# Software Survivability: Where Safety and Security Converge

Karen Mercedes Goertzel
*Booz Allen Hamilton*

*As safety-critical software moves from closed environments to open and commodity technologies, security threats will inevitably increase. Organizations dependent on mission-critical systems and networks are recognizing that the traditional "protect-detect-react" (PDR) strategy for countering intrusions and attacks is ineffective. A new information assurance and cybersecurity strategy is needed that augments PDR with the ability of systems and networks to "fight through" attacks. This article examines techniques that both security- and safety-critical software developers can leverage to increase their software's survivability.*

Software security and software safety share the need to assure that software will remain dependable under *extraordinary conditions*. Extraordinary conditions—those which software was not intended to gracefully tolerate—will either cause it to behave unpredictably or fail outright. What distinguishes software *safety* from software *security* is what constitutes an extraordinary condition for that software, and what is at stake if it fails as a result.

Extraordinary conditions that threaten software safety are termed *hazards*, reflecting the perception that such conditions are accidental. By contrast, extraordinary conditions that threaten software security are termed *attacks* or *exploits*, indicating their intentionality. The objective of most attacks on software is to sabotage or subvert the software's operation by exploiting one or more weaknesses in the software's execution environment (e.g., failure of the application firewall that blocks malicious input from entering the system), design (e.g., accepting input from unrecognized entities), implementation (e.g., accepting input in a fixed-length buffer without first validating that input's length), operation (e.g., a failure of user interface software, thereby exposing the system's command line), or development process (e.g., poor configuration control, peer review, and testing practices that allow a disgruntled programmer to surreptitiously embed malicious logic).

## Intentional Threats to Safety-Critical Systems

Software failures that result from safety hazards can have dire, even fatal, consequences due to the extremely strong linkage between the software and the physical system that it is supposed to control. Whether the software constitutes the single small, closely contained embedded program that controls an automobile's anti-lock braking system, or several dozen modules dispersed throughout a distributed supervisory control and data acquisition (SCADA) system controlling an entire region's wastewater treatment, the functions performed by the physical components are what determine whether the system (including its software) is safety-critical. If a system failure results in damage to the physical environment in which people live, physical maiming, damage to health, or death of one or more humans, the system is safety-critical. The failure of software in such a system can have catastrophic results.

Safety hazards tend to be straightforward and accidental. By contrast, security threats are intentional: the result of human creativity and perspicacity absent from safety hazards (although a hazard may introduce a vulnerability that an attacker can intentionally exploit). Because they are guided by human intelligence, security threats are usually less predictable, more complex, more numerous, and more persistent than safety hazards. The same system may be repeatedly targeted by a variety of simultaneous and sequential attacks, some aimed at the interface level, others at the application components, and still others at the execution environment level—all orchestrated to accumulate and intensify until they collectively produce the critical failure(s), enabling the attacker to achieve his objective.

Google "Ariane 5 Flight 501," "Therac-25 accidents," or "Toyota Prius software bug" to read about some dramatic instances of safety-critical systems that failed as a result of design flaws or implementation errors in their software. These were unintentional flaws and errors, caused by developer inadvertence, negligence, or misapprehension, but their impact was dramatic. How much more disastrous might they have been had their cause been intentional exploitation or implanted malicious logic?

Now Google "trans-Siberian gas pipeline" + "software bug." What you'll get are reports of the 1982 technology coup. The CIA, having learned that Soviet spies planned to secretly acquire a gas pipeline controller developed in Canada, planted a Trojan horse (logic bomb) in the controller's software. Once installed on the trans-Siberian pipeline, the controller ran a test of the pipeline's pressure gauges during which the logic bomb reset those gauges to double gas pressure in the pipeline. The resulting explosion was, up to that time, the largest non-nuclear explosion ever photographed from space [1].

In the 25-plus years since that incident, attacks on safety-critical systems involving the embedding of malicious code or direct penetrations have proliferated, several of which have been perpetrated by the systems' own disgruntled developers or administrators. Such attacks are proliferating due in part to opportunity: More safety-critical systems are built from or hosted on commodity software, the vulnerabilities of which are widely publicized and well understood by attackers, then exposed on semi- or fully open networks (including the Internet). The increasing software intensiveness of safety-critical systems means more of their critical functions are performed by software than by hardware, and that software is necessarily larger and more complex, making its vulnerabilities harder to predict and detect.

As with safety hazards, the impact of software failures resulting from attacks and exploits depends on the nature of the targeted system. A threat to a safety-critical system can have the same dire consequences as a hazard. Even in non-safety-critical systems, the consequences of failure can be catastrophic: Insider sabotage of an intelligence database application may enable an attacker to steal the names of undercover operatives in an adversarial country and sell it to that country's

counterintelligence service, which then has them captured and executed. The subversion of software in a military logistics system that calculates the number of biochemical suits may result in a shortage of protection for forward-deployed forces during a chemical weapons attack.

## Embedded, Not Isolated

Many safety-critical systems are embedded. Until recently, that meant they were small, relatively simple, and isolated from direct interaction with humans (they even lacked means for such interaction). Today's embedded systems are different. They both benefit and become vulnerable from the increased power of the processors on which they are hosted. These are processors that enable the use of commodity operating systems, such as Microsoft CE, which share security problems with non-embedded operating systems sharing the same kernel code[1].

The less proprietary and more connected embedded systems become, the less specialized expertise attackers need to target them. Systems from temperature controls to medical devices to on-board automobile computers and sensors are now accessible via wireless Radio Frequency Identification (RFID), cellular, and satellite links that use standard communications protocols. Implanted medical devices are increasingly accessible via RFID [2]. A DoD telemedicine application enables surgeons in U.S. military hospitals to issue commands, via a satellite uplink, to a software-controlled robot in Iraq, thereby performing laser surgery on wounded soldiers in theater [3, 4].

But where there is a wireless network, one can almost guarantee there will be an attacker attempting to locate, intercept, and tamper with the signals transmitted between the systems at either end of the wireless link. Consider telematic systems such as GM's OnStar, Ford's remote emergency satellite cellular unit and vehicle emergency messaging system, Volvo's On Call, BMW's Assist, and Mercedes-Benz's Tele Aid and COMAND. They all use cellular or satellite connections to allow their call center representatives to perform remote diagnostics on the onboard computers of subscribers' vehicles. Privacy concerns about certain data collected by these telematic services are well documented, but a recent addition to OnStar is even more worrying. Owners of 1.7 million OnStar-equipped 2009 GM vehicles can allow their engines to be "remotely switched off through the OnStar mobile communications system" [5] at the behest of the police. The goal is to stop stolen GM vehicles in their tracks during high-speed police car chases, thereby reducing the number of fatal accidents associated with such chases. The implications of OnStar's transition from a passive monitoring and diagnostics system to an active controller of a safety-critical embedded system (the engine) have been noted:

> [Some] automotive communication networks have access to crucial components of the vehicle, like brakes, airbags, and the engine control. Cars that are equipped with driving aid systems allow deep interventions in the driving behavior of the vehicle .... Malicious attackers are not to be underestimated. [6]

The next logical step—remote updates via telematic links to embedded software and firmware—would create an ideal conduit for insertion of malicious logic into embedded computers or causing denial-of-service by injecting "garbage bits" into telematic data streams [7].

## Security of Safety-Critical Infrastructure

Along with embedded systems, another type of safety-critical system never originally intended to support publicly discoverable/accessible wireless network connections is the industrial control system. Both the SCADA and distributed control systems (DCSs), along with air traffic control systems, are safety-critical hybrids of information systems, command and control systems, and physical process control systems. They support the same open networking protocols, remote accessibility, and even Internet connectivity typically found in information and command and control systems. Like those systems, safety-critical control systems are being built from commodity and open components and hosted on mobile devices running commodity and open operating systems.

A sobering example of where such advances can lead occurred in the Maroochy wastewater treatment facility in Queensland, Australia [8, 9]. The DCS that controlled the facility included remote administration software that ran on Microsoft Windows and provided remote wireless network access to the facility's physical control functions (including opening and shutting valves). Vitek Boden, a former contractor who helped install the system, later submitted a job application that was rejected. The vengeful engineer applied his expert knowledge: Over the next four months, on more than 40 separate occasions, he parked his car near the water treatment plant and, with a laptop that had a wireless radio transmitter, used a stolen copy of the DCS's remote administration software to identify himself to the DCS as "Pumping Station 4," then issued commands that suppressed the DCS's alarms and changed its settings to place excessive back-pressure on the valves.

By the time the plant's operators finally figured out that the series of inexplicable failures in the plant were caused by sabotage of its DCS and notified police, Boden was in the midst of his 46th incursion into the system. In the end, he managed to release between 264,000 and 1.18 million gallons of raw sewage (including human waste): The Maroochy River tributaries turned black, marine life was poisoned, and the air reeked.

Not only does the Maroochy incident vividly illustrate the danger of the insider threat, it shows how vulnerable remote-controlled safety-critical systems can be[2]. As the *Washington Post* observed:

> … like thousands of utilities around the world, Maroochy Shire allowed technicians operating remotely to manipulate its digital controls. Boden learned how to use those controls as an insider, but the software he used conforms to international standards, and the manuals are available on the Web. Nearly identical systems run oil and gas utilities and many manufacturing plants. [7]

## Secure Development of Safety-Critical Software

Software engineering for safety-critical systems is impressively scientific and disciplined. It is driven by heightened quality and fault-tolerance imperatives and has careful, thorough hazard analyses, fault-tolerant designs, and rigorous testing. As well, *safe* subsets of programming languages are used and formal specification, modeling, and verification is utilized. As a result, most safety-critical systems can tolerate and continue operating dependably in the presence of the unintentional faults and failures associated with safety hazards.

But safety-critical software must be equally intolerant of failures caused by intentional threats and keep operating dependably even under attack. This means eliminating weaknesses, bugs, flaws, errors, etc., that don't necessarily lead to failures, but which can be exploited by attackers.

Security for safety-critical systems—and indeed for all software-based systems—must be achieved at the functional, data, and environmental levels. At the functional level, the software must be able to withstand threats to its own integrity and availability; these include threats of denial-of-service, intentional corruption or tampering with the software's executables and/or control files, and embedding/insertion of malicious logic. At the data level, inputs received and outputs produced by the software may be tampered with or intentionally corrupted. If the system stores, manipulates, or transmits information, that information is also subject to the same threats, plus the threat of inappropriate disclosure. The software's execution environment is subject to threats to its availability and integrity, along with a further threat of hijacking or *theft* of computing resources (memory, disk space, computing power) by illicit processes that make those resources unavailable to valid processes.

Software security focuses on specifying software's internal workings to remain dependable in the presence of potentially hostile external interactions. Moreover, the software must not contain design weaknesses or implementation errors that, if intentionally or accidentally escalated, could lead to a failure (i.e., any incorrect or unpredictable behavior) that could leave the software exposed and vulnerable to direct attack. Such failures may result from a hostile input to the software itself, or from a fault triggered by an attack on the software's environment.

## Secure = Survivable

To date, the established paradigm for system security has combined proactive PDR strategies (which includes recovery). Protection is often achieved through defense-in-depth layering of security mechanisms, controls, and procedures at the functional, data, and environmental levels of the system. Detection of threats (or more accurately, their manifestation as intrusions and attacks) is achieved by a combination of intrusion detection, event logging, and usage auditing and monitoring. Reaction to intrusions and attacks focuses on minimizing the extent, intensity, and duration of the incidents' impact and the likelihood of their recurrence. Reaction often comes at the expense of dependability because it requires rejecting certain types of inputs (some of which may in fact be valid), terminating some user sessions, shutting down some or all functions, or disconnecting the system from the network (to disengage it from the suspected attack source).

In the DoD, practitioners of information assurance, computer network defense, and cybersecurity have begun to admit that this PDR paradigm is essentially flawed. Attackers have become too skilled, too expert, too flexible, and too ingenious for countermeasures that rely on the ability to recognize the threat to keep up. Information and cyber warfare fought on current terms is not just being lost, it is unwinnable.

The DoD and numerous other organizations now recognize the need for a *paradigm shift* to enable their systems to survive high-intensity intrusions and attacks. Survivability (also referred to as resilience), which has always been required for safety-critical systems, must become the norm for mission- and security-critical software as well.

Designing for survivability means including redundancy and rapid recovery features at the system level (e.g., automated backups and hot-sparing with automatic swap-over of high-consequence components and modularized designs that enable those components to be decoupled and replicated on *hot spare* platforms). It means implementing significantly more error and exception-handling functionality than program functionality: error and exception handling that is purpose-built, not generic, to minimize the possibility of faults escalating into failures. If possible, rather than failing, the software should be able to keep running at a degraded level of operation (i.e., reduced performance, termination of lower-priority functions, rejection of new inputs/connections). If it must fail, its exception handler should prevent the failure from placing the software into an insecure state, dumping core memory, or exposing the content of its caches, temporary files, and other transient data stores. For safety-critical software—in which there is no threshold of tolerance for the delays typically involved in post-failure recovery and restoration—survivability measures must prevent failures, full stop. This is true whether the failure was accidental or intentionally induced.

### *Engineering for Survivability*

Survivability has become the subject of research, as demonstrated by the Survivable Systems Engineering program at Carnegie Mellon University's Computer Emergency Response Team Coordination Center (see <www.cert.org/sse>), the Willow Survivability Architecture developed by University of Virginia's Dependability Research Group (see <http://dependability.cs.virg

## Coming Events

**November 1-5**
*SIGAda 2009*
Tampa Bay, FL
www.sigada.org/conf/sigada2009

**November 2-4**
*The 13th IASTED International Conference on Software Engineering and Applications*
Cambridge, MA
www.iasted.org/conferences/home-669.html

**November 2-6**
*Software Assurance Forum*
Washington, D.C.
https://buildsecurityin.us-cert.gov/daisy/bsi/events/1102-BSI.html

**November 9-11**
*International Conference on Software Quality*
Chicago, IL
www.espresso-labs.com/icsq2009

**November 9-13**
*16th ACM Conference on Computer and Communications Security*
Chicago, IL
www.sigsac.org/ccs/CCS2009

**November 9-13**
*Agile Development Practices Conference 2009*
Orlando, FL
www.sqe.com/agiledevpractices

**December 13-16**
*Winter Simulation Conference 2009*
Austin, TX
http://wintersim.org

**April 26-29, 2010**
*22nd Annual Systems and Software Technology Conference*


www.sstc-online.org

***Coming Events:*** **Please submit coming events that are of interest to our readers at least 90 days before registration. E-mail announcements to: <marek.steed.ctr@hill.af.mil>.**

inia.edu/research/willow>), and Mithril, developed by the National Center for Supercomputing Applications (see <http://security.ncsa.uiuc.edu/research/mithril/>). These efforts combine techniques for engineering high-confidence and safety-critical software with software security assurance principles and practices, and generate methodologies and tools for producing software that can remain survivable in the face of intentional threats as well as accidental hazards.

Emerging survivability techniques, as well as more established high-confidence and safety engineering techniques, methods, and tools can benefit developers of software-based systems with strong security imperatives, including systems that are larger, more complex, more interactive, and more extensively networked than most safety-critical systems, high-confidence embedded systems, cryptosystems, and so forth.

## Software Practices that Aid Security and Safety

Just as developers of security-critical software can benefit from safety engineering practices, safety-critical software development needs to undergo its own *paradigm shift* to account for *intentional hazards*. Researchers in both the safety and security communities are adopting and adapting software assurance principles, practices, and tools from the other community to aid them in producing software that is safe and secure. Among these efforts, three significant trends stand out:

### *Simplification of Formal Methods*

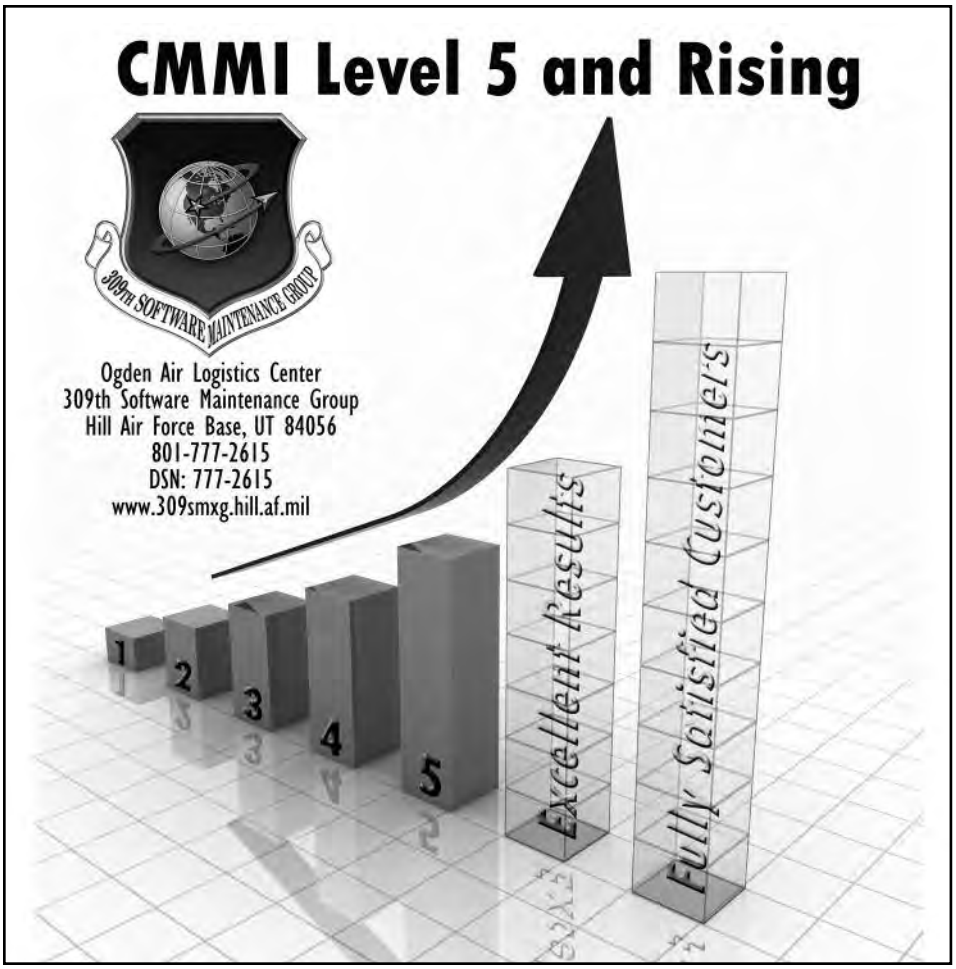
Tool-supported modeling and proofs of security properties in large, complex software systems is made possible by *semi-formal* methods, such as: 1) Praxis High Integrity Systems' Correctness-by-Construction, which is a structured development methodology into which formalisms have been selectively incorporated; and 2) tools that automate formal activities so they can be performed by non-experts. Examples include Correctness-by-Construction's supporting tools, Munich University of Technology's Autofocus and Quest, Jean-Raymond Abrial's B-Method, and (to some extent) the Object Management Group's Model-Driven Architecture.

### *Hybrid Assurance Cases*

Hybrid assurance case standards, templates, and processes (including both safety and security arguments and evidence) are emerging. Examples include the SafSec standard developed by Praxis High Integrity Systems for the United Kingdom's Ministry of Defense and ISO/IEC 15026, System and Software Engineering–System and Software Assurance. Also noteworthy are the safety and security extensions defined for the integrated CMM® and CMMI® by the Federal Aviation Administration and the DoD [10]. Their objective: extend processes defined by and validated under those CMMs to include safety and security engineering practices.

### *Biological Models and Computer Immunology*

Biological models and computer immunology are being applied to software resilience/survivability to achieve diversity and evolution/adaptation through: 1) creation of different instantiations of software programs whereby the computational results are identical but the architectures, source code, and/or binary images diverge and thus are not all equally susceptible to the same threats and 2) use of pseudo-genetic algorithms to gradually evolve executables over time within the acceptable bounds of the software's functional specifications, thus enabling them to continue operating correctly despite the transformation. Specific techniques include: dynamic software composition, N-version programming, and code filtering. Other biological metaphors have resulted in *software rejuvenation*, phylogenetic trees for predicting vulnerabilities, and techniques for nature-based modeling of software systems.




## Conclusion

Survivability as an adjunct to the PDR model of information assurance and cybersecurity is expected to be embraced more fully by DoD and by other communities that operate mission-critical, safety-critical, and life-critical systems. To the extent that software safety engineering minimizes or eliminates implementation errors and environment faults, it contributes to the security of that software. It cannot, however, achieve security on its own because it does not consider design weaknesses that can be exploited as vulnerabilities or exploitable errors and faults that are not expected to result in failures. Adding software security principles and practices to software safety engineering can bridge the gap between producing software that remains dependable in the presence of unintentional hazards and software that remains dependable in the

® CMM and CMMI are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

presence of both hazards and intentional threats.◆

## Resources

Information on secure software development practices, methodologies, and tools is proliferating in print and on the Internet. Three resources of particular value (both for their own content and for the extensive lists of references they contain) are:

1. The DHS' "Build Security In" Web site at <https://buildsecurityin.us-cert.gov/>.
2. The Information Assurance Technology Analysis Center and the Data and Analysis Center for Software's (DACS) "Software Security Assurance: A State-of-the-Art Report," is available online at <http://iac.dtic.mil/iatac/download/security. pdf>.
3. The DHS (sponsor) and DACS (publisher) document, "Enhancing the Development Life Cycle to Produce Secure Software," is available online at <https://www.thedacs.com/techs/enhanced_life_cycles>.

## Software Defense Application

DoD developers of weapons systems, avionic systems, surgical robots, and other safety-critical systems should find this article helpful in clarifying the security threats such systems face as they are increasingly networked and, thus, exposed not only to safety hazards but to intentional attacks and exploits by nation states and cyberterrorists. DoD developers of classified information systems, security controls, cryptosystems, and other security-critical software-based systems should benefit from the discussion of safety engineering techniques that can increase the survivability of those systems. Finally, survivability (as a concept) provides a shared point reference from which developers of safety-critical and security-critical defense systems can establish an ongoing dialogue to share the contributions each of their communities can make, in terms of engineering methods, techniques, and tools, to advancing the state-of-the-art of software survivability engineering.

## References


1. Quinn-Judge, Paul. "Cracks in the System." *Time*. 9 Jan. 2002 <www.time.com/time/magazine/article/0,9171,260664,00.html>.
2. Becker, T.J. "Improving Medical Devices: Georgia Tech Research Center Expands Testing Capabilities to Help Reduce Potential Interference." *Georgia Tech Research News*. 25 July 2006 <www.gtresearchnews.gatech.edu/newsrelease/eas-center.htm>.
3. Ackerman, Robert K. "Telemedicine Reaches Far and Wide." *SIGNAL*. Mar. 2005 <www.afcea.org/signal/articles/templates/SIGNAL_Article_Template.asp?articleid=693&zoneid=128>.
4. Murakami, H., et. al. Tohoku University Sendai. "Telemedicine Using Mobile Satellite Communication." *IEEE Transaction on Biomedical Engineering* 41:5 (1994): 488-497.
5. Woodyard, Chris. "Device Can Remotely Halt Auto Chases." *USA Today*. 9 Oct. 2007 <www.usatoday.com/money/autos/2007-10-09-onstar-stop-pursuits_N.htm>.
6. Farwell, Jennifer. "Hijacked, Corrupted and Crashed: Does the New Generation of Computerized Cars Pose a Security Threat?" *PC Today* 3.10. Oct. 2005 <www.pctoday.com/editorial/article.asp?article=articles%2F2005%2Ft0310%2F05t10%2F05t10.asp>.
7. Gellman, Barton. "Cyber-Attacks By Al Qaeda Feared." *The Washington Post*. 27 June 2002 <www.washingtonpost.com/wp-dyn/content/article/2006/06/12/AR2006061200711. html>.
8. Slay, Jill, and Michael Miller. "Lessons Learned from the Maroochy Water Breach." *Critical Infrastructure Protection* 253 (2007): 73-82.
9. Graham-Rowe, Duncan. "Power Play." *The New Scientist* 2447: 15 May 2004.
10. Goertzel, Karen Mercedes. "Computer Immunology." *DoD IA Newsletter* 7:(2). Fall 2004 <http://iac.dtic.mil/iatac/download/Vol7_No2.pdf>.


## Notes

1. Granted, additional processor power also makes computing resources available for security countermeasures, such as input validation and sophisticated exception handling.
2. In the 1990s, the Nuclear Regulatory Commission prohibited remote control of industrial control systems at nuclear power plants. See Scott Berinato's article for CIO entitled: "Cybersecurity – The Truth About Cyberterrorism" <www.cio.com/article/30933/CYBERSECURITY_The_Truth_About_Cyberterrorism>. Serious efforts to improve SCADA and DCS security increased after Sept. 11, notably in the Departments of Homeland Security and Energy and their counterparts in other countries. Most of these efforts have focused on system-level and cybersecurity threats; few are attempting to address software vulnerabilities or malicious code embedded during software's development.

## About the Author


**Karen Mercedes Goertzel,** Certified Information Systems Security Professional, leads Booz Allen Hamilton's Security Research Service. She is a subject matter expert in software assurance, cybersecurity, and information assurance. She was lead author of "Software Security Assurance: A State-of-the-Art Report" and "The Insider Threat to Information Systems," published by the Defense Technical Information Center. She has advised the Naval Sea Systems Command and the DHS Software Assurance Program; for the latter, she was lead author of "Enhancing the Development Life Cycle to Produce Secure Software." Goertzel was also chief technologist of the Defense Information Systems Agency's Application Security Program, for which she co-authored a number of secure application developer guides. She also tracks emerging technologies, trends, and research in information assurance, cybersecurity, software assurance, information quality, and privacy. Before joining Booz Allen Hamilton, Goertzel was a requirements analyst and architect of high-assurance trusted systems and cross-domain solutions for defense and civilian establishments.

**Booz Allen Hamilton**
**8283 Greensboro DR**
**H5061**
**McLean, VA 22102**
**Phone: (703) 902-6981**
**Fax: (703) 902-3537**
**E-mail: goertzel_karen@bah.com**