Resilient Security Architecture

A Complementary Approach to Reducing Vulnerabilities

Today, the IT world places little emphasis on “getting security quality right” from the beginning. The most common approaches to the latent (generally called 0-day) vulnerability problem fall into one of two categories:

- **Do nothing.** Wait for vulnerabilities to be discovered after release, and then patch them.
- **Test security in.** Implement code with vulnerabilities, and invest in finding or removing as many vulnerabilities as practical before release or production.

You won’t find advocacy for “do nothing” here because we must protect assets and reduce breaches. Regarding testing security in, I’m an advocate for security code review and scanning, testing, and solid security patching processes and policies, but are they enough?

The software industry would benefit from more emphasis on avoiding security mistakes in the first place. That means security requirements analysis and architecting and designing security in, an approach that’s currently rare but that provides substantial benefits. I wouldn’t expect to see much disagreement in principle from readers of this department. However, take a good look at your organization’s information assurance investment profile; compare how much you’re investing in getting it right to how much your organization spends fixing it, and you’ll see my point.

At Hewlett-Packard, we’ve developed HP Enterprise Services Comprehensive Applications Threat Analysis (CATA) Service (www.hp.com/go/cata), a methodology that takes an early-life-cycle and whole-life-cycle perspective on security quality improvement. Using it, we’ve avoided introducing thousands of vulnerabilities in hundreds of applications, dramatically reducing rework costs while increasing assurance. Our requirements- and architectural-analysis approach works effectively to identify and reduce security risks and exposures for new applications, for applications undergoing maintenance and modernization, and for assessing fully deployed stable systems. So, the methodology is effective regardless of the mix of new development and legacy systems.

The Deming Lesson

Those who cannot remember the past are condemned to repeat it. — George Santayana

Broadly speaking, the IT industry hasn’t remembered the quality improvement revolution or applied it to IT security quality. This isn’t surprising, because specialized disciplines tend to advance primarily on their own, and the cross-disciplinary application of lessons learned is less common. To make the connection clear, I start with an abbreviated history of quality and W. Edwards Deming’s role in igniting the quality improvement revolution.

In the 1950s, global manufacturing quality was poor. Repeatability was poor, and defects were rampant. Deming had been developing statistical process controls and quality improvement methodologies and had been presenting this work. His ideas first gained traction with the Japanese manufacturing industry, which is why Japanese cars have been known for so long for superior quality and reliability. Of course, high quality and repeatability have benefits beyond improved reputation and market differentiation; they can also dramatically reduce costs and increase productivity. However, Deming’s quality message didn’t gain traction in the US and the rest of the world for another 30 years.

What we’re seeing in IT security is much the same problem Deming saw in manufacturing quality—high incidents of defects, few quality controls, expensive rework, and so on. Consider a simple back-of-the-envelope calculation—the US...
National Vulnerability Database (http://nvd.nist.gov) lists more than 40,000 unique vulnerabilities. Independent analysis by HP\(^2\) and by IBM\(^3\) indicates that the total number of vulnerabilities is at least 20 times the number of reported vulnerabilities, leading to at least 800,000 unique vulnerabilities (including latent and otherwise unreported vulnerabilities). Consider that each application can have many (maybe dozens or even hundreds) of vulnerabilities, and you quickly arrive at millions to tens of millions of vulnerabilities across IT development.

The proximate cause (I say proximate because the economic root causes are beyond this article’s scope) for the large number of latent vulnerabilities is the lack of attention to the lessons from quality. Just as you can’t test quality in, you can’t test security quality in. You must architect and design it in first and then test to find and fix the smaller number of vulnerabilities introduced.

Cost-of-quality analysis from decades past established that defects cost orders of magnitude more to fix the later in the life cycle they’re discovered, fixed, or avoided. Typical study findings range from 30x to 100x increases, with some studies showing increases as high as 880x for postrelease defect repair versus repair in the earliest life-cycle stages. The most widely quoted figure is 100x, based on research by Barry Boehm.\(^4\) Studies specific to security vulnerabilities track well with the findings for quality in general (in other words, vulnerabilities can be considered security defects). See Figure 1.

This confirms that the return on investment (ROI) will be highest the earliest we deal with security defects. Pushing at least some security quality improvement investment to earlier in the life cycle will help improve security quality ROI and reduce cost.

**The Reactive Approach**

Historically, the IT industry has taken a reactive approach to security quality—it has worked backward, starting at the life cycle’s end and gradually working earlier (see Figure 2).

Security patching to fix vulnerabilities after a product’s release is critical, of course, but shouldn’t be the primary way to deal with vulnerabilities. However, it’s how the industry had first responded to them.

Some security quality investments then moved to the prerelease stage, but near the life cycle’s end. This work focuses on security testing of running code, in the form of vulnerability assessment and penetration testing (human or tool based), called *dynamic application security testing*. This technique was a significant improvement because it finds vulnerabilities before release. However, it’s still reactive, and rework is costly. This is because the code already contains vulnerabilities, and the goal at this stage is to find and fix as many of them as is practical.

Next in the progression, and a step earlier in the life cycle, is finding vulnerabilities in source code through *static application security testing*, either through automated scanners or human-expert security code review. This technique
improves the ROI; however, it's still reactive in that it removes vulnerabilities instead of preventing them.

**Checklists Don’t Work (in Isolation)**

Security checklists are an easy way for organizations to improve security quality. Unfortunately, unless employed carefully in the context of a broader security quality program (or as a specific finding in an assessment), they'll more likely produce a false sense of security rather than real improvements. Checklists only address somebody’s list of egregious security issues. If all you do is address such a checklist, consider how much larger the set of (serious) unaddressed security issues is! Fully addressing the checklist doesn’t tell you much about how secure the resulting application is! Fully addressing the checklist doesn’t tell you much about how secure the resulting application is because it tells you nothing about what remains exposed.

**The Proactive Approach**

To achieve the maximal ROI, you’ll need to use these two methodologies in the life cycle’s earliest phases:

- **Security requirements gap analysis.** Rigorously examine the security requirements relevant to your project and your commitment level to meet those requirements, addressing gaps and disconnects.
- **Architectural threat analysis.** Examine the planned or implemented architecture for attack surfaces, consistent application of sound secure-design principles, security vulnerability robustness, and resiliency. The goal is to dramatically reduce the probability and severity of latent vulnerabilities, both known and unknown.

This approach reverses IT’s tendency to address security reactively. Instead, it starts at the life cycle’s beginning, reducing the need for rework, and emphasizes quality throughout.

We found this early-life-cycle approach necessary because when we looked under a (security) rock, we almost always found something. So, we realized that a more proactive approach was the only way to get ahead of the problem. We first started examining layered (user space) operating system software. But the more we looked at the different functional areas of software, firmware, and hardware across different industry verticals, the more we saw the universality of the problem and our solution. Systematic, scalable, repeatable solutions are required; reactive elements, although necessary, can’t solve the problem by themselves.

You might argue that your organization already considers some security requirements and some security design principles, so is this really new? It’s great if you already do, but this approach is more than basic consideration. It’s a systematic examination of security requirements and security design principles—a quality and completeness check. And even when teams pay attention to these issues up front, we still consistently discover some major gaps and security issues.

You might also argue that this approach makes sense only in a waterfall life cycle and that you use agile (or iterative) development. It is simplest to discuss this approach using waterfall terminology, but there’s nothing inherently waterfall about it. You can just as easily consider the various techniques I’ve outlined as tackling security from different abstraction layers (requirements, architecture, design, source code, or dynamic behavior), regardless of the order of implementation. Small changes in architecture can dramatically reduce the probability of vulnerabilities in large quantities of code—for instance, by using a checkpoint security design pattern or reducing excessive elevation of privilege. If you remain exclusively at the code or runtime-behavior layers, you must deal individually with each vulnerability, rather than potentially eliminating tens or hundreds of vulnerabilities at a time.

Adopting this proactive approach creates some challenges because security expertise is far from pervasive, and that’s unlikely to change any time soon. So, we needed an approach that didn’t require pervasive security expertise. Also, because programmers are human, we can’t keep them from ever making mistakes. So, because we couldn’t rely on defect-free software, we realized our approach needed to significantly reduce the probability that ordinary defects would become vulnerabilities.

Over several years, we’ve developed and optimized our methodology and early-life-cycle security quality processes. We scale by requiring security expertise in a small cadre of certified reviewers who can review many projects. The larger development teams don’t require security expertise. (Of course, such embedded expertise obviously helps, and we encourage expanding that expertise through security training and participation in security reviews.) We apprentice and certify our reviewers in CATA.

**Security Requirements Gap Analysis**

Although I’ve made a big point about the early-life-cycle approach, we added security requirements gap analysis only after we’d been using architectural threat analysis for a couple of years. This happened because, when we asked development teams during architectural threat analysis what their security requirements were, they frequently had insufficient information to answer and looked to us for guidance.
There were several reasons for this; the most significant was that the end users weren’t the security stakeholders. Development teams often have good processes to communicate with potential customers or end users, but not with security stakeholders. Typically, the security stakeholders are IT information security departments, business information security managers, CIOs, chief information security officers, and so on.

Also, the security requirements’ sources might be laws, regulations, and practices, which are far outside most developers’ field of vision or experience. When development teams gather their requirements from application users, they gather an important set of requirements but not a complete set of nonfunctional requirements, such as security.

We’ve developed methods, intellectual property, tools, databases, and years of expertise to help us translate security requirements from the stakeholders’ language to the developers’ language. Without such translation, development teams often fail to implement the underlying security mechanisms needed to enable cost-effective application deployment that meets regulatory-compliance requirements. This failure results in increased use of compensating controls, more audit findings, and acceptance of higher risk in deployment environments.

Applying our methodology, we’ve been able to consistently identify otherwise-missed security requirements that can, if addressed early on, significantly reduce deployed applications’ total cost of ownership. A typical assessment using our methodology finds 8 to 10 issues during requirements analysis (and a similar number during architectural threat analysis).

Some issues can translate into a high probability of dozens or hundreds of vulnerabilities if not addressed early. For example, one case involved two nearly identical applications (the same functionality independently developed for two different OS platforms, eventually resulting in a project to merge the two development efforts). One application had applied CATA (because the development occurred in an organization that had adopted the methodology); the other hadn’t (that organization planned to adopt the methodology). At the last count, the first application avoided more than 70 vulnerabilities; the other had to issue several security bulletins to patch the more than 70 vulnerabilities.

Late-life-cycle fixing can be 100 times more expensive, and breach disclosure costs, security-related regulatory-compliance penalties, and downtime costs can amount to millions of dollars. So, it’s easy to see that small-to-moderate expenditures up front can easily pay for themselves many times over.

Architectural Threat Analysis

We analyze security and control requirements and architecture to evaluate how robust or resilient the application architecture and high-level design are with respect to security vulnerabilities. This is based partly on known approaches to threat analysis, such as attack surface analysis, and quantitative and qualitative risk analysis. Over years of use and improvement, our methodology has evolved into something unique, as we found that no prior methodology scaled adequately or generated consistent high-reliability results.

For instance, structured brainstorm-based approaches (most industry approaches rely somewhat on structured or unstructured threat brainstorming) depend heavily on the participants’ creativity, security expertise, and stamina. Variability in these factors produces dramatically different results. Bruce Schneier said

Threat modeling is, for the most part, ad hoc. You think about the threats until you can’t think of any more, then you stop. And then you’re annoyed and surprised when some attacker thinks of an attack you didn’t.

With our consistent, repeatable, and scalable methodology, we typically can find several fundamental architectural security risk issues that, when addressed, can avoid many vulnerabilities (in some cases, hundreds with a single finding). Our methodology also achieves completeness that a brainstorm-based approach can’t—we know when we’ve completed the analysis, and not simply because we “can’t think of any more.”

Comparisons

Which is more beneficial—security requirements gap analysis or architectural threat analysis? Both provide substantial but different benefits. Getting requirements wrong is a huge issue, because you can do the greatest job of building the wrong application, and it won’t achieve its purpose. However, if the application isn’t architect-ed to be robust and resilient from a security perspective, it’s doomed to be riddled with vulnerabilities and thus likely won’t meet its security requirements.

How does our methodology compare to Microsoft Security Development Lifecycle (SDL; www.microsoft.com/security/sdl/default.aspx)? We start with substantial analysis to identify missing security requirements, whereas SDL’s requirements analysis is more limited to security-process requirements. We both have a threat-modeling component, but Microsoft’s falls more in the structured-brainstorming model.
The Optimized Approach

No single approach is a panacea, so we combine early- and late-life-cycle approaches. Optimized security requires a full-life-cycle perspective, with increased emphasis in the earliest phases (see Figure 3). Security requirements gap analysis ensures we’re building the right product from a security perspective. Architectural threat analysis ensures we’re building the product right—dramatically reducing the number of vulnerabilities in the application. Dynamic and static application security testing identify most of the remaining vulnerabilities, reducing the need for security patching.

Most of you would probably agree that the IT industry has a security quality (latent or 0-day vulnerability) problem and that security quality investments must consider the whole life cycle. What might be a new idea for some is that security quality investment profiles must shift to earlier during development.

One sign of the increased recognition of the need to solve this problem is the 2011 US Department of Defense authorization bill, which requires

- “assuring the security of software and software applications during software development” and
- “detecting vulnerabilities during testing of software.”

Another is the relatively recent creation of the Certified Secure Software Lifecycle Professional credential. It should be self-evident that security quality improvement can’t rely on programmers never making mistakes. However, the implication that far greater robustness and resiliency must therefore be designed into applications might not be so evident.

References

John Diamant is a Hewlett-Packard Distinguished Technologist and HP’s Secure Product Development Strategist. He founded and leads HP’s security quality program. Contact him at john.diamant@hp.com.

Interested in writing for this department? Please contact editors Brian Chess (chess@hp.com) and Brad Arkin (barkin@adobe.com).