Developing secure software is easy! Just don’t have any bugs. Oh, and use security APIs correctly, and make sure your software doesn’t have any undocumented functionality or side effects, doesn’t have race conditions, and doesn’t use unsafe environment variables or any other input, for that matter. It also needs to mediate completely any access to any protected resource. The mediation must be tamper-proof, nonbypassable, and small enough to verify its correctness. Your software’s security should support the users’ needs. It should also be easy for users to figure out, set up, and inspect, even if they do it only once a year and don’t bother to read documentation or stay up-to-date on the latest attacks and defense techniques. All this stuff must not affect your budget, development schedule, or, of course, the functionality of the software you’re building (what the customers are really paying for!). Easy, huh?

Joking aside, while security was once a specialty of interest to only a few programmers, it’s now a critical topic for almost all software engineers, project managers, and decision makers. This is not only because pervasive software insecurity has been in the news for the last several years but also because all software developers now feel the pressure (or at least expectations) from end users to deliver secure software.

Yet no company can afford to throw away all its existing software and redevelop it from scratch or retrain all software developers to be security experts. This is why we need methods, processes, and tools for improving the security of software created by a wide range of developers with a wide range of resource constraints, development methods, time-to-market schedules, contexts, and organizational and national cultures.

Historically, many publications specializing in security have focused on the development of software for security-critical domains (for example, military, safety critical, and government), where the production cost and time-to-market are, although nonnegligible, of secondary concern. Unlike those publications, this special issue focuses on creating and maintaining secure software by the wide range of developers who constitute the software industry—many who work in domains where cost (both production and maintenance) and time-to-market are the main driving factors.

**What’s special about software security?**

When most people think about software security, they think about authentication, encryption,
and access control. All these security features are important, but security features alone don’t make a system secure. A system’s design and the techniques used to construct it have as much, if not more, impact on the system’s eventual security. Why would an attacker break encryption if he or she can just exploit a buffer overflow? Why crack passwords if the attacker can bypass the authentication system entirely by typing a special URL into his or her browser?

When the Internet was younger, organizations tried to cordon off vulnerable software by putting up firewalls. A firewall would protect a vulnerable internal network from the ravages of the unruly Internet. The problem is that customers are increasingly demanding software that communicates and interoperates with other systems. Such programs must be designed and built so that they can handle attacks and intentional misuse.

Software vendors can’t solve the digital security problem alone. They have to rely on secure networks and system administrators who think about security when deploying new services. Developers also have to rely on users to make reasonable choices about security. But developers can’t just throw up their hands and treat security as someone else’s job. Also, techniques for building high-reliability systems won’t necessarily help developers build adequately secure systems. For example, redundancy is good for combating independent random events, but an attacker who can force a system to fail can often just as easily cause a redundant system to fail.

In this issue

Getting security right is hard because an attacker—having virtually unlimited time—needs to find only one vulnerability in a system to succeed, whereas the defender—constrained in time—must ensure that the system has no weak points. Software systems are specifically difficult because of their inherent high complexity (Windows Vista, for example, is rumored to comprise 60 million lines of source code), connectivity, and extensibility. On top of that, software systems aren’t linear or continuous, which makes their analysis and testing (either functional or security specific) even harder.

Security bugs themselves are harder to catch. They show up not only as a lack of required functionality (for example, missing authentication) but also as incorrect implementations (for example, easy-to-spoof authentication) or, even worse, as unintended, undocumented, or unknown ‘‘features’’ (for example, buffer overflow in authentication). Charlie Lai discusses several such unintended side effects of Java constructs in this special issue. His article, “Java Insecurity: Accounting for Subtleties That Can Compromise Code,” examines three fundamental (and language-agnostic) coding guidelines that Java developers commonly follow to ensure their programs are safe: minimizing accessibility, creating copies of mutable inputs, and preventing the unauthorized construction of sensitive classes. But rather than focusing on the guidelines themselves, he explores the subtleties involved in applying them in Java and suggests approaches Java developers can use to account for them in their work.

But before developers can even have a chance to deal with design or implementation vulnerabilities, they need to properly collect and analyze security requirements. The right set of security requirements will lead to security features and an implementation focus that’s appropriate for the job at hand, whereas the wrong requirements can lead to a never-ending cycle of security failures and associated finger-pointing. This nontrivial exercise gets even harder if developers who aren’t security experts will manage the process. “Security Requirements for the Rest of Us: A Survey,” by Inger Anne Tondel, Martin Gilje Jaatun, and Per Håkon Meland, examines how people elicit and record security requirements for software projects. The authors present their own requirements methodology and argue for bringing security awareness to all participants in a software development project.

About the Authors

Konstantin (Kosta) Beznosov is an assistant professor in the University of British Columbia’s Department of Electrical and Computer Engineering. He founded and leads the university’s Laboratory for Education and Research in Secure Systems Engineering. He previously was a security architect with Hitachi Computer Products America, where he designed and developed products for security integration of enterprise applications. He has also been a consultant for large telecommunication and banking companies on the architecture of security solutions for distributed enterprise applications. He’s a coauthor of Enterprise Security with EJB and CX com (John Wiley & Sons, 2001) and Mastering Web Services Security (John Wiley & Sons, 2003). He received his PhD in computer science from Florida International University. Contact him at the Dept. of Electrical and Computer Eng., Univ. of British Columbia, 4047-2332 Main Mall, Vancouver, BC V6T 1Z4, Canada; beznosov@ece.ubc.ca.

Brian Chess is a founder and chief scientist at Fortify Software, where he focuses on practical methods for creating secure systems. Secure Programming with Static Analysis (Addison-Wesley, 2007), which he cowrote with Jacob West, shows how static source code analysis is indispensable for getting security right. He received his PhD in computer engineering from the University of California at Santa Cruz. Contact him at 2215 Bridgepointe Pkwy., Ste. 400, San Mateo, CA 94404; brian@fortify.com.
Another important security-specific activity that must be properly done during requirements and design phases is threat analysis—the analysis of what can go wrong. Even in theory, building a 100-percent secure yet usable system is impossible, without regard to budget and time-to-market constraints. So, developers must identify and prioritize threats to the system. Only then can they design countermeasures against the most critical threats and build them into the system. Jeffrey Ingalsbe, Louis Kunimatsu, Tim Baeten, and Nancy Mead reflect on their team’s threat-modeling experiences at Ford Motor Company. In “Threat Modeling: Diving into the Deep End,” they discuss how they applied the Microsoft Threat Analysis and Modeling process to a dozen projects, what benefits they realized, and what lessons they learned.

Despite all the advances in security technologies, development of secure software remains costly. For example, Microsoft has spent over US$200 million since 2003 on its Security Development Lifecycle, according to Steve Lipner, Microsoft’s Senior Director of Security Engineering Strategy in Trustworthy Computing. The amount of investment in security is a risk management decision: what’s the likelihood of an attack, and what are an attack’s likely consequences? Estimating the answers to those questions is tricky. Shari Lawrence Pfleeger and Rachel Rue begin their article, “Cybersecurity Economic Issues: Clearing the Path to Good Practice,” with a simple observation: time, money, and good people are always in short supply. So how much time, money, or effort should you devote to security? The authors explore how organizations search for answers and what obstacles they often encounter. They discuss how to gather the right data and make good trade-offs between cost and level of protection. They present a framework for comparing economic security models so that stakeholders can make the right investments in security.

This special issue reports on the state of the practice and recent advances in engineering secure software for the wide range of industrial application domains. The articles explore practical and pragmatic ways of meeting this challenge; we hope you find them helpful in your work.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.