Temporal Memory Safety in C and C++: An AI-Enhanced Pointer Ownership Model

Featuring Lori Flynn and David Svoboda as Interviewed by Tim Chick

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.

Tim Chick: A critical security vulnerability has been discovered in Redis server that allows authenticated attackers to achieve remote code execution through the use-after-free *flaw* and Lua scripting engine. The vulnerability CVE 2025 49844 poses a significant risk to organizations relying on Redis and in-memory safe data storage. In today's podcast, we will discuss recent updates to the pointer ownership-model to address this and other types of temporal memory safe vulnerabilities.

Welcome to the SEI Podcast Series. My name is Tim Chick, and I am the technical manager of the Applied Systems Group within the CERT division. Today joining me is David Svoboda, who is a senior software engineer and Lori Flynn, who is a senior research engineer. And thank you both for joining us.

David Svoboda: Thanks, Tim. Good to be here.

Lori Flynn: Great to be here.

Tim: For those who are new to the podcast series, we always start off with, kind of: What have you been working on? What's interesting? a little bit of background, and what really brought you here to the SEI to begin with. Lori, why don't we start with you?

Lori: Sure. I am interested in researching novel ways to automate software security and functionality. And so here at the SEI and previously, I have focused on areas that involve static analysis and AI (or, LLMs and machine learning) and malware analysis and cybersecurity.

Tim: Great. So, the research brought you here, the research brought you to Pittsburgh or the SEI?

Lori: The research brought me here, yes!

Tim: Dave, how about you?

David: Well, Tim, I have actually been a professional programmer for several decades. I did most of my initial work with C, and I discovered that C has some security problems. I didn't really worry about them so much until I joined the SEI and discovered secure coding. Secure coding had a major effect on me when I realized that buffer overflows are a real problem, and it's really actually not that difficult to take over a program that has problems like buffer overflow or temporal memory safety problems.

I have been teaching secure coding for a while since then. Many students had the same epiphany. It's a lot like discovering that you have been a sinner all your life, and the best you can do is go and sin no more. I have been focusing on trying to improve software security and make software more secure, make people less sinful.

Tim: That is a great analogy. We are going to talk about the Pointer Ownership-model, which isn't a new concept, but we have some updates. LLM, the new technology, really enables some of that. We will talk about that. Give us some of the background about the concept.

David: All right. So, Pointer Ownership-model is an example of either a formal or informal model. Formal models have been an aspect of

programing and computer science since I was a student. When I first saw them, I thought they were interesting, but they have two big problems. It is hard to build a model because it is a lot like building a mathematical proof.

It is like proving that this program has no bugs. So, you have two problems. First of all, how do you build the proof? And, second of all, how do you confirm that the proof is correct, and the program complies with that?

Tim: And is just time consuming.

David: It is very time consuming.

Tim: Most developers don't want to do it.

David: To do this manually. Again, it is just like building a mathematical proof, and I spent a lot of time. I was a math major, so I spent many nights trying to prove something, which…And these were the only things that were well-known and well-understood, and they were mathematical exercises. To try to prove something that isn't…is, unsolved, right now that is even harder. Ultimately, formal models are something in the research field that have…Well, they have had some adaptations, but the adaptations are usually rather quiet, and they are expensive, and they are slow. The seL4 kernel, for instance, has several formal proofs to guarantee its value. The DoD, like everyone else, wants software assurance, and usually what they get is various testing mechanisms, static analysis, as Lori mentioned, the absence of known bugs and also various techniques to hunt for bugs, but they don't prove the absence of unknown bugs. That is the payoff that formal models can provide. So, it's a big payoff, but it's also a big price. Our hope is with the updates that the price has gone down, and we can automate some of this and hopefully formal models…formal methods can get the light of day.

One other thing I learned about as a research project when I was a student was neural nets, and neural nets were another interesting mathematical abstraction. And I thought*, It's going to be a long, long time before these see the light of day as well*. And it was a long, long time.

Tim: But here we are.

David: But here we are.

Tim: Right? It just takes time and energy and technology. Right?

David: And lots of work.

Tim: Lots of work. Lots of energy. I mentioned temporal memory. It was the catalyst for this particular model. Can you give us a bit of background about that and explain that.

David: Memory safety can be divided into two things: temporal and spatial memory safety. What people are used to dealing with is spatial memory safety. If you have say five elements in a row: *boom boom boom boom boom*, and I ask you for the seventh element, that is a spatial memory safety problem because there is no seventh element. Most computers today will just simply give you whatever is in that memory, whether it is a real element or not. That is a spatial memory safety problem. Those are big problems. Buffer workflows are a good example of that.

Temporal memory safety is a case where I ask for, say, the third element in the array, but I ask for it after the array has been freed, and there is nothing there. That is again a problem. It's an example of undefined behavior in the C standard. Because a C standard says, if you ask for an object whose life has ended, the behavior is undefined. That is their way of kind of washing their hands, saying, *We make no prediction of what your computer's going to do. It could play the game of LIFE. It could publish your root password to the internet.*

Tim: Basically, whatever is in that space in memory is what it is going to give you.

David: Exactly.

Tim: Assuming it doesn't, it doesn't like, crash and like throw an error.

David: If you are lucky, what is in that space is what it's going to give you. If you are not so lucky, the compiler might say, *This is a can't-happen condition. I'm just going to do something else. It might trap it; it might stop running*. Again, the system is allowed to do anything when you have undefined behavior, and that includes temporal memory safety issues such as use-after-free.

Tim: Okay. Well, thank you. So, Lori. In the Pointer Ownership-model, this kind of the design of a new temporal safety model, the C as invitation to enforce it. Can you tell us about these updates.

Lori: So, the major changes are that we have added more automation. Now we have incorporated use of LLMs and SAT solvers to automate what

developers needed to do by hand before. So, we use LLMs to create per-program models or p-models, which previously needed to be done by hand. The developer would have to understand the POM constructs in detail and translate their code into that. It was time-consuming before, but now with the use of LLMs such as—we are using GPT with our scripts right now—but LLMs include Gemini, GPT, Claude and Llama, for instance. By incorporating those, we automate that process. After a per-program model is created, POM needs verification. Previously, that was done through static analysis, which depending on if it was coded quite right, there could be some failures. Now we use the SAT solver to verify the p-models are correct. What comes out of the SAT solver is SAT or UNSAT, whether it is, satisfiable or unsatisfiable.

I can back up a little bit and talk about what a SAT solver does generally. So, a SAT solver takes as input a Boolean formula, and it tests to see—and the Boolean formula can have any number of clauses (where there are Boolean values, *true* or *false*, or [actually there are] variables that have Boolean values) that are conjunctions that are *ands*, *and* clauses, and *or* clauses. [Better restatement: The Boolean formula is made of Boolean variables connected by logical operators (like AND, OR, NOT) as input.] The SAT solver determines if there is any possible assignment of values of true and false that can make the overall Boolean statement true. If so, it outputs SAT, it is satisfiable. If not, there is a problem with, in our case, the p-models.

David: The per-program models.

Lori: Yes, so if an UNSAT is output by the POM SAT solver, then POM outputs some helpful kind of pointers, that help the developer to modify either their p-models or their code or both.

Tim: Let me get this right, which is actually kind of one of my questions. so LLMs are great, but sometimes they hallucinate and stuff, right? So, the SAT model is what you are using to validate both the LLM;s output, whether it maybe made a mistake, as well as the developer ‘s code.

Lori: Absolutely. Yes. And, since LLMs sometimes do hallucinate, sometimes those p-models may be wrong. Also, developers sometimes make mistakes.

Tim: Either side of the equation, but the SAT solvers is helping to identify that there is an issue. Then, obviously, an engineer or developer has to figure out which side of that equation it is.

Lori: Initially just kind of the raw output of the SAT solver that we are using, it

would just provide a bunch of clauses and the steps that it took, that led to the UNSAT result. But, we have incorporated use of this DIMACS SAT solver after our initial SAT solver; that reduces the set of clauses to a set that is smaller. We have developed some scripts that connect the variables used in the SAT solver (by the SAT solver) to particular lines of code and variables, both in the source code and in our LLVM intermediate representation. So, that should help developers to...

Tim: To tell them where to look.

Lori: Exactly. Yes.

Tim: All right. We can't cover everything in a podcast. There is a technical report out. I think just today or yesterday a blog post was just posted.

David: Yesterday actually.

Tim: So, a couple of other resources. We will provide links in the transcript of this podcast for those who are interested. What other resources are available in terms of adopting this and doing this?

David: There is a GitHub project. I forget if it is called POM or Pointer Ownership Model- we'll provide the link there. And that has basically all of our source code. That will be linked from our "Collection" page. We have a single Collection page which links to the blog post and the GitHub, and it will link to everything else related to the Pointer Ownership Model project. So, watch that link for developments.

Tim: Okay. So, in that technical report you talk about, I believe it's called CHERI. I wasn't sure whether it was SHERI or CHERI, so I actually watched a couple of YouTube videos, and it sounds like CHERI is what they call it, and the POM. They kind solve the same thing but very differently. So maybe Lori you can explain the difference?

Lori: Yes. CHERI requires special hardware to be able to provide the excellent level of memory safety that it does. CHERI is able to trap executing code, when it identifies that there would be, for instance, a temporal memory safety violation. And then, depending on how the code is programmed, the program might stop. It might do some other action when there has been a safety violation that has been caught ahead of executing.

POM, on the other hand, uses static analysis which analyzes code without

executing it and enables developers to fix safety issues before any violation happens, maybe even before the code is deployed.

Tim: So, it doesn't require any specialized hardware, right? And it can be used during the development engineering phase, not operational phase.

Lori: Exactly, exactly. CHERI is currently in the process of being standardized. There are a lot of big companies that are part of the standardization group for CHERI. So, for instance, Google, Microsoft, Intel and AMD because chips/hardware need to be modified to make it widely available. Most people think that, sometime in the near future, maybe three years, maybe ten years, these widely available, widely used, CPUs and GPUs will incorporate CHERI. However, we think that CHERI and POM complement each other really well because of what we just talked about. How POM enables catching and eliminating the problems ahead of time. CHERI catches the problems if they slip through.

Tim: Great.

David: I will add that, yes, CHERI provides the elimination of known bugs whereas POM provides software assurance of proof of no bugs. Not all bugs, proof of just no temporal memory safety bugs. So that is all correct. I also associate the University of Cambridge with CHERI because of Peter Sewell, who is on the project and works there. So, the universities, they are also involved. They will probably have a big solution in five or six years.

Tim: A lot of software will be written between now and then though.

David: Much will happen between now and then.

Tim: Right. So, they can use this now and use that then, right?

David: Pretty much.

Tim: Or both together. What is next? We have done this. It's out on Github. People can use it. What are you guys working on that might bring you back for a future podcast?

David: Well, there is much work to be done. I could list all the things that we want to do that we didn't have time to do, but I will simply be optimistic and simply say, this is future work to do. For instance, right now, POM doesn't handle static pointers, global pointers that live in the same place in memory

throughout the lifetime of your program. That is something for us to handle. There are some standard functions that are complex in how they manage pointers. The UNIX `realpath` function is one such example. It returns a pointer that might be responsible or might not, depending on the parameters you put in. So that is something that we can handle as well. As Lori mentioned, we just did our testing with ChatGPT, I believe 4.0 [We actually used o4-mini]. There are several newer versions of ChatGPT out, and there are plenty of other LLMs we can use. We can alter and diddle how the SAT solver and LLM work. Surely having the SAT solver verify that the LLM didn't hallucinate is important, but it's not the only possible way of doing things. To me the biggest question is right now we are just handling temporal memory safety things, but we haven't done anything about spatial memory safety. Could we use the same approach to, say, detect buffer overflows without actually having to run the program and overflow a buffer? That is an important question. Lori, would you like to add anything else?

Lori: Sure. We are currently working on a research paper and just a teeny preview just as of yesterday, I calculated the results that Jeffrey got from his latest testing. And 95 percent of the Juliet test suite's temporal memory safety tests were identified correctly as good or bad with 0 percent false negatives. It was pretty exciting results!

Tim: False negatives kind of just eats everyone's time and their lunch and everything else.

Lori: False negatives are where you don't have a warning about a very real bug. So, they are very scary.

David: They kill software assurance.

Lori: Yes, so that is forthcoming soon. Also, we will be submitting presentations on our work to various venues. Those will be published to the Collection page that David talked about. Our current project funding ends this very month, but we're very interested in doing more to build on the POM work. So, we are hoping that that may happen in the future. Also, now that our team has this experience using LLMs and SAT solvers and models, we are very interested in possibly applying those skills to other interesting software security and software functionality topics.

David: It is time for formal models to have their day in the sun alongside LLMs.

Tim: So, then Lori. You actually have another blog coming out soon?

Lori: Yes!

Tim: What is that going to be about?

Lori: Let's see...Lyndsi Hughes, who is leading the other project that the blog is about, and I are talking about the use of software quality attributes and an architectural tradeoffs analysis method that was developed at the SEI decades ago. We are talking about how we are using those methods along with modern DevSecOps and a tools developed in our project, Silent Sentinel. We are using all of those together, to try to enable software developers and stakeholders to better set and meet requirements and stakeholder quality priorities.

Tim: Because not every requirement is a functional requirement. Sometimes it has to be secure. You talk about assurance cases, which is basically it only does what it is supposed to do and nothing else. It is ultimately what we want as a developer in everything that we do. So yeah, so quality attributes are quite interesting, and I look forward to that as well.

Thank you for taking time to join us on this podcast. I look forward to future podcasts. I look forward to your upcoming blog as well.

For our audience, we will include links to transcripts to the resources mentioned during this podcast as well as the blog. Finally, as a reminder to our audience that our podcasts are available on SoundCloud, Apple Podcasts, and Spotify as well as the SEI's YouTube channel. If you like what you heard, please give us a thumbs up. Thank you for joining us.