Safeguarding Against Recent Vulnerabilities Related to Rust

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 Software Solutions Division. Today, I am joined by David Svoboda, a software security engineer in the SEI's CERT Division. In this podcast, we are going to take a look at some of the most recent security vulnerabilities in a programming language called Rust. We have spoken about this language before, but now we are going to get into some details. David, thank you for joining us today.

David Svoboda: Hi, it is good to be here, Suzanne.

Suzanne: I do want to welcome you back to the podcast. We have had previous podcasts with you, but for those that have not met you before, can you tell us a little bit about how you ended up at the SEI and the kind of work that you do?

David: How I ended up, you make it sound like an accident.

Suzanne: Well, you know, for some of us it is.

David: Yes. Well, I first joined the SEI back in 2007 in the secure coding group. And at that point, I had been a programmer for about 18 years professionally but joining the SEI, I learned about secure coding and software security. I knew at that point that there were certain programming things you could do like buffer overflows or command injection, lots of things that were no-no's but to me, security was kind of an abstract thing. It did not really, it did not really affect me. It did not affect my code. Most of my code was research code. It was all really used to write academic papers and sometimes used by private corporations who would use the code, but they were not looking for vulnerabilities in it. Anyway, joining secure coding, I underwent an epiphany where I discovered that, you know, having buffer overflows or other bugs in your code are not only bad for software, but here is how you can actually hack them. It is actually not that difficult to hack a program and make the program do whatever you want instead of what the developer wants it to do.

Suzanne: Right.

David: That has kind of become my mission ever since then, to raise the level of security in software. Fortunately, I am not alone because in Rust I sense a kindred language. They also want to raise the level of security by promoting, well, secure coding techniques. Before that, of course, our secure coding has always been how to write secure code in C or Java or, you know, the other languages.

Suzanne: And those languages are not really tuned to be secure. They were not designed to be secure, and that is really the difference with Rust, Rust was designed with the idea that we want to build security into the language. Is that correct?

David: That is correct. I am sure that the people like James Gosling would say, "*Hey, I designed Java to be secure*!" In particular, Java is what we call memory safe, which means that while there are shenanigans you can do with the language, you cannot normally corrupt its memory. You know, if it has this memory designed to hold your password, it is not going to give that memory to anyone else. It is not going to actually free it. And so, it provides a degree of memory of, well, memory safety. You know, it will not give that memory to other programs, but C is not memory safe. It can do that. And it has done that in the past. The problem is that memory safety is expensive. It makes Java slow, and Java takes more memory to do the same things that C

takes. So far, we have had this dichotomy where there are some languages that are C that are memory fast and unsafe. And in the other camp, there are languages like Java and Python and JavaScript that are memory safe and slow. We have not had a language that is both memory safe and fast until Rust. That is what makes Rust special. It promises the speed of C while promising the same memory safety of Java.

Suzanne: But memory safety is only one thing, and you have spoken to us before about vulnerabilities, not just in Rust, in other things. But we have talked a little bit about vulnerabilities in Rust. We are going to link to podcasts and blog posts so people can get the background on that. But beyond what you have already said, can you give us a little bit more of an overview over Rust as a programming language? I know that the memory safe plus speed makes it popular, but are there other aspects of Rust that make it popular right now? Because it is gaining in popularity.

David: Yes. I am tempted to say, no. I tend to think that every programming language out there is really one specific technology that is special. But programming languages are supposed to cover many different things, you know, managing memory, managing input and output, managing disk access, things like the web, networking, music, audio, video, lots of things. Most languages just kind of take the best of other languages and they add their one special secret to a mix of which is largely like other languages. Java, for instance, lifted its syntax from C and its magic thing was its own interpreter, which guaranteed the memory safety. Python is another one that came out about the same time. It is an interpreted language that totally abstracts away, you know, C's unsafe things and replace them with safe things. Rust's special secret sauce is its borrow checker that enforces memory safety at compile time, so that the memory safety adds no runtime performance overhead. Now, aside from the borrow checker of Rust, the other things that it provides are mostly the same things that you get in other modern languages like Zig or Dart or some things from Python, the other modern languages. But the borrow checker is the one thing that makes Rust unique and special. I would argue that that is the main thing that makes it popular because you can write things like web browsers in Rust that you cannot write in Java or Python because they would be too slow.

Suzanne: Got you. Despite the growing popularity, your most recent blog post details some new security vulnerabilities in Rust. This is really what this podcast is about. What are those security vulnerabilities, and how serious are they?

David: Right. I was hoping that this would be a podcast talking about Rust. And if I am going to talk about these vulnerabilities, I have to talk about a lot of things that have nothing whatsoever to do with Rust. The short story is both vulnerabilities were effectively in the C language, in projects in C, and Rust provided APIs to use them. The vulnerabilities, therefore, could be accessed from Rust. What we noticed is the second vulnerability called BatBadBut—that is a tongue twister there. I did a Google search just for BatBadBut and discovered that five of the hits claimed that it was a vulnerability in Rust. And they did not mention the fact that this vulnerability also affected JavaScript and PHP and Haskell and several other languages, too. There is focus on Rust, but the vulnerabilities are not specific to Rust. They do affect Rust, but they affect other languages too.

Suzanne: I see. OK. All right, and how were these vulnerabilities discovered? I know people like to know some of that kind of stuff.

David: Actually, the vulnerabilities, they were both discovered in around late March and early April [2024]. The first vulnerability which has a CVE identifier of 2024-3094. This is a vulnerability—actually, every time I mention to my friends about this vulnerability, they say, "*It is not a vulnerability, Dave. It is a backdoor*." It is a backdoor, and there is a whole human-interest story here because effectively, the XZ utility or library is part of our critical infrastructure. I have asked Dan to show up a diagram that shows its role in our computing, in our whole computing methodology. The fact that it is maintained by one single volunteer who was overworked and did not really have much effort to put into it. Effectively what happened was that a stranger came in and offered to help with supporting this language, and over a couple of years, they earned the trust of this singular maintainer. Eventually, they now became the primary maintainer, but they allowed this backdoor to seep in the vulnerability, and it is more malware. It is more that xz can now create a back channel to an SSH server so that you can send remote commands to a Linux server that is vulnerable and take over the Linux machine just because they have had the xz library installed. It is malicious because xz is an open source project. It is on GitHub. The source code is all there. However, the vulnerability is not there. If you just git clone the source code, you are not going to get the exploit. If you download one of the release tarballs, you get the exploit. Usually, a release tarball is just simply a packing up of the source code. The process is usually transparent. But in this case, they just add this extra sneaky file. Bottom line is, there was malicious intent. This is not some programmer being careless. This was somebody being evil. If I had a whole hour, I would talk about how this was done. And there are a lot of people who know more about than me. But yes, if we do that, we are never going to

get to how this involves Rust at all.

Suzanne: Obviously people who use the xz library need to be on the lookout for this vulnerability depending on how they download it and everything. Who else should be on the lookout for this and the other second vulnerability? First off, what are some ways to mitigate them? How do you make sure that you're not affected by them?

David: The xz vulnerability, of course, there is a completely Rust-less—is that a word?

Suzanne: It is now.

David: Yes. There is a completely Rust-free way of thinking of this because it is a C vulnerability, or it is a vulnerability in xz. That's a problem if xz is on your system. If you have a standard Linux system, then you probably have xz involved. And, you know, if you have ever had to download the Linux source code, it is compressed using xz. So, you need xz to uncompress it. That is not a vulnerability in the Linux kernel. That is just simply how popular xz is. It was used to compress the Linux kernel source code, so if you want to download it, you need xz. The usual answer, of course, for if you are a user and you want to not *get pwned* is you make sure you are using correct versions. My blog post simply suggests that, let's see, so there is a library. There is a Rust crate that simply links in the versions of xz. Version 5.4.6 was good. Versions 5.6.0 and 5.6.1 were both vulnerable. Those are both bad. You want a newer version, or you want an older version of those. Someone produced an API for xz so that you could invoke xz from Rust. They created a crate for it called liblzma-sys. I think liblzma was the former name of the xz project which is where they got the name. The Rust *crate and crates* is just simply the Rust name for their packages. There is a website called crates.io where the Rust crates are published. Anyway, the Rust crate was first published on September 23rd last year [2023]. They spent six months releasing new versions of the crate as new versions of xz came out, and there was never a problem. But when the vulnerable version of xz came out, then two versions of liblzma-sys came out, the versions 0.3.0, 0.3.1, and 0.3.2, and these ones were vulnerable. These ones had the malicious test file. You know, somebody, whoever built the crate did not just simply git clone the xz code, they actually downloaded the release, and so they got the malware embedded inside it.

Suzanne: Gotcha.

David: In fact, thinking about it more, this is less a problem with Rust and more a problem with crates.io because crates.io was hosting this crate that supported malware. And they did not know about it because the xz code is not in Rust. They have a secondary problem in that since it is malware, they do not want people using it. But some people are probably already using it. And if they take it down, they can create something like the left-pad fiasco. The left-pad fiasco is another thing I could spend an hour on if you let me.

Suzanne: All right. You got two minutes.

David: Yes. Oh, thanks. But basically, it was a bit of JavaScript code in JavaScript's own archive which they call npm, Nicholas Pueblo Mike, and basically, if you wanted to use it, you could just simply say in your web page, import left-pad, and then I look at your web page and your JavaScript runs just fine. What happened is that the originator of left-pad got angry because of politics or drama or something, and he just simply removed left-pad from npm. And suddenly, when I look at your web page, your web page says, *Error 404, left-pad not found*.

Suzanne: Cannot find it. Yes.

David: Yes. Cannot find it. The people who maintain crates.io had this conundrum. Do they leave this malware up so that people can access it? Or do they take it down and secure crates.io? And the answer they did was they yanked it. By yank, what I mean is they did not actually remove it from crates.io. They just simply forbade people from using it if they are not already using it. If it is already in your web page or your software, you can keep using it. But if it is not in your software, then it will not turn up in your searches. You will not ever know that it exists. That is kind of their way of securing, securing . . .

Suzanne: It is securing for the future, but it is not protecting the people that might not know that they have a vulnerable version of xz . . .

David: Right. The best they can do there is hope these people see, read the news, and discover that they are using a vulnerable version. And then they can either download to a version before 0.3.0 of yank, or they can upgrade to 0.3.3 and there may be a newer version since then. That is just what I have in the blog post.

Suzanne: OK.

David: It is not so much a Rust vulnerability as it is a supply chain vulnerability. One language simply took software from another language that had malware embedded in it, and no one knew about the malware when it was done.

Suzanne: Right.

David: This is the kind of thing that is going to happen all the time.

Suzanne: Because it is accessed from Rust, as you have said, some of the press and whatnot out on the web attributes it to being a Rust vulnerability.

David: Exactly.

Suzanne: But it is really a vulnerability of something that Rust accesses. What is your current view of the security of Rust? It has been about for about nine years since its first stable release. How mature is it? And are you feeling comfortable with the way that they handle security issues and security vulnerabilities that are found like this?

David: I discussed this a lot in our Rust security and maturity blog post, which you will provide a link.

Suzanne: I will link to that.

David: Yes, in where I was simply asking about, asking how it compares to other languages. And what I realized is simply that I do not know how to rank the security and maturity. We can say how old the language is. Rust is clearly a lot younger of a language than C but that does not mean it is less secure. It does not mean it is less mature. It depends on how you look at things. And what I discovered is that C first came out in the '70s and went down a path towards maturity which is very different than the path that Rust is going toward maturity. Simply because our notions of a mature language today are very different than our notions of mature language in 1980. C mainly went down a more bureaucratic path of establishing a standards committee, of establishing processes for updating it. Rust is more going down a technical path of having a foundation to maintain it, to provide the documentation, having a foundation to maintain its archive, doing extensive testing on Rust crates, and whatever public Rust code they can see in case they make a new change that might possibly break code. They can preview if any tests break before they publish a possible incompatible change. Rust is going in a good direction for maturity. I cannot say if it is done. Well, no. No language is ever

done with maturity. It is certainly ready for more widespread use, whether if you want to try writing your operating system currently in Rust, you know, please go ahead and try. And if it is successful, then I might consider using it.

Suzanne: There you go. All right, as an FFRDC, a federally funded research and development center, we always ask about transition in our podcasts since that is part of our mission. For people that are basically saying, *I am ready for Rust, you are actually giving me some, you know, comfort that even though some of the stuff I am reading on the web blames Rust, it is really not necessarily Rust at the root*. How can they learn more about security in Rust and how to get started using Rust if they have not used it before?

David: Right. Ultimately, what you are asking is a bunch of resources. And of course, those are links. You can surely give them the links to, you know, this podcast and the corresponding blog posts and our previous two blog posts about Rust. The first one was from a programmer's standpoint. If I am a programmer, then how does Rust make me secure and how does it fail to make me secure? Obviously, Rust can fail to make you secure from things like xz, malicious versions of xz, but that is kind of out of this scope. You cannot really expect Rust to cover those kinds of things. Similarly, rust-lang.org, created by the Rust Foundation which provides extensive documentation first about how the Rust language is put together. There are also tutorials. I went through an excellent tutorial there where they basically gave you like 50 incorrect Rust programs saying, "*This program does X, change it so it does Y or this program does not compile. It has a bug in it, please fix it*." And then it would monitor and make sure that you compiled the program and the program behaved accordingly. It was a very useful tutorial, and it was completely automatic. They have lots of reference documentation about Rust. And they also are associated with crates.io which has all the various Rust crates you can use. Unfortunately, the best we can say in order to remain secure is to keep your ear to the grapevine and notice when people announce vulnerabilities in Rust like the xz vulnerability.

Suzanne: OK. All right. That makes sense. And it sounds like at this point, it is not going to be difficult to get involved with Rust. It has come far enough along that these resources are available publicly and you do not have to sort of really dig for them. Is that correct?

David: Oh, right. It is all public. It is all free. It runs on all three platforms so you can develop no matter what your computer is. At least, I mean, I did this development on my Mac, so that works. Yes, when I first wrote these things, I did not know Rust at all. I did not know the language, but it did a very good

job of clueing me into how the syntax works. I was able to write some simple Rust code. I did not write a lot of Rust code. I mostly debugged code partially from the tutorial.

Suzanne: Sure.

David: But it is ready for beginners to start using it just to learn how it works, and there is certainly lots of open source code that you can work on. There are obviously other libraries, you know, I think along the lines of xz, but there are other libraries that you can write Rust APIs for. Because at this point, the biggest job is giving interfaces, Rust-type interfaces, to the wider world of open source software. There are lots of projects to work on if you're looking for something to do.

Suzanne: What are you doing next? What are you working on now?

David: What I am working on now, unfortunately, has nothing to do with Rust. I hope to come back to Rust soon. What I am working on now is automatic code repair. You may remember us talking about it, and we should be releasing a blog post on it real soon now as well. This program is actually specifically for C and C++, where we have a program that will repair your C program if you have vulnerabilities in it.

Suzanne: Yes, and we have talked about how valuable that is going to be, and we will refer to that podcast as well.

David: Yes, that will be helpful. And actually, that program is open source as well. It is available on GitHub, so you can try it yourself.

Suzanne: Great.

David: But it specifically works with C and C++ programs, unfortunately not with Rust. I suspect that repairing Rust programs, it is going to be hard in one sense and easy in one sense. The one sense in which it is easy is it should be easy to modify Rust code because Rust has a pretty simple clean syntax. The hard part is simply finding out what kind of errors people will make in Rust, especially errors that might make Rust vulnerable.

Suzanne: Gotcha. Yes, and that you are not going to get until you see a lot of examples of Rust . . .

David: Yes, until we see examples. Most errors are best determined by just

simply seeing examples of what people do wrong.

Suzanne: Yes, unfortunately. All right. Well, I want to thank you for speaking with us today, David. It is always fun to talk to you about some of these more technical topics and to get your perspective and to help people. I really think these podcasts help the software engineering community stay up to speed with what kinds of things they should be looking for. Thank you for taking the time with us today.

David: Oh, sure. You are welcome.

Suzanne: For our audience, we spoke about a lot of different things in this podcast. We will have links in our transcript to all the resources that we've mentioned during this podcast, including the blog post on this topic and other topics that David has mentioned. And a final reminder that our podcasts are available pretty much wherever you find your podcasts. Our favorite, of course, is the SEI's YouTube channel. If you like what you see or hear today there, give us a thumbs up. That is always appreciated. I want to thank all of you in the audience for joining us. Thanks.