Identifying the Architectural Roots of Vulnerabilities

featuring Rick Kazman and Carol Woody as Interviewed by Suzanne Miller

----

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. Today's podcast is available on the SEI website at sei.cmu.edu/podcasts.

My name is Suzanne Miller. I am a principal researcher here at the SEI. Today, I am very pleased to introduce to you to Drs. Carol Woody and Rick Kazman, my colleagues. We are here today to talk about a tool that they have developed for identifying vulnerabilities and other issues in software architectures.

First a little bit about our guests. Dr. Woody is the technical manager of the Cybersecurity Engineering Team of the CERT Division. Her research focuses on defining, acquiring, developing, managing, measuring, and sustaining secure software in very complex network systems and systems of systems. That is a lot of stuff. Carol is very busy.

Carol Woody: Yes.

Susan: As far Rick, in addition to his research here, he is also a faculty member at the University of Hawaii. At the SEI, his research focuses on software architecture, software engineering economics, design and development tools, and software visualizations. Welcome to you both. Thank you for joining us.

Rick Kazman: Thanks, Suzanne.

Susan: Let's talk about your decision to focus on software architecture as the root cause of vulnerabilities. We have lots of focus on code. We have lots of focus on different aspects of trying to find out where the vulnerabilities in a system are. You decide to focus on software architecture. So what is [it] about that? Why did you decide to go there?

Carol: Let me give a little background. When we look at cybersecurity engineering, most folks think of vulnerabilities. Typically, those are found in the code, and those are later in the lifecycle. But, there is a major block of design weaknesses that are implemented in the system and are very hard to change later on.

In terms of overall problems, design weaknesses represent probably about 40 percent of what we have identified as general weaknesses that you are dealing with, with code and design. However, when we look at what experts have identified as the top 25 problems, design weaknesses represent over 75 percent of those. This is an important area. It is a difficult area because the way that we typically think about security, we think of it as completed systems and analyzing its overall security. Here we have to think about what is the system doing? How are we building it so that it does not have those problems when we ultimately field it?

Susan: As we all know, the later that you find these problems that were introduced in an earlier phase like design, the harder it is. This is part of the difficulty, so if we can find design weaknesses in the architecture and fix them then, then we have a much easier time all the way down the lifecycle.

Carol: That is what initiated discussions that Rick and I were trying to look at to say, *What can we do?* He will give you a little more detail about the tools he is starting to work with but, he had some pieces. He was looking at architecture, in general, for flaws. We said, *OK*, *how can we structure those in a way to find these major problem areas that we are trying to explore*?

Susan: That is a big part of the challenge is we have all these architectural patterns, and those patterns are designed to meet certain functional and non-functional requirements that we have but, the awareness of how different patterns affect security may not be as great as it needs to be within the architectural community as well as the design and implementation community. Is that fair to say?

Rick: Yes. And more than just the patterns, the ways in which people get them wrong.

Susan: Good patterns gone wrong.

Rick: Or, even patterns that they should have used but didn't.

Carol: The patterns they thought they did…

Susan: …they thought they used, but they really implemented something else [that] was not as effective and could have introduced a vulnerability or a weakness. You tend to talk about *weaknesses* rather than *vulnerabilities*. Is that a purposeful word choice?

Carol: Weaknesses tend to be the way we build things that allow vulnerabilities to occur.

Susan: So they are more of a root cause of a vulnerability.

Carol: As we are pushing up earlier in the lifecycle, we don't really have tools that can go in and say this is a vulnerability. It is more of how the pieces are being glued and designed together. I think as Rick gets into some of the details about what we are doing, you will find that we don't really have ways of pinpointing in the designs what will cause the vulnerability, but we can certainly begin to narrow the area we need to look in to help us better…

Susan: …Provide focus.

Carol: Focus and allow us to reduce the amount of deep dive we have to do to find problems.

Susan: Because that is expensive, right? Trying to look across a broad spectrum of potential weaknesses at a deep level is a very expensive proposition. Let's talk about your approach. You have got an approach that you have developed in collaboration with Drexel University, from what I understand. That collaboration has resulted in a structure that seems to have some promise in this area. Tell us about it.

Rick: This collaboration began in late 2009. We have been collaborating five or six years. I have been collaborating with professor Yaunfang Cai at Drexel University and some of her Ph.D. students.

If we just back up before that time, we had done a lot of work at the SEI in architecture analysis and developed a number of methods. We got a lot of leverage out of those methods. We could go a long way to finding potential risks, but they tended to be, what do I want to say…? The risks that we identified tended to be broad brushes. What we wanted was the ability to zoom in a little bit closer and try and find the architectural weaknesses or flaws, or we sometimes call them *hotspots*, that are the likely cites of future headaches, to use a non-technical way of describing that.

Susan: But that describes what is going to happen if one of them gets implemented.

Rick: It is going to be headaches. What we have developed over the years is a design representation and associated set of tools that allow us to reverse engineer an existing code base; extract architectural information, design information; manipulate that information in interesting ways; and then analyze that manipulated form of the information to find architectural flaws. This is something that we were never able to do before. This is all automated, which is great because all the techniques we developed before were completely manual, which meant it was very labor intensive. It doesn't scale up very well.

Susan: So in 1993, I attended a software engineering symposium the SEI sponsored. One of the researchers, one of our colleagues, Dennis Smith, was talking about reverse engineering. I went to his tutorial on reverse engineering. The big thing there is *This would be so wonderful if we could automate this* because to do this by hand, to do this manually, takes a huge amount of resource, almost impossible in that timeframe. You have actually made good on one of Dennis Smith's promises for the future, which was *When we can automate this, we can do something with it*.

Rick: That is right. We have actually crossed that bridge pretty successfully at this point.

Susan: What are some of the highlights of the approach itself, especially for people that are architects in our audience? What are some things that would help them understand what it is that this actually does for them?

Rick: Let me actually explain that by talking about the tool chain that we have developed. On the left side of the tool chain are all the inputs. The inputs are things that every project has. Every project has source code, or every *mature* project let's say, has source code. Every project uses some kind of revision control system: Subversion or GIT or something like that to manage commits and manage releases and so forth. Every mature project uses some kind of issue tracking system, like JIRA let's say, to manage their bugs and changes to the system. So all of these are inputs, and so it is more than just reverse engineering code, which is the kind of Dennis Smith vision of the world.

Susan: The dream of 1993.

Rick: Yes, the dream of 1993. We have now realized, actually, there is a lot of information in these other artifacts that augment just that code view. We reverse engineer the code, and right now, we are primarily using just a commercial reverse engineering tool called Understand. But we could, in principle, use any reverse engineering tool that will tell us about the basic facts of the system: *File A inherits from file B,* or *file A calls file B,* or *file A depends on file B,* or aggregates, or anything like that. Any system that gives us that information we could use.

In fact, we have done a version of this system where we sucked in information from enterprise architects, so we can even do the analysis on designs. This outputs a set of dependencies between files. So you can think of those as triples: file A, file B, and some relation between them

Based on that, we generate a big matrix. It is a design structure matrix. That is just a square matrix where the names of all the rows are the file names. On the columns, we have the same set of file names in the same order. So a square matrix, the diagonal is the self-dependency or self-relationship. Every cell in the matrix represents some dependency between file on the row and file on the column or no dependency if there isn't a dependency between them.

We have got that DSM, that design structure matrix. We can analyze and manipulate that. What we do is we cluster it using an algorithm we have developed. It is called the design rule hierarchy algorithm. A design rule would be something like an important interface or an important class in the system. My favorite example is if you are using Abstract Factory in your system, you are going to have a lot of other files that depend on that abstraction. There will be classes that implement the Abstract Factory interface, that inherit from it, that depend on it, that use it, that aggregate it. That abstract factory representation is a design rule, and a lot of other files will depend on that design rule. What we do is we cluster the architecture in terms of dependencies. At the top are all of the leading classes, the most important design rules in the system.

Susan: And most important in terms of the most dependencies others using that…

Rick: Exactly. The classes, or the files that lots and lots of other files in the system depend on. We would cluster that into a layer, those files into a layer. Then the layer below that would be the files that directly depend on those files. The layer below that would be the files that depend on layers one and two and so forth, so it is a hierarchy. In this way, we can cluster the entire system. Then we can analyze that hierarchy, and we can run all kinds of interesting analyses just based on the structure. For example, we can find cycles of dependencies as pretty simple examples. If file A calls file B, B calls C, C calls A, that is almost always a bad thing because when you change A or B or C, you do not really know what the ripple effects. These cycles can be a lot more than three files long.

Susan: It can be a very large cycle not a small.

Rick: It could be a long cycle and a lot of non-local…

Susan: Not-easy-to-see relationships.

Rick: Not-easy-to-see subtle changes that can happen when you tweak one file. Do you want to make a point, Carol?

Carol: I was just going to point out that that is something a programmer can’t easily find because you are looking at millions of lines of code. How do you then determine how these pieces fit together?

Susan: Typically you are working on this piece over here and this is C. There is D, E, all the way back to A, that are not part of what you have been asked to deal with. You may never see them.

Carol: They may also be managed by other people.

Rick: That would be an example of an architectural flaw or a hot spot. Another example would be what we call improper inheritance. If you have file B inherits from file A, but A depends on B. We actually see this in large numbers of systems. You think that is crazy, that it violates good design principles. The response to that is, *Yes*, and yet you see it all the time. It is essentially what Carol said: it is so easy, as a programmer, to just insert a dependency. I am just going to call that method from this method. You are not thinking about design at that point, you are thinking about getting the job done. These problems just creep into a system. They are like rust. They just accumulate. Every system has them and they get worse over time.

Susan: This is one of the reasons the revision history is important, because there could be points at which that was not an issue. Then, all of a sudden, something gets introduced in a revision that changes the relationships that might have been positive before but now create a dependency that is inappropriate.

Rick: Yes**.** Let's actually look at that. We also grabbed the revision history. We would grab, let's say, an SVN log. We would process that. If you think about the commits in a revision control system, those commits are like a dependency. When file A and file B are committed together, there is some reason they are committed together, because they both needed to change to fix some bug or add some feature.

Susan: It was perceived as they both needed to change.

Rick: Right. We call that a history relationship. They are historically related, and so we can build a history DSM, a history design structure matrix where, again, all the files on the rows, all the files on the columns and every time they change together, we put a mark in that cell.

Susan: Then can I look at *How are my history relationships reflected as actual relationships in the structural DSM?*

Rick: Bingo. That is another kind of architectural flaw. When you get files that have no structural relation, but they consistently change together in revision history…

Susan: There is an implied relationship.

Rick: There is something going on there. We looked at one system recently that changed the notion of the time in the system. Previously it had measured time in seconds and then, at some point, they decided they had to measure time in milliseconds. What happened was lots and lots of files had to change. That information was not modularized anywhere. It meant every place that knew about time had to change. We call that a modularity violation. It is a modularization that they should have done but didn't. As long as they continue to not modularize the notion of time, every time something to do with time changes, you have got to find all the places that might have to change and make commits.

Susan: This is a great example of, in say the first 20 percent of the system modernization, I make that decision. If I don't modularize it, then I have got all these other design elements coming behind that nobody knows that we have done this change. I am still writing things in seconds because that is what my spec says to do. Then somebody is going to have to come afterwards and change it to milliseconds and everything else. Oh yes, that headache, very big headache.

Rick: We sometimes called these *shared secrets*, right? And they are bad. They are always bad because you rely on tribal knowledge basically for the changes to be done correctly, and that breaks down pretty easily. *Joe is the guy that knows this, and Joe is away. Joe is on vacation. Joe is sick. Joe has left the company. We are in a world of hurt.*

Susan: All the architects that are listening to this have got to be very excited about this, but they are also going to want to know what kinds of testing have you done to sort of validate that this provides utility beyond what an architect would normally do in their own analysis?

Rick: Absolutely. The third piece on the left there is an issue list. What we can do is we can grab the issues from your favorite issue tracking system. Those are typically classified as either bugs or changes. We can associate those again with every file in the system.

What we see is once we have built these DSMs and we have analyzed them, we can now look at the correlation between the places that we can find flaws and the rate of changes, the rate of bugs, the rate of churn, which is the number of lines of code that have to be committed to fix those bugs and to make those changes. What we see is incredibly high levels of correlation. Basically, the greater number of architectural flaws a file participates in, the more bugs it experiences. The more changes it experiences, and the more lines of code need to be committed.

Susan: Every time you do that you have potential unintended consequences. It is an amplifying system.

Rick: People typically have no clue that these flaws are there. They just know that they are swatting bugs, and they are dealing with the symptoms.

Susan: They are dealing with the local effect. They have no way of understanding that there might be a global effect.

Rick: If you have got a bee hive in the corner of the room, and the bee comes by, and you swat it, the hive is still there. It is going to just generate a steady stream of bees until you figure out where the problem is coming from. That is basically what is going on in most software projects.

We can detect these problems. We can identify them, categorize them. We can rank them according to their severity. We have a GUI [graphical user interface], which allows you to visualize these DSMs and run the various analyses and manipulate them…

Susan: This has really come a long way since 1993.

Rick: It has come a long way.

Susan: If I am a program manager, an architect out in the world, government, industry, how do I make use of this? Where is it available? What have you written? Where are the tools? All that kind of good stuff. How do I get my hands on it?

Rick: Right now, the software is being licensed by Drexel. They make it available. Ask them. Certainly, for academics, it is free. Otherwise, they might want some kind of licensing agreement with them, but they have been quite open about sharing that.

Carol: We are working with DHS right now, too, to set this up so that it can be available in their software marketplace.

Susan: OK, so that would be a government source. Excellent.

Carol: If you can upload your code, not everybody can, but if you can then you can run this tool suite against it there.

Rick: That is all being packaged into a virtual machine that we are giving…

Susan: That is consumable.

Rick: Yes. We are giving them, and you can just run your code through this tool change.

Susan: What is next for this? I mean I have got all these things ripping off in my head about different ways that you might want to take advantage of this. Where is your research going, both of you, Carol and Rick, in terms of taking advantage of this.

Carol: We want to leverage this to see, *Can we pinpoint where the security hot spots are?*

Susan: Wouldn’t that be nice?

Carol: Yes, you may have a lot of architectural issues, but can we get the subset of security issues that are really going to be weaknesses that need to be addressed early. That is what some of the more recent research we have been analyzing is helping us do.

Rick: We have looked at probably about 150 systems. We have analyzed using these tools and this technique, but we have looked specifically at security for probably 10 to 20 systems. What we found is exactly the same patterns, which is to say, the greater number of architectural flaws a file is implicated in, the greater number of security bugs it experiences. Design flaws don't care. They are going to make everything worse, including security.

Susan: Security is one of the things they are going to make worse.

Rick: We think this really helps a quality assurance person or a testing person or a security expert figure out where to spend their assurance dollars, where to try and refactor or analyze or test more heavily. That is our current thrust.

Susan: I am not at all surprised that this is very exciting work because everything I have done with Carol and Rick in the past has always been things that give us some important insights. I think this falls right into that.

I would encourage all of our listeners to try and take advantage of this, especially if you have got a legacy system that needs to be analyzed this way. Rick and Carol, thank you so much for joining us today. As always it is a pleasure to talk to you about the work that you are doing.

I do want to highlight that Rick recently gave a talk on this work called *Locating the Architectural Roots of Technical Debt* at the 2015 Saturn Conference. That is available on our website at www.sei.cmu.edu/saturn/2015/video.

As always, we provide links to such resources in the transcripts. You can look there for those kinds of links and the podcast itself is available on the SEI website at sei.cmu.edu/podcasts and on Carnegie Mellon University's iTunes U site. As always, if you have any questions about this, please don't hesitate to contact us at info@sei.cmu.edu. Thank you for watching.