That also includes static analysis tool vendors like Parasoft.
Robert: Right. Right. Again, with respect to the new way to measure

remediation costs compared to the old, can you outline how you have tested and evaluated the new metric and its impact on how people are using the standard for their code for secure coding?

David: Right. At first, this was a solution that was strictly in my head, and I thought, *I will just roll this out. No one will really complain about it, right?* Well, it is not that way. I decided that what we should do is pick a handful of rules, and we wound up picking the C integer rules. I think there are seven rules. They are INT30-C through to INT36-C.

Robert: So those are all related to integer types of weaknesses or problems that might cause insecurity in code.

David: Right. Such as avoiding integer overflows, avoiding division by zero, that kind of thing. And we simply said, *Here are the old metrics, old remediation costs. Now given our new system, what would be the new remediation costs? Do any of them change from the old?* What we tried doing is we did an exercise where it was you, Joe, and me, and, did we have someone else, I think Caden?

Joseph: Yes, we had Caden on board.

David: Right. Each of the three of us tried to figure out what would be the new remediation cost for these rules. How should they be changed? What we discovered is that we all had the seven different metrics. I thought this one would be high, and Joe thought it would be medium, and Caden thought it would be low. That just simply meant that we need to be more precise about the definitions. It also meant it was very prudent of me not to rush ahead and make this change without at least, without being a lot more careful in involving Joe and Caden in it. They are both smart guys, and they both helped me discover what was unclear or how we interpreted the definitions differently.

Robert: Right. Right. Because you each had a different perspective, so there was some ambiguity that you wanted to make more precise.

David: Exactly. At this point, we came to an agreement about what the remediation cost should be. We streamlined the definitions so that we all had consensus. Now at least, we have consensus between the three of us. We want to bring this new method to everyone else to see if there are any other problems or difficulties with consensus. Does the consensus only apply to the three of us, or does it apply to you and everyone else as well?

Robert: Have you had any chance to evaluate the impact of the new metric and how this new metric with respect to adding the fixability perspective will be used either by tool vendors or by developers with respect to this new dimension that you have added to the metric.

David: My previous boss always said that the goal of the SEI and the goal of our work with the DoD was to address the DoD's pain points. What problems do they have that are really painful for them to work with? One of the pain points, as we have already talked about, was the deluge of alerts. They had far too many alerts to deal with. Now this is definitely true, but I have experienced this pain point very personally because I have done a whole bunch of audits of DoD code, and that means that I have to deal with the deluge of alerts. I have had code bases with hundreds of thousands of alerts to deal with. Of course, I can only look at maybe 1,000 or 2,000 of them and have to leave, you know, 98,000 unexamined.

Robert: Yes, it is just overwhelming. Actually, I know that sampling and selecting which ones to evaluate is a whole different topic, but go on.

David: First of all, our Redemption project was partially designed to help deal with this deluge of alerts by automatically repairing alerts so that 100,000 alerts comes out to a much smaller number like hopefully, 30,000 or 40,000, but that is Redemption. As far as the CERT standards go, we would like for the remediation cost to help with addressing that deluge of alerts. For example, an alert that is automatically repairable, you don't need to audit that alert. You can use a tool to repair the alert automatically, and, therefore, you don't need to audit it. If an alert is not automatically repairable, but it is automatically detectable, then, as Joe said, that means the detection is very reliable. The chance that it is a false positive is low enough that it is not worth your while to audit the alert, and it is just best to pass it off to a developer saying, *This is a true positive, or let's assume this is a true positive. Please fix the code*.

Robert: Right. So it may be that an impact of use is that for some projects, if there is a high automated fixing aspect of the rule, they just fix it, and they don't worry about the other portions of the priority or whether or not it is specifically findable or specifically determined whether or not it is a true positive, so they could just choose to not spend time on determining whether or not that particular high automated fixability weakness is actually a problem. They can just fix it.

David: Precisely.

Robert: Great, so I think that could be really useful. As we said, the efficiency and the cost of reviewing and auditing all of the alerts and findings that have been found is really high. As automated fixing tools in the future are improved, this can help the auditors decide which findings they don't have to worry about or spend much time on. They can just let the tool fix it. Maybe the tools even fix it before it gets to the auditors in some future version.

David: Right. Theoretically, the technology will be there even if it is not here today. While we were obviously thinking of the Redemption project when we did this, there are other automated program repair tools that will come along and maybe perhaps do the job better.

Robert: As a federally funded research and development center, an important aspect of our work here at the Software Engineering Institute is transition. Can we talk about how the outcomes of this research could impact the DoD, the Department of Defense, and other users?

David: The main thing is obviously the DoD, they first of all have a lot of C code in production and under development. Obviously, if they have C code, then they will have lots of alerts from static analysis tools. Obviously, they could just choose to ignore them. They could choose to not run the static analysis tools. What that means is that the alerts that they don't fix translate to risk. If they put the code in production, and that code goes into your tank or your helicopter, that is some risk that is inherent. The DoD doesn't like risk. I mean, nobody does. The point is this chops in half or into a third the amount of risk they have to deal with by automating some of the repair and by skipping some of the auditing that they have to do. It does require that they use a good static analysis tool and a good repair tool. But again, we are kind of assuming that those will exist. Static analysis is a 40-year-old industry. We have a good idea of how to do it now. Program repair is much younger, but there are good examples of program repair, including but not limited to Redemption.

Robert: Right. Going back to prioritizing the findings, this is another new way that can help with triaging which findings to address, specifically triaging and identifying which findings can be addressed automatically. So it should help the efficiency of making much more secure code and reducing risk at a low cost because you are trying to help them identify which findings can be fixed automatically without much effort.

David: Exactly

Robert: OK. Great. What upcoming work and updates do you have planned for the secure coding standards?

Joseph: We have two initiatives going on right now. The smaller of the two is just an overhaul of the website, how we host the standards. They are going to live on GitHub pages. We are going to accept new comments and suggestions via the existing GitHub pull request system.

Robert: Right now, it is on a wiki site. Is that correct?

Joseph: Yes. It is running on a confluence site we host right now.

Robert: We are converting from a wiki site to a GitHub where people will be able to basically manage the documents as code.

David: Yes, there will be GitHub repositories. So if you want to make changes, and you are authorized to, you can check out the code and just change it like you change any other GitHub project.

Robert: Okay. Not to take too much time on it, can we say a little bit about what is leading that change? Is it to try and, for example, interact with the community in a different way that we think would be more useful for them?

Joseph: That lets us be more friendly to people who want to contribute constructively without having as much of a burden of fighting the spammers and other kind of malicious actors on the internet. GitHub already has infrastructure for that.

David: It also allows us to automate some things which are difficult to do. For example, our coding rules have lots of non-compliant code examples. You know, *Don't do this at home* and compliance solutions, *Here, you should do it that way instead*. But right now, these are all on a wiki. They are hard to test. It is hard for us to make sure that there aren't typos or reasons why these don't actually compile or run properly or behave the way that we say they do. One thing we want to be able to do is automatically scrape them and make them available as test suites.

Robert: Right. So instead of the code being part of the documentation, you want to take them out as snippets of code and manage them as code. So you can automatically test them or use those code snippets for other mechanisms that use code, basically handling it as code instead of documentation.

David: One other thing that we could do—I don't think this is going to happen this year—but we could institute a playground whereby when you see a code snippet, there will be a *Run* button which you can click on and see what the code actually does. Surprise, it doesn't do what it was expected to do, but it does what we say, *This thing should be printing one. It is actually printing two. Click this button to find out*.

Robert: What other updates are being planned?

Joseph: The more significant update I expect more people will be directly interested in is we are looking at adding more languages that we have coding standards for.

Joseph: Right now, our shortlist is Rust and Python. We are actively in the process of compiling ideas for rules for both of those languages.

David: It is a fairly straightforward matter to simply look at, say, the C standard and say, *Which of these rules would still apply to Rust? Which ones would still apply to Python?* That is easy to do, but at the SEI, we prefer hard problems because they are hard.

Robert: Right, and so Rust and Python are both going back to that term of memory-safe languages or, if you use them in certain ways, memory-safe, but this will not be our first memory-safe language with standards. We have Java, which also is memory safe. There are obviously, since we have a standard for Java, more secure coding guidance and weaknesses in the language than just memory safety issues. So you are going to try and find those and help developers with that in the new languages, it sounds like.

David: Yes. Java and Python are both memory-safe, and Rust is memory-safe, but it is not memory-safe in the way that Java and Python are. Because Java and Python completely automate away the issue of managing memory. They both use garbage collectors to collect the garbage. But Rust does not do that. It simply provides a prover inside the compiler, what they call a borrow checker, which makes sure the code that you have written is memory-safe, and if you write memory-unsafe code, it simply won't compile.

Robert: Right. As I mentioned earlier, we have blogs and podcasts on the C, C++, and Rust differences in memory safety as well.
Joseph: To emphasize one of the points Dave was making, a key difference of

how Rust works is the kind of mistakes that will give you a runtime error in Java or Python, they will get caught at compile time in Rust, so it is one less thing you have to worry about running into for the first time in production.

David: In theory, you will get the same performance and low memory usage, and low energy or battery usage in Rust that you get in C and C++. You don't have to pay for the excess energy that you would if you use Java or Python.

Robert: Right, with its active runtime management. Yes, great. Anything else you want to add?

Joseph: I think we have said everything.

David: Yes.

Robert: Well, Dave and Joe, thanks again for joining us today. For the audience, we will include links in the transcript to the resources mentioned in the podcast today, including the blog post on the topic. Finally, as a reminder, our podcasts are available pretty much anywhere you can access podcasts, including SoundCloud, Spotify, Apple Podcasts, as well as the SEI's YouTube channel. If you like what you saw and heard today, please give a thumbs up. Thanks again for joining us.