Automated Repair of Static Analysis Alerts

Featuring David Svoboda as Interviewed by Suzanne Miller

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.

Suzanne Miller: Welcome to the SEI Podcast Series. My name is Suzanne Miller, and I am a principal researcher in the SEI's Software Solutions Division. Today, I am joined by my friend and colleague, David Svoboda, a software security engineer in the SEI's CERT Division. In this podcast, we are going to explore using automated code repair technology to improve static analysis alerts during the software debugging process, and I am very excited about this particular podcast because anybody who writes software in controlled environments knows that the static analysis alerts are a bane on their existence. I am so excited, David, that you are here today, and welcome back to our podcast. For those who do not know you already, can you tell us a little bit about how you ended up at the SEI and the cool work that you do here?

David: I joined the SEI in 2007, and before then I had been a professional programmer for about 18 years. And during that time, of course, I had read lots of stuff about how to program and talked with people about how to program well. But there was always something missing. Until I joined the SEI,

I did not know about how to program securely. It was kind of this hidden section about programming that is critically important. It is much more important now even than it was in 2008. But I had something of an epiphany where I realized that it was kind of scary in that I had programmed for 18 years without knowing all this stuff. I knew that if you try and write 12 characters to a 10-character array, you can get, *the behavior is undefined*. Do not do that. But I did not know how easy it was to hack someone who had done that. The first major task that I did in the SEI was teach secure coding, and I have seen students and professional programmers who learn secure coding also get this epiphany. They realize that they have been in secure programming for their professional lives, and they had to go and sin no more. It was an epiphany. It was a *come to Jesus* moment, and I have seen this in many other people. That is why I joined. That is how I joined. And that is what I have been working on since I joined the SEI.

Suzanne: Excellent. All right. One of the tools that came into being as part of the whole secure coding movement is a type of tool called a static analysis tool. And there is good and bad about those tools if you are a programmer. But what are they, and what is the good part of static analysis tools?

David: I am reminded that when I first joined here, the job description said, *familiarity with static analysis*. And I did not know at the time what static analysis was. I knew static analysis tools. I just did not recognize the phrase. But fortunately, that ignorance did not stop me from getting this job. The best way to think of…

Suzanne: We were grateful that it did not stop you.

David: Thank you, Suzanne. The best way to think of static analysis tools, if you are not a programmer, is think of it as a spell checker. If you are working with a document like Microsoft Word, it will highlight the words that you misspell and it will now highlight grammar things, if your verb and noun doesn't agree, for instance. In our secure coding business, we have sometimes made a business of, *we will audit your code and tell you what is wrong with your code for X dollars*. There are several government and military groups that do the same thing, and businesses. That is analogous to proofreading a document looking for spelling errors, doing it yourself. But if you are proofreading a document, then you could spend the time just reading through the document from top to bottom. And that takes a long time. Or you could just see what the spell checker says and fix things as the spell checker notices it.

Suzanne: OK. That is a good thing, that we can spell check and not make mistakes, and similarly, that we can use static analysis tools and highlight potential mistakes. But I will note that using a spell checker, sometimes it is wrong. Sometimes I have some context of use that is different than what the spell checker and the grammar checker is thinking. It is what we would call a false positive in terms of what the spell checker shows. A human typically has to make that decision as to whether I am going to accept what the spell checker gave me or if I am going to ignore it or add it to the dictionary or whatever. What is the analog in static analysis tools? Because I know there is one.

David: Yes, I still am trying to convince Microsoft Word that *Svoboda* is not a misspelled word.

Suzanne: There you go.

David: I am guessing you do not have that problem yourself.

Suzanne: Miller is not one of the names that causes that problem. But, yes, I get it.

David: Right. Lucky you. Of course, static analysis tools are often subject to false positives. In fact, the definition of static analysis is it studies your code without actually trying to run it. You know, if there is a concern that the code might have malware in it, then not running the code is a very good thing. But, as you noted, it has a problem with false positives. And sometimes there can be so many false positives that the programmers just say, you know, *screw it. I am not running this. I am ignoring the rest of this stuff*. It dampens people's faith in the tool and makes them want to switch, want to ignore it from then on. There are other problems with static analysis. Probably the biggest problem is something that no one really talks about, and that is false negatives. Static analysis tools do not find every possible problem. Partially that is because the problems are often not well defined, but sometimes they are simply too difficult for the tool. Static analysis tools, for instance, are not that good. They will try, but the are not that good at, say, discovering SQL injection. It is just a naturally difficult problem to detect statically without actually trying to run the code. But the biggest problem, and this is not so much the fault of static analysis as it is the fault of the programs, is that sometimes programs are large. We have done some analyses here at the SEI, and the average size of code that we have analyzed is about two million lines of code.

Suzanne: That is the average.

David: That is the average. I have seen code, you know, you can write code that is as small as one line of code or as large as 300 million lines of code, and, you know, trying to statically analyze two million lines of code is going to get you lots and lots of alerts. Some of those alerts are going to be false positives. I actually did a back-of-the-envelope calculation last year, where I discovered that trying to analyze all the alerts produced by the average code base and fix them all would take about 15 person-years. It would take you or me 15 years to do it. It would take 15 people one year to do it, and that is just not reasonable. That is far too many. We have now nicknamed this problem the deluge. But that is really the biggest problem facing static analysis today.

Suzanne: But you are researching at least a partial solution to this problem. Tell us about your automated code repair, which just the name alone, you know, makes me feel happy, but I know that it cannot be a complete solution at this point in time, so tell us about it. How far have you gotten?

David: I did not invent that name, actually. That name came five years before on different research. But what I am solving, on the first level, the program basically does repairs based on static analysis tools. If a tool complains that, you know, you are adding two numbers which might overflow on line 10, then we write a check on top of that. It is actually rather difficult to statically analyze if an addition overflow is possible, but it is easy to add an automated check saying *is an addition overflow possible?* If so, then, raise the alarm, you know, sound the trumpets or something, and if not, then just keep going on. That is pretty easy to add automatically. And the nice thing about this is that we are ignoring the fact that an alert might be a false positive. In fact, the original title of this project was called Redemption of False Positives. It used to be that if we find an alert that is a false positive, we ignore it and discard it and it never gets fixed, and that means the static analysis tool keeps complaining about it forevermore. But if we fix them, then the false positive is not ignored; it is redeemed, and so, it has unofficially been called the Redemption project. We are trying to basically take three types of alerts that most tools produce. They will tell you if a pointer might be null when it is dereferenced, which is a no-no. They will tell you if you are trying to read a variable that has never been initialized, that is also bad. And they can tell you if you are ever executing code that has never executed, dead code. We are basically trying to fix these three categories of static analysis alerts. And with those three we are trying to obtain—we are trying to make sure that we repair as many of them as possible. We probably cannot repair 100 percent of them, so we just drew a line in the sand and said 80 percent. We will try

and get 80 percent, see if we can repair 80 percent, and we will report on how many of these things that we repair over the course of our two-year project.

Suzanne: How is that going? I mean, are you seeing measurable progress towards that 80 percent goal? I have to imagine there is a lot of nuance in terms of, you know, this kind of error in this category has, you know, six or seven or 20 different symptoms, and you have got to be able to deal with all those symptoms to be able to actually figure out what the correct redemption—I love that word—what the correct redemption is. How is that coming along?

David: It has now become the official name of the tool itself. We had to change the name of the project to Automated Repair of Static Analysis Alerts, which is, you know, wordy and more descriptive but less memorable. To get back to your question, of course, we picked those three categories because they are the most prominent. They are, first of all, fairly well-known categories of vulnerabilities. Secondly, they are the most prominent. We basically ran some static analysis tools on some open source software, specifically Git, which lots of people use, and Zeek, which is not as widely known, but it is used by several of our more important clients, so they would love to see some repairs to add into Zeek. And in these two code bases, we found lots of alerts of null pointers and uninitialized variables and dead code. We basically have lots of examples that can be repaired. Well, there are too many examples for us to even look at them all, so we have a whole bunch of buckets of subcategories of these, and we basically picked five alert categories for each one. We are trying to repair all five. And at this point, you know, the code is all written, the testing is there. It is sort of approaching 80 percent. It is not quite there yet. Our hope is that it will be 80 percent by the end of, well, by the end of June [2024].

Suzanne: Excellent.

David: At this point, we have until September [2024] to finish because that is when the project ends and the funding stops. Our hope is that we can convince either our collaborators or external people to continue funding so that we can add more repair algorithms and give it more capabilities. Right now, the project is an open source project, it is available on GitHub, and we can publish the URL, so anyone can download it, play with it. And it is containerized; it is running inside a single Docker container, which removes, well, it removes lots of portability issues, both for people who need to run the tool and for us, in that we are all developing the tool, and you know,

some people have Windows with Linux virtual machines. Some people have Macs, and so it saves us from a bunch of headaches.

Suzanne: When you start thinking about automating this process, what that opens up is the ability for this automated code repair to be integrated into a continuous integration [CI] pipeline. And are you doing that? Can you give us an overview, first of all, give us a quick overview of continuous integration so those that have not run into it will know what we are talking about? What are you doing in relation to bringing this into the continuous integration pipeline?

David: Sure. In fact, I am now realizing that when I compare static analysis to a spell checker, that glosses over several additional difficulties because a spell checker will highlight your words as you are typing them, which means you get feedback very quickly. Static analysis is slower. It does not give you feedback immediately. I mean, there are a few tools that specialize in that, but most tools do not; they take a long time.

Suzanne: The metaphor holds well enough.

David: Fair enough, yes. To me, continuous integration kind of solves two problems. First of all, there are lots of tests that we need to do in order to make sure that our software works as well as it does. It is very easy to modify software, to introduce a bug, or to break some code, something that used to work well and no longer works. There are lots of testing that has to be done, and unless the testing is automated, it does not get done consistently. I might have someone run a test, and they say they ran the test. Well, did they run the test? Not really sure. They did not complain. They said it worked, so you kind of have to take their word for it. But if the test is run by an automated system, then you can always look back and it is logged, and it will simply say, the test succeeded. It did not tell you anything about it. The second problem that—well, it occurs even with individuals, but it occurs a lot more with organizations. We wind up going through a lot of computers. We wind up having a lot of old hardware around. Still works, still perfectly good, but what do you do with it? When I first started programming, programmers were cheap, and computers were really expensive. And today, the opposite is true. CPUs are especially cheap. You can rent CPU usage on Amazon or Azure Cloud. Today, continuous integration sort of kills both these birds with one stone. We have a lot of extra CPU cycles. Let's put those CPU cycles to running automated tests. Even though the tests are unlikely to come up with a problem, they still are useful in detecting the regressions when they happen. Continuous integration is a very good thing that, you know, the SEI is

now promoting and, you know, we are using it, we are using it in our own project, but everyone should be using it. They should be making their tests as automatic as possible so that you do not have to rely on the perseverance of people who would rather be writing code anyway rather than testing it. That is why continuous integration is a good thing. Most development teams will simply have a continuous integration server that monitors any time someone submits new code, it will simply try and build the code, run the automated tests, and holler if the tests fail. If anything fails with it, then the CI system will say, *you have a failure*. *This test at this time, given this input, you know, this is the bug that you need to fix*. That is continuous integration in general. How does that work with our automated code repair, our Redemption tool? The answer is that oftentimes, people will already run static analysis inside a CI system, run a CI, sorry, run a static analysis tool and take the alerts the tool generates and send them back to the developers, and then let them figure out what to do with them. Now as we already know, there can be a deluge of static analysis alerts, far too many for developers to deal with. In our case, let's simply embed the Redemption tool into our CI system and have it offer a repair, have it repair the code. Actually, it does not have to repair the main code. It can simply fork a branch off of the main code, repair the branch, and say, *here's some repaired code. Would you like to accept this repair or reject it or modify the code yourself?* It is a lot like a spell checker that if you right-click on a misspelled word, it can offer you, you could spell this word or this word or this word, or you could decide to accept the current spelling as it is. We are not judging here. We are just simply saying this is the traditional way to do things. Basically, CI, continuous integration, lets us turn the problem of static analysis and redemption into something that looks a lot more automated and quick like a real spell checker. That makes my analogy better.

Suzanne: Yes, there you go, and I can imagine that there are a whole bunch of programmers listening to the podcast with the words *where do I get it?* on their lips. Let's talk a little bit about how we are transitioning this tool out into the community. You mentioned that in our GitHub, we have got a Docker with the tool inside available for people to use. Are there other transition materials? Do we have any training materials? Any other things that people can use to become familiar with this Redemption tool and to make best use of it?

David: Well, the answer is yes, and to go into details. I will simply say that the materials that we are developing are mostly demand-driven by our collaborators. At this point, we have four collaborators in the DoD, and we are mainly doing things to try and convince them, *hey, this tool is worth using. You should try it. It should, you know, try using it to repair some code on your*

end. Our end goal is that they decide this is so good we are going to use it ourselves. We are going to integrate it into our own CI processes and, you know, that is my end goal. That is what I hope that happens by September 30th. However, what they have done, of course, is as any users do, they ask questions, and they say, *can you do this? Can you add, you know, support for fractions?* I am thinking, *what is support for fractions? What does that have to do..*? Well, and I say, *tell me more. What do you mean? What do you—what are you thinking of*? For instance, a few of them initially said, you know, *this is hard to use. I am not sure, like, what the correct command line invocation is. How do you—how would you put this into a CI system*? What we have done, what I have been doing for the past two months is making demos. First of all, the demos were partially to try and make it easier to use from the command line, and I have created some demos. They are in the GitHub repository. Right now, they are just simple README files. You know, first do step one, then do step two, then do step three. We are now actually making a few demo videos where I simply step through the demos and show them. We have not published any of the videos yet, but we are now talking about the feasibility of putting these videos on somewhere public, like YouTube. That is to be discussed. At this point, I mean, for this…

Suzanne: We have an SEI YouTube channel, so what better use to make of it?

David: We do. That is probably one place we are going to be looking at. Right now, the demos are primarily for instruction. It is simply easier for me to make a demo video than it is for me to fly down to Florida or wherever and sit with them while they try and type their way through the demo. Eventually, I think we are going to be making some promotional demos to simply convince people, *hey, you need this Redemption tool*.

Suzanne: In your research review presentation, which you did last year, you talked about there are—we are only dealing with three of the categories. If I remember correctly, you said there were at least 10 categories of significant vulnerabilities that static analysis will typically catch or attempt to catch. Is that how you are going to go forward, is to kind of go against those other 10 or other seven, and/or do you have something else in mind for continuing the project? What is next?

David: That is what I have in mind for the project; however, I also have collaborators, and what they have in mind is that they want—is simply that they want their code to work. The answer to your question is we are kind of transitioning from development to transition. Put another way, we are transitioning from developing the things that I think that the tool should do

to things that our collaborators think that the tool should do. You know, if the collaborators say, *do what you think is best*, then I am going to start adding the other seven CERT rules to the list so that we do what I think is the 10 rules that give the biggest bang for our buck. Of course, the collaborators might have different ideas, and they will certainly, such as they want support for fractions. I do not even know what that means as far as my tool, but we will have a discussion and probably wind up doing that.

Suzanne: Well, and to be fair, there are domains, if you think about nuclear power plants for an example, where there are some very specific kinds of vulnerabilities that are particularly dangerous in that domain that may not be as dangerous in other domains, and so…

David: Especially buffer overflows.

Suzanne: Well, there you go. You want to—you know, those domains may be ones that really have a particular need that does not come under the sort of just the general 10, so you can see where that can happen.

David: Right. The CERT coding standard is a general purpose standard for all C programmers, but there are specific domains, as you say, where certain things are far more critical or they have more stringent requirements such as, you know, they are using the MISRA standard or—well, I do not remember details, but yes, the nuclear, the National Nuclear Safety Administration has their own set of standards which I think are based off of MISRA, but they extend it. Yes, there are far more specific things you can do.

Suzanne: I cannot tell you how excited I am about this. I would call myself a retired programmer. I have not been, you know, actually building code for a long time, but there was static analysis in my day and the idea of having to deal with all of those alerts and either ignore them and know that they were going to be coming back again or try and deal with them. But we never had the 15 staff years to deal with just one batch of code. I am thrilled that the SEI has taken this on and that you are moving forward with it. And I want to give you great congratulations on achieving, just the top three is an amazing accomplishment in my book. Thank you very much, and thank you for talking about it today.

Suzanne: Well, thank you, Suzanne.

Suzanne: I hope other people will find this and will become happy coders that can be more secure in the way that they approach their code.

David: They will be more secure coders and they will still have some alerts to deal with, but the alerts will be more interesting because there will be like the one-of-a-kind alert that simply does not occur enough for us to try and do a repair for it.

Suzanne: Right. All right. Thank you again, David. I look forward to talking with you in the future about more progress with this. We will leave that open. For our audience, we will include links in the transcript to the thing you really want, which is the link to the GitHub where you can get this thing, but also to other resources that David has mentioned, and there is a blog post on this topic. Things like that, you will be able to look at beyond just grabbing the Docker. Finally, a reminder to our audience that our podcasts are available wherever you get podcasts: SoundCloud, Apple, and of course, as we talked about earlier, the SEI's YouTube channel. When we are not putting up demo videos for David, we have got a whole bunch of podcasts out there that we hope that you will enjoy. If you like what you see and hear today, please give us a thumbs up, and thank you again for joining us.

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