Understanding Vulnerabilities in the Rust Programming Language

featuring David Svoboda and Garret Wasserman 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 Software Solutions Division. Today I am joined by David Svoboda, and Garret Wassermann, colleagues of mine that are both software engineers in the SEI's CERT Division. We are here to talk about their recent work with a programming language called Rust.

You may have seen our earlier podcast with David and Joseph Sible discussing security issues in Rust. In this podcast, we are going to examine tools for understanding vulnerabilities in the Rust ecosystem as well as the maturity of Rust overall so that people can make better decisions about whether or not and how they want to use it. I want to welcome both David and Garret, thank you for joining us.

David Svoboda: Thanks for having us.

Garret Wassermann: Yes, hi. Thank you.

Suzanne: Before we dive into our main topic, we always begin by having you tell us a little bit about yourselves, what brought each of you to the SEI, and what is special and wonderful about the work that you do here. Garret, let's have you start.

Garret: Sure, I came to SEI originally in 2014 as a vulnerability analyst where I was interested in learning more about cybersecurity and how to defend all of our software, computers, and networks from various security threats that exist out there on the internet. For a number of years, I actually worked in the CERT Coordination Center, where I published vulnerability notes, which are publications that detail software security issues that exist in software that you use everyday as well as details of workarounds and fixes and patches, whatever is necessary to try to address the issue and improve the security. We would work with both security researchers as well as the software vendors themselves to try to get these patches of information out to the public and improve our general security posture.

As I was studying those vulnerabilities and learning more about software, I moved into more executable code, or binary-code analysis work, where we try to take all of the information that we have learned about how these vulnerabilities come about basically and turn it into tools that we can automate or at least assist with analysts in order to better understand vulnerabilities, to find them, and to correct them. I would say that the best part of the job has been to explore new ideas related to this. That is what eventually brought me to looking into Rust for this podcast.

Suzanne: All right, and David, refresh everybody's memory as to how you came here and what it is that you enjoy about working here.

David: Sure, I joined the SEI in 2007 in the Secure Coding group. The Secure Coding Initiative was actually an offshoot of the Coordination Center, Garret's same Coordination Center. We were focused on the programming mistakes that allow these vulnerabilities to exist. I had been a professional programmer for over 15 years before joining, but I discovered that secure coding was sort of a secret section of programming that programmers never worry about.

If you want to be a programmer, you have to be an optimist. You have to assume that the input will be correct and all the data will work and nothing goes wrong. And then your program runs correctly, and everything is beautiful, and the angels sing.

But life never turns out that way, and things go wrong. I had an epiphany that when it comes to software security, most people are sinners. They create lots of vulnerabilities and problems. Garret is kept busy by detecting these things long after the software has been released. I have taught secure coding in lots of places around the world. Most people have the same reaction, it is something of an epiphany to people. They leave my course, and I say, *Go and sin no more!* They may still make mistakes, but they are all missionaries in this. It is almost a new religion of coding securely.

That is also the really great thing about this job, it teaches people who think they know everything—and I was one such person—that they are annoying for those of us who really do know everything.

Suzanne: We all have things to learn and truly, the technology evolves in ways that we cannot always predict what the vulnerabilities are going to be that come out of this new thing that we can do that we could not do before. We all have to have a learning mindset if we want to not get taken by some of the new things that happen.

David: Yes, technology has evolved to require a lot more security. You can thank the internet for that.

Suzanne: Yes, thank you internet. All right, so let's talk about Rust. We have described Rust before. I liked the way you described it in our previous podcast, because it is the first language that has really been able to compete with C in the embedded-systems world for software. It corrects some of the danger zones—I love that term—some of the danger zones that are present in the C language. That is one of the reasons that it is being adopted by people. I do not know if there is anything else you want to say about the language itself before we get into your blog post, and the vulnerabilities. I will let you answer that, David, if you have anything else you want to add.

David: I will go ahead and add that everyone has been aware of the danger zones pretty much since the late '80s. The catch is, of course, that many languages like C++, for instance, tries to mitigate the danger zones by providing safer versions of many of the dangerous techniques that C has. So C++ will have vectors for doing safe array managing. They will have variadic templates for handling variadic functions. But you can still use all the dangerous C mechanisms.

Java tries to replace the runtime environment with a safe runtime

environment that somebody says, *Hey, you are trying to access the 12th element of an array of only 10 elements. Your program is shut down now*. The problem, of course, is that Java imposes a performance constraint, because it has to check that the array is only 10 elements long. Other languages like Python generally follow the same footsteps. They provide you a lot of safety, but they cost you in terms of performance, and they use more memory than C does.

Rust is the first language that tries to give you both the runtime performance and slim memory footprint of C, while giving you the same safety that you get in Java and Python. They do this by adding lots of compiler checks, and that is why Rust should be able to compete with C in the embedded world and the small world where memory is limited and performance is truly critical.

Suzanne: Yes, and so that is really the exciting piece, well one of the exciting pieces of Rust. But it has some constructs that we have talked about that actually correct some of the most egregious danger zones, but that is all happening within an ecosystem. Any time we have a new language coming into the foreground, there is a lot that needs to go on with it, community building, and all kinds of things. We talk about that as the maturity of the language. Why is that maturity important in a programming language, and where do you think Rust fits in terms of the level of maturity and its ability to be supported so that people that want to use it do not feel like they are off on the edge?

David: We have divided up the blog post where Garret provided the vulnerability detection in Rust, and I provided details on the maturity. I realized when I started to write the maturity section that while I have an idea of what stability is, I did not really know how to define maturity for languages.

People say C is a mature language. Is it? I mean, why? I wanted to create a table saying here are a whole bunch of properties that languages have that are considered to be mature. For instance, C has a standards body. It is a long name, ISO/IEC/JTC1/SC22/WG14. The *WG* stands for working group, which is not very helpful, but the point is, it is a committee of about 25 people or companies that steward the language. I have been on and off the committee pretty much since I joined CERT. That is a sign that C is mature enough, it is not going to change. The committee itself is very conservative in that they have a successful language, they do not want to change it. The changes that they make to it, they want to be as harmless as possible,

because there is lots and lots of C code out there, and they do not want to break it.

So, after I made the table, and the table is in the blog post. It took me a day to add that and get all the facts right, but that table became the framework for the maturity section, and what I realized only after I made the table was that I still do not have a good definition of maturity, but the properties of maturity have changed for languages.

In the '70s and '80s, when C was big, maturity was seen as having a bureaucracy, a committee to steward the language. Today, Rust, as well as languages like Python, do not have that, they just have a foundation. There is usually a not-for-profit company that stewards the language, but they are basically a bunch of volunteers, although there are usually volunteers from companies that say*, You can spend some time making Python better*, for instance.

Suzanne: Right. So it is not quite an open-source model, but it is moving more towards an open-source model.

David: It is more towards open source, and it is more technical, because Rust for example has Crater. You can think of it as a continuous regression-testing system that tests all of the Rust code out there. Testing all the code out there was something completely unthinkable in the '70s and '80s. We barely had the internet! Now, not only is it thinkable, but it is actually being done. If you write some Rust code, and you post it to GitHub, and you add a test suite to it, then that test suite will be run by Crater. If it fails, then you are going to get some sort of notification. That is a sign of maturity for Rust that is very new. I am not sure if that is mature or not, but that is the way that Rust is measuring maturity. The definition of maturity is kind of immature itself.

Suzanne: Well, and that is fair. The importance of that, though, is what you talked about before. You do not want to invest in a language like Rust, write a bunch of code embedded in the hardware, and then have it change every three months just because somebody decided that *this* is better than *that*. *And now my code does not run anymore, and will not compile anymore, because this construct has been taken away.*

David: Precisely.

Suzanne: There is a level of, you mentioned stability, right? There is an

aspect of maturity that relates to the stability of the language. Is it still in a state where it is not very stable and it is changing all the time because people are still learning what works and does not work for the language, or has it matured enough that I feel safe if I write something in Rust, it is not all of a sudden going to break in three months because the language has changed yet again, right?

David: Right.

Suzanne: Even though you may not have a strong academic definition of maturity, the concept of maturity is one that people can get in relationship to a language. What I am hearing you say is that Rust may not be as mature as the C language, but it is mature enough that people can consider using it for real-world projects, not just for model problems. I see Garret enthusiastically nodding his head to that. Did you want to comment on that, Garret?

Garret: Yes, actually to expand from what David was talking about with maturity and what you are saying about feeling comfortable at writing new code in this, that is essentially what got us into looking at this. Or certainly myself, was the catalyst for looking at Rust, because the language and the ecosystem as a whole has matured so much. The language itself has matured, the compiler has matured. There is now a whole crates.io website, where you can find libraries and better, fairly well-developed and maturing, to support many different types of programming.

Because of this level of maturity, you are seeing businesses become much more interested in using it for their own applications and also open-source projects using it for their applications. This has grown into, for example, the Linux kernel itself, which is such an important piece of software for the entire global internet. It is at the heart of so many devices and technologies and all.

The kernel itself now is creating a project to support Rust code, which is very important because the Linux kernel standards were always focused on C. In fact, C++ and other languages were always frowned upon for various reasons. So integrating a new systems-level language like Rust is a really big deal into the Linux kernel, which is going to expand its exposure in the world. This moment of maturity where it is becoming part of such mission-critical software is really the drive, the alarm system. Well *alarm* is maybe not the right word, but reminding us that this has really become such a mature thing that we need to start looking at it very seriously like we would for C or C++ or any other type of software code.

David: Right, the fact that about 20 years ago, Linux considered allowing C++ code to be added into the kernel and rejected it, mainly because C++ did not have a stable ABI, which is application binary interface. The fact that Linux is now as of version what 6.1, Is now allowing kernel code to be written in Rust, it is a positive sign of maturity.

Suzanne: Yes, that is a significant sign that this is not going away. It is not going away, which means, Garret, that we really need to understand what are the vulnerabilities in Rust? It may address some of the vulnerabilities in C, but what are Rust-innate vulnerabilities that programmers need to be aware of and to be cautious of?

Garret: Rust, as has been discussed, one of its primary features is to try to improve memory security by having a certain memory security model that it enforces through the compiler, and features of it, for example, called the borrow checker. One of the most important aspects of Rust is the concept of ownership of memory and borrowing access to it in different pieces of your software. That model provides a lot more protection from common things, particularly toward C where there is no borrow checker. You have to be very careful what you are doing. Rust is very opinionated; it is trying to push you into more secure coding by default. But that model has certain limitations and does not necessarily cover all aspects, and so we can look for types of vulnerabilities that exist out there.

One way that the global internet community looks at vulnerabilities is we catalog them in the CVE [common vulnerabilities and exposures] system. For folks that are familiar with CVE, it is a globally unique identifier for vulnerabilities that exist in software, where it can pinpoint the type of vulnerability it is, what software it is in, what version of the software it is in, and what the fixes are ideally. That flows into a bunch of both public- and private-sector processes to make sure that our computers and networks are safe. When you look in the CVE catalogue for Rust-related vulnerabilities, there are actually already several hundred of them that are already catalogued, exist, and have been looked at, researched, and…

Suzanne: I want to stop you there, just for a minute, because the way in which, if I heard you saying that and I do not understand the CVE system, I might go, *Oh my god, there are hundreds of vulnerabilities already!* But, compare that hundreds to what, like how many are there for C?

Garett: Yes, good point, yes.

David: Have we reached 100,000 yet? CVEs, Garret? I forget?

Garett: Yes, I am not sure.

Suzanne: So in the larger scheme of things, the number of vulnerabilities are growing, and it is new, so that is going to grow as well. You should not be alarmed as a potential user, because there are several hundred vulnerabilities already logged, is that a correct statement?

Garett: Yes. There are examples of vulnerabilities in software that uses Rust that have been catalogued in the CVE system, and we have identified a couple hundred of them in various software. Considering how much software exists in the world, literally tens of thousands of CVEs are assigned every year. So it is not necessarily a large amount of the vulnerabilities that exist out there, but it is a sign that Rust's memory model is not perfect, that there are possibilities where vulnerabilities can creep in. Even though it does a very good job at trying to prevent certain types of memory issues, they can creep in. You can end up with certain kinds of vulnerabilities related to memory with like double-free's and things like that was I think one example we looked at in our blog post.

There are examples of vulnerabilities out there, and these vulnerabilities are not necessarily even in software that is in a lot of use. I think many of them were in lesser known libraries, for example. So all of this has to be taken into account, and that is part of the whole CVE security ecosystem, looking at how severe are vulnerabilities, who might be impacted, how widespread is the impact, all that stuff, there is a lot that goes into it.

But from a development-and-analysis perspective, it tells us that more work needs to be done in order to better understand the consequences of the Rust memory model and how its borrow checker works and if there are problems with the borrow checker that we can improve in the future. And either way, improve our binary-analysis tools so that we can better understand these vulnerabilities and find them after software has come out so that we can, of course, engage in all of the appropriate processes to fix them and get fixes out to the public.

Suzanne: Are you perceiving that there is an active community that is actually trying to build some of the kinds of tools that you are talking about yet, or is that something that is sort of on the *We are not as mature with Rust*, in terms of having a lot of toolsets that try and help us do that analysis?

Garett: There is a pretty mature community and many researchers that are looking at the binary-analysis and code-analysis aspects. But I think up until this point, a lot of that focus has been on the most common softwares that exist, which would be code that is written in C or C++, especially for like Intel X86 architectures. Now that there are more devices—cell phones using ARM processors and all—there are more architectures to think about, and now there are more languages to think about. Of course, those languages get interpreted to the architectures, so we have this expanding view of code analysis that goes beyond its original research aspects, which is what makes all of this very interesting. How much of those assumptions in our analysis were actually assumptions that were based on the way the C or C++ standard works, especially with how compilers translate it to, for example, the Intel architecture? Maybe not all of those assumptions are actually valid for general binary code.Those are the questions that our blog post starts to look at.

When you look at the Rust language standards that are being built up, they have made some very conscious decisions in the Rust language to actually break compatibility with C in certain aspects, for example, the name mangling that compilers do. At first they actually investigated looking at making it compatible with the C standards. Then they decided over time that just was not really long-term viable because of all the new features that are in the language, the modern features that provide security. So there are breaks in the way that various program-language concepts are represented in the binary code, from the ways Rust does it compared to the way C did it and all the C-related languages, the C-inspired languages, that are around that whole ecosystem that tried very hard to keep compatibility.

Suzanne: Oh, I see. OK. David, you had something you wanted to add?

David: Yes, I would like to add something to Garret's explanation of the number of CVEs we have in Rust, and that is that partially this was an explicit decision made by the Rust community. Where, as we said, Rust has a borrow checker, which helps to enforce the memory safety, and they suddenly decide that any bug they had in the borrow checker merited its own CVE. This was their way of taking security seriously, trying to generate faith and make the language mature. But that means that, that should help to make sure that the borrow checker is bug-free itself because bugs become CVEs and become public embarrassments.

Suzanne: I take your point that they are not trying to sweep them under the rug. That is the thing that engenders trust. When you have a language that

advertises itself as being a language that is security-centric, you have to take those seriously or else you will lose trust very quickly.

David: In the short term, you can say it engenders mistrust, because, *Hey, all of these CVEs!* But in the long term it does engender trust, because we are at least being open and honest about the stability and maturity of the language.

Suzanne: I see a few potential areas of research coming out of what you guys are talking about: the SEI actually getting engaged and building analysis tools, the SEI evaluating analysis tools, and the kind of ecosystem that they support; also helping the Rust language, as you said, researching the memory-safety model that is used. What are you looking at as future research in relationship to Rust? Or David could start.

David: I was going to simply suggest that we started scratching the surface of static analysis, both of Rust's source code and of binary code. Because as Garret noted earlier, Rust binaries, while they support more memory safety than C does, they often change things. For instance, they do not use the same name demangling that C does. They compact their structures, their data structures, differently than C, and that can confuse current binary-analysis tools like Ghidra.

We have been using Ghidra, which is mainly for binary analysis. It was released as open source by the NSA in 2019, and it is a popular software for research. Ghidra has a harder time with Rust binaries than with C binaries. That is something that can be easily fixed. It is just a bit of engineering work that needs to be done.

Suzanne: What about for you, Garret? What do you see as some of the research opportunities here?

Garett: Yes, I agree with what David was talking about there, that there are ecosystems of tools that exist that look at binary-code and executable-code analysis that we mentioned earlier. But because of the changes in the way the Rust language works and the way that it is represented in binary code, many of those tools need to be improved. Sometimes those improvements can probably be relatively easy, just engineering sort of decisions, just kind of throw people at it, but sometimes there might be more sophistication behind it. One example that comes to mind is right here at SEI, our CERT Coordination team has the Pharos binary-analysis framework for example, which is a number of tools to help reverse engineering and help analysts do reverse engineering. One example is a tool known as OOAnalyzer, and it is

called that because the OO is object-oriented. It is designed to recover object-oriented code. But from this C++ perspective, from understanding how C++ classes and types and all are generally representative of binary code. As we were mentioning, those assumptions are not necessarily true anymore, and they would need to be updated for Rust.

Rust's type system is much more sophisticated than what you see in C++. There are some theoretical challenges there, too, not even just engineering challenges, where we need to better understand the typing system as a whole and ask questions about how much of that type system can be recovered. Because the compilation process itself loses a lot of that information from the source code when it moves to binary code. I think there are a lot of interesting challenges in reconstructing that and decompiling.

The Ghidra tool, for example, that David also mentioned is a really great tool and it has built-in disassembly and decompiler and all, but again, the decompiler is focused on C code. Making it an accurate decompiler for Rust, I am not entirely sure what all would go into that, but those are the interesting research projects.

We have our own extension for Ghidra that is known as Kaiju that is meant to start building up more binary-analysis tools, similar to Pharos, but whereas that is built on the ROSE compiler framework, this would use Ghidra. Again, making those tools work properly with Rust code is all interesting open research questions that I am interested in getting into.

Suzanne: Which takes us back to the maturity question, because you are not going to invest in that kind of research for tools that you do not perceive are going to persist, for languages that you do not perceive will persist. The fact that people are taking on some of these larger research challenges related to Rust is also an indicator that it is going to be around, and it is going to be in use, so it is worth actually investing in some of that research.

David: And we will have to investigate Rust's binary state, to see if there is any malware or surprises in them.

Suzanne: Yes. Surprises! We do not like surprises in code. We like surprises at birthday parties, but not in code.

Garett: And actually if I could add on real fast…

Suzanne: Sure.

Garett: David had just mentioned, part of the goal here of looking at Rust code is because in our reverse-engineering work we are looking at things like malware. We are trying to understand what the software is doing and how we can counter various things and improve our security. As mentioned in the blog post, we have been seeing this uptick in malware that comes from these malware packages and all that are increasingly adopting other languages outside of the C family. Rust is starting to be used a lot more, but there are some other languages too. As we see more malware taking advantage of vulnerabilities, and that code itself is written in Rust, and then we take advantage of Rust code, it is important for us to understand the Rust ecosystem and other languages too, how all of the software plays together.

Suzanne: I am not sure that I am happy that Rust is starting to be used by the malware community, I guess it is a sign of maturity, but it is also—ugh!

David: I think I would be more unhappy if it was being ignored by the malware community.

Suzanne: Oh, OK, all right.

David: Say what you want about them, but they want their software to work just like you and I want ours to work.

Suzanne: Yes, yes. If they are putting their faith in Rust…I get you.

To finish off our discussion, for people that are out there, security analysts, programmers who are more interested than they were when we started on security vulnerabilities related to Rust, what kinds of resources are there besides your blog post in helping them understand where they need to look for those vulnerabilities and so on?

David: That is a good question. For what it is worth, I have read a lot of information about Rust, about how to program in Rust. In our first blog post, we, Joe and I, talked about some of the resources you could use if you want to learn Rust.

I have not read much in the way of Rust from a security standpoint. Or rather, I have read about how to write good Rust code, but I have not read about how to keep yourself secure when running someone else's Rust code. That is some material that we still need more information on, and that is one of the reasons that we wrote this blog post. I specifically asked Garret to

write what we know about source and binary analysis in Rust, because we have not really seen a lot of material out there. I hope this blog post starts the ball rolling and has people focus on this aspect of Rust because it needs more love.

Suzanne: OK. Did you want to add to that, Garret?

Garett: Yes, there are some tools that we highlighted in the blog post actually that are the beginnings of some static analysis of source code, Rust source code specifically. But these are all research tools, still being worked on academically, so they have not been expanded enough to be able to go out to developers and more general use. There are some tools to look at but they need more development, and that is still primarily in the source-code area. Like I said, in terms of binary analysis, in terms of the executable-code analysis, there really are not a whole lot of tools out there, even in research, that have been designed and updated to work on Rust code. This is definitely an avenue of future research that SEI and others can take. We have our own tools that I mentioned at SEI, like Pharos and our Kaiju tool extension for Ghidra, which tackle a lot of these problems but have not expanded into Rust yet. I think those are future things that we will look at.

David: This is the domain where we could use more maturity. To be fair, C did not get these kinds of maturity either until early in the '90s, when people started to take security seriously.

Suzanne: I see more podcasts in our future, because I think this is an exciting…There are a lot of things to do here. We are in the upswing of the interest in Rust and how to make Rust more secure, how to use Rust in a secure fashion. I look forward to other conversations we are going to be able to have in the future about this.

I want to thank you both for taking the time to talk with us today and to give us that perspective on both Rust maturity and on some of the vulnerabilities that we need to be aware of when we are actually out there using Rust in the wild, as it were, so thank you both.

I need to tell our audience, to remind them that you can get this podcast wherever you want, wherever you get your podcasts: if it is Stitcher, if it is Google, or my favorite, the SEI YouTube channel. It is available lots of different places, so we encourage you to find it. We will have references in our transcript to our prior podcasts related to this topic as well as lots of other resources that we have talked about here: Ghidra, Kaiju, etc. Please look at the transcript and

find those resources, and go forth and use them. And as David said, *Go forth and sin no more*! Thank you all very much for joining us, and we look forward to future conversations.

David: Thanks for having us, Suzanne.

Thanks for joining us. This episode is available where you download podcasts, including SoundCloud, Stitcher, TuneIn Radio, Google Podcasts, 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 do not hesitate to email us at info@sei.cmu.edu. Thank you.