Scale: System Development Challenges

Carol Woody
Robert J. Ellison

May 2007

ABSTRACT: The usage and characteristics of large systems or systems of systems can challenge many current development assumptions. Vulnerability analysis has typically concentrated on vulnerabilities induced by errors in coding or in the interfaces among components. System interactions can also be a seedbed for vulnerabilities, however. This article describes software assurance challenges inherent in networked systems development and proposes a structured approach to analyzing potential system stresses using scenarios.

INTRODUCTION

The following quote from Trust in Cyberspace notes the difficulty for solving some of the integration problems associated with deploying networked information systems (NISs) [Schneider 99]:

NISs pose new challenges for integration because of their distributed nature and the uncontrollability of most large networks. Thus, testing subsets of a system cannot adequately establish confidence in an entire NIS, especially when some of the subsystems are uncontrollable or unobservable, as is likely in an NIS that has evolved to encompass legacy software. In addition, NISs are generally developed and deployed incrementally. Techniques to compose subsystems in ways that contribute directly to trustworthiness are, therefore, needed.

The theme for the 2007 International Conference on COTS-Based Software Systems (ICCBSS) was "Systems' Composition and Interoperability – A World in Transition." The title of Steve Esterbrook's 2007 keynote address was "Scale Changes Everything: Understanding the Requirements for Systems of Systems." The scale that may be encountered in new systems can fundamentally change how system security is addressed in system development.

Systems vulnerability analysis can require different techniques than those used to analyze source code and components for vulnerabilities. Vulnerabilities can result from unanticipated interactions among systems or among systems, users, and system operators. A successful attack on a system can indirectly affect systems that exchange information with the exploited system and particularly those that assume a trusted relationship with that system.

Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh, PA 15213-2612

Phone: 412-268-5800
Toll-free: 1-888-201-4479

www.sei.cmu.edu

Divide and Conquer - Software and Systems Engineering. The expanded system scale that we can achieve with software integration also challenges the historical separation of systems and software engineering. Software issues that arise if we maintain that separation are discussed this section.

- **System Characteristics.** The usage and characteristics of large systems or systems of systems raises analysis issues that may not be addressed by existing techniques.
- **Effects of Scale on Security.** The integration of systems of systems, as well as technologies such as web services, change some of the design assumptions for security.
- **Failure Analysis: What Can Go Wrong.** Four types of system integration errors are described.
- **Failure Analysis: System Behavior.** Systems interaction failures are a source of security vulnerabilities. Usage and system integration influences the failures that should be considered.
- **An Approach to Failure Analysis and Management.** The 2003 failure of the National Power Grid is used as an example of how failures in software systems are increasingly a result of a combination of user, software, and system management errors.
- **Systems: Consolidating Aspects of Security with General Failure Management.** A successful cyber attack on a system can generate faults for other systems. Those systems with the secondary failures typically do not have knowledge of the actual cause and have to apply general mitigation tactics.
- **Failure Analysis Using Scenarios.** The use of scenarios provides a structured approach for analyzing potential system stresses.
- **Concluding Observations.** Continual monitoring and analysis of system behavior are essential.

DIVIDE AND CONQUER - SOFTWARE AND SYSTEMS ENGINEERING

The earlier quote from Trust in Cyberspace noted the need for techniques to compose subsystems in ways that contribute directly to trustworthiness. We build a large system by first decomposing it into pieces that are more easily managed. The challenge is to decompose the system in such a way that those individual pieces can be individually implemented and that the composition of those components meets system requirements.

The decomposition is often constrained by a need to integrate legacy systems, a requirement to use a commercially available component, or a desire to reduce costs by reusing available components. Decomposition can be even more diffi-

cult if we have too many cooks in the kitchen. For large systems the systems and software engineers are both decomposing large components to better manage development, and it is not unusual to find that both groups have to work around decompositions initiated by the other group. The early decompositions that often have to be done with only limited system understanding can create problems later in the development cycle when the detailed interactions among components are better understood.

The separation of systems and software engineering is historical. Boehm notes several trends that caused systems engineering and software engineering to initially evolve as largely sequential and independent processes [Boehm 06]. Systems engineering began as a discipline for determining how best to configure various hardware components into physical systems such as ships, railroads, and defense systems. The objective of systems engineering was to create a sequence of independent tasks to produce the components that would be assembled into the system. Early software engineering drew on a highly formal and mathematical approach to specifying software components. The software and systems engineering worlds co-existed as separate entities. As software became a system component, the systems engineer added additional independent tasks for the software components. The software engineer was not involved in the analysis or allocation of the system requirements.

As systems become increasingly software intensive, there is an increasing need to integrate the two disciplines. Systems of systems can be a mix of software and hardware systems. The electric power grid as described in [US-Canada 04] is such an example, but systems of systems are increasingly a software creation. Networking and distributed system integration techniques such as web services make it much easier to compose systems. Boehm has an extensive discussion of future trends and their impact on systems and software engineering [Boehm 06]. Maier's paper includes a number of examples of the development issues that might arise when the software and systems engineering activities are not well integrated [Maier 06].

A systems engineer might describe a software interface between systems components in terms of the data being exchanged and the communications protocol used. The software engineer concentrates on the behavior of the data flow. How is the data flow controlled? Is it synchronous or asynchronous? Do we have data push or data pull? If we have data push, does the data arrive at regular scheduled times or at irregular times with varying volumes? Those differences can lead to significantly different software architectures.

Maier also discusses other differences in how systems engineering and software engineering are currently practiced. Canonical systems engineering has a flavor

of the waterfall development model. The functional and often performance considerations lead to a top-down decomposition of a complex system into subsystems and eventually decompose subsystems into hardware and software components. Ideally each step is completed before proceeding to the next one. Maier observes that the idealized system decomposition process breaks down when it confronts reality. While the system has to deal with both hardware and software, the software costs can account for 80% or more of the total development and integration budget. Software development increasingly uses an incremental development model, which may postpone some development decisions that a systems engineer would have made earlier in the design.

SYSTEM CHARACTERISTICS

A system of systems (SoS) is a good example of the problems noted in the Trust in Cyberspace quote in the opening paragraph. The set of characteristics that distinguish an SoS from a large monolithic system are

- **operational independence of the elements:** Component systems are independently useful.
- **managerial independence of the elements:** Component systems are acquired and operated independently; they maintain their existence independent of the SoS.
- **evolutionary development:** The SoS is not created fully formed but comes into existence gradually as usages are refined, new usages developed, and old usages phased out.
- **emergent behavior:** Behaviors of the SoS are not localized to any component system. The principal purposes of the SoS are fulfilled by these system behaviors rather than component behaviors.
- **geographic distribution:** Components are so geographically distributed that their interactions are limited primarily to information exchange rather than exchanges of mass or energy [Maier 98].

The SEI considered the affects of the characteristics of an ultra-large system (ULS) on development [SEI 06]. Many of those characteristics are applicable to systems that we are building today:

- **continuous evolution and deployment:** There will be an increasing need to integrate new capabilities into a system while it is operating. New and different capabilities will be deployed, and unused capabilities will be dropped; the system will be evolving not in phases, but continuously.

- **heterogeneous, inconsistent, and changing elements:** A system will not be constructed from uniform parts: there will be some misfits, especially as the system is extended and repaired.
- **erosion of the people/system boundary:** People will not just be users of a system; they will be elements of the system, affecting its overall emergent behavior.
- **normal failures:** Software and hardware failures will be the norm rather than the exception.

The term ULS suggests a large system in terms of its "size" measured, for example, by the total lines of source code. Easterbrook proposed in his ICCBSS keynote that ultra-large could also refer to the scale of usage. Applications that support collaboration such as email, chat, or Word are not ultra-large in a "physical" sense but can have ultra-large effects across an ultra-large population of users and organizations.

It is also valuable to be flexible in how we think of an SoS. A discussion over whether a specific system satisfies Maier's defining characteristics can evolve into a debate over interpretations of the characteristics that cannot be resolved. While an SoS is defined in terms of five characteristics, it is productive to consider how each characteristic affects development to explore at least qualitative measures for individual characteristics. For example, Maier distinguishes two kinds of administrative structures for an SoS [Maier 98] that reflect how managerial or operational independence might be measured:

- **directed:** Directed systems are those in which the integrated system of systems is built and managed to fulfill specific purposes. It is centrally managed during long-term operation to continue to fulfill those purposes and any new ones the system owners may wish to address. The component systems maintain an ability to operate independently, but their normal operational mode is subordinated to the central managed purpose. For example, an integrated air defense network is usually centrally managed to defend a region against enemy systems, although its component systems may operate independently.
- **collaborative:** Collaborative systems are distinct from directed systems in that the central management organization does not have coercive power to run the system. The component systems must, more or less, voluntarily collaborate to fulfill the agreed upon central purposes. The Internet is a collaborative system. Agreements among the central players on service provision and rejection provide what enforcement mechanism there is to maintain standards.

Even for a directed system, any existing legacy and COTS components, as well as distributed operations procedures, can constrain the desired central controls that can be implemented.

As systems expand in scale, evolution is a factor that could be considered independently of the other SoS characteristics because of the costs associated with replacing a large system.

EFFECTS OF SCALE ON SECURITY

The increased deployment of distributed systems using technologies such as web services challenge some of the assumptions used in the development of standalone systems. For a collaborative SoS as defined by Maier, the designer should not expect to have full knowledge of the global system state and should not expect either participating systems or the collaborative SoS to share common risk profiles. A user of a collaborative SoS should have a caveat emptor perspective, particularly with respect to a security failure that compromises any of the participating systems.

Unfortunately, a caveat emptor attitude may be also appropriate for a directed SoS or a complex non-SoS system. As the scale increases, it may not be practical to maintain the global-state information, and the capability to effectively analyze that global state diminishes more rapidly than the capability to gather the information. With the continued dissolution of the organizational system boundary, the sources of vulnerabilities, unexpected events, and changes in technology and usage are increasingly external and beyond the immediate control of the organization. A response to an unexpected event now has to be chosen with incomplete knowledge about the cause of the event and the corresponding global state, with the increased likelihood of false positives and negatives for the necessity of an action. The complexity raises the cost of controls for managing software errors. While we can have system reliability that is better than the reliability of the individual hardware components, the necessary controls and redundancy have a continuing cost. Component redundancy typically addresses hardware failures. The mitigation mechanisms for external sources of software errors such as application proxies can be complex. The number of such controls proliferates as a system usage expands.

A security risk analysis considers the risks associated with specific organizational usage. As we build systems that can be adapted for changes in usage, we can significantly change the ordering of the risks and the desired mitigation strategies for those risks. We likely would be wary about using a collaborative system for a highly sensitive task, but many of the same issues can arise when usage

changes in a centrally managed system. Although the adaptable functionality of a system may enable the execution of a highly sensitive task, the implementation of that functionality may have been designed for usage with limited security needs and cannot meet the new security requirements. Legacy systems and frequently COTS components are a good example of this situation.

Design paradigms such as service-oriented architectures and technologies such as web services can change software control assumptions that have been previously been used. The interface to a remote service is frequently asynchronous so as to avoid either party from having to tie up resources waiting for a response. The initiator of a web-services-based transaction wants to control future access to that data and may want the source of any response to be authenticated. The organization that receives that transaction would want to verify that the transaction was authorized. The web services security protocols support such distributed control through mechanisms such as encryption, public and private keys, and signing. Security Assurance Markup Language (SAML) enables the sharing of user authentication, entitlement, and attribute information. The data as well as the rules for managing that data are described using XML. The security policy is now attached to the data, and that policy is now interpreted by multiple control points.

Shared business services such as those proposed by a service-oriented architecture (SoA) can complicate the tradeoffs among requirements that are made during development. General functional interoperability is not sufficient. An SOA can encounter the same problem with a change in usage as was described for a collaborative system. A shared service can be used in multiple contexts. A shared service may be the common access point for an enterprise data set and in that role enforces the desired authentication and authorization requirements. Initially the request for that service may have always come from a trusted computing platform such as an enterprise-maintained desktop, but over time usage has changed to allow access from a remote laptop or a PDA, which introduces new risks that were not considered by the initial service design. As use of shared services expands, there is greater likelihood of mismatches in assumptions among components. For example, a shared service whose response to an adverse event is a graceful shutdown creates an event (the loss of that service) that may not be well tolerated by all systems that use the provided service.

FAILURE ANALYSIS: WHAT CAN GO WRONG

For large software projects it is not unusual to find that the reliability development activities have addressed hardware failures. Hardware reliability improve-

ment activities identify unexpected failure modes and the stress points that are likely points of failure. Hardware reliability improves as these design defects are mediated. Software reliability typically receives less effort. Software failures such as race conditions and system deadlocks over the use of shared resources receive attention, but in general predicting and then improving the reliability of a software design before it has been implemented is not a standard practice. Software reliability efforts are more likely to focus on modeling defect trends identified via code reviews and testing. Finally, the hardware and software reliability analysis each assumes that the reliability of the other factor is 100%, and there is little consideration of the interaction of software with non-computer hardware or with operations.

Part of the problem is that software errors can be difficult to identify. All software failures are deterministic in the sense that they occur every time certain conditions are met. For most errors we can find a normal use case that repeats the failure, but errors such as race conditions or memory leaks are much more difficult to reproduce because they depend on non-deterministic timing and usage patterns and often appear to be random. As we move from components to systems and systems of systems, the essential properties may be emergent, that is, derived from the collective behavior of the components rather than from a single component.

A failure in that emergent behavior can occur even though each component meets its specifications. Leveson separates safety accidents into two types: those caused by failures of individual components and those caused by dysfunctional interactions between non-failed components. In most software-related accidents, the software operates exactly as specified, that is, the software, following its requirements, commands component behavior that violates system safety constraints, or the software design contributes to unsafe behavior by human operators (i.e., system-level analysis identifies multiple contributing factors rather than a single failure) [Leveson 05].

Failure analysis associated with malicious events has to deal with additional complexity.

- Hardware reliability analysis often assumes that multiple faults are independent. However, an attacker can purposely inject multiple events, such as disabling both an essential service and the recovery mechanisms.
- There is no measure for intentional acts equivalent to the mean-time-to-failure measure used for hardware failures. While severe or catastrophic attacks can be speculated, the probabilities of such events may be quite small and depend not so much on system attributes but on the likelihood of per-

sonnel, political, or financial circumstances that would motivate a skilled attacker to act.

- Detecting a security fault can be more difficult than sensing a software fault. A skilled attacker may be able to identify the security detection mechanisms employed and tailor the attack appropriately to avoid detection. This punch and counter-punch response pattern exists now between those who write email viruses and those who design the detection algorithms for email scanners. Detection may attempt to identify abnormal behavior, but normal system usage evolves over time and requires continual changes to the criteria.

As we deploy larger systems, the software development effort concentrates more on integrating existing components or systems than on building new ones. The article Trustworthy Composition: The System Is Not Always the Sum of Its Parts considered four software artifacts in the analysis of how software vulnerabilities might be introduced as a system is assembled from its components.

1. **Specific interface.** An interface controls access to a service. Interfaces that fail to validate the input are frequent members of published vulnerability lists.
2. **Component-specific integration.** Assembly problems often arise because of conflicts in the design assumptions for the components. Project constraints may require using components, COTS software, or legacy systems that were not designed for the operating environment, which raises the likelihood of mismatches. The increasing importance of business integration requirements compounds the component integration problems and is the motivation for designs based on SOA.
3. **Architecture integration mechanisms.** Commercial software tool vendors often provide the capability for the purchaser to integrate the tool into their systems and tailor its functionality for their specific needs. However, the capability to reconfigure a system rapidly is matched by the increased probability of component inconsistencies generated by the more frequently changing component base, as well as the increased risk that the dynamic integration mechanisms could be misused or exploited. These mechanisms represent another interface that must be properly constrained.
4. **System behavior: component interactions.** The behavior of a system is not the simple sum of the behavior of the individual components. System behavior is strongly influenced by the interactions of its components. Components may individually meet all specifications, but when they are composed into a system, the unanticipated feedback among components can lead to unacceptable system behavior. Security and safety are system rather than component requirements. We can build a reliable system out of unreliable components by appropriate use of redundancy. Components that are

not secure as standalone components in an operating environment may be secure when used within the constraints maintained by a system.

A primary objective for the Build Security In web site is to reduce the number of vulnerabilities in a system. Good progress has been made on some classes of vulnerabilities. Static analysis tools should reduce the vulnerabilities introduced by poor software coding practices. The documentation of attack patterns has identified vulnerabilities associated with specific interfaces and technologies. Tools exists that can support the analysis for popular interfaces such as those associated with web development. Vulnerabilities associated with architecture integration or extensibility such as web services and scripting require attention. Vulnerabilities associated with system behavior are likely to be increasingly important with the deployment of systems of systems, as well as with the use of shared services as in SOA. The following quote from Trust in Cyberspace notes the importance of system interactions and the scarcity of effective analysis methods.

> *System-level trustworthiness requirements are typically first characterized informally. The transformation of these informal notions into precise requirements that can be imposed on individual system components is difficult and often beyond the current state of the art. Whereas a large software system such as an NIS cannot be developed defect-free, it is possible to improve the trustworthiness of such a system by anticipating and targeting vulnerabilities. But to determine, analyze, and, most importantly, prioritize these vulnerabilities, a good understanding is required for how subsystems interact with each other and with the other elements of the larger system. Obtaining such an understanding is not possible today [Schneider 99].*

FAILURE ANALYSIS: SYSTEM BEHAVIOR

Business demands increasingly lead to work processes that span multiple systems and, in some cases, multiple organizations. Given the difficulty of that integration, it is not unusual to find that the initial development focus is on simply getting the processes to work. The initial design is often driven by what are often called “sunny day” scenarios, that is, scenarios that assume close to ideal operating conditions. Such ideal conditions are rarely the norm for networked systems and are even less likely if the work processes span multiple organizations.

The security professional can have two very different roles during development. One role involves the design and development of security functionality that supports authorization, authentication, auditing, and cryptology. The second role

deals with identifying potential security vulnerabilities associated with the design, development, usage, and management of the system. The use of a central data store creates an attractive target for an attack. The use of particular communication protocols may require analysis to ensure that best practices were followed.

The increased scale of business systems naturally leads to the need to scale up security services. Business governance policies, as well as regulatory requirements such as Sarbanes-Oxley, have led to the consolidation of authorization and authentication services across multiple computing platforms and applications. The centralization of such security services reduces the likelihood of homegrown and error-prone implementations, but those central services are also an inviting attack target. As single points of failure for most enterprise systems, consolidated services need to meet high standards for security, fault tolerance, and availability.

The system complexity associated with business integration requirements expands the spectrum of development, user, and system management failures that security analysis has to consider. One way to partition that effort is to consider two perspectives for the analysis of work processes that span multiple systems:

- **End-to-end perspective:** How do we demonstrate the quality requirements for a multistep process? Each system may have its own authorization and auditing requirements. Does the composition of the functions associated with the individual systems satisfy the end-to-end security requirements for the work process? For distributed systems, external visibility and controls may be limited or nonexistent.
- **Service provider perspective:** A multisystem work process can create usage patterns or risks that were not anticipated in the design of a specific system. The system owner may have requirements for the protection of the system assets that conflict with the desired usage. The evolution of usage that is associated with an SoS implies that eventually abnormal behavior will occur.

Those two perspectives reflect the fact that distributed decision making across both physical and organizational boundaries is a necessity for software-intensive, complex, human-machine systems. As work processes extend beyond the corporate IT perimeter and incorporate services and data provided by external systems, the concept of a perimeter becomes even more elusive. Frequently each interface must be monitored to reflect the dynamically changing assurance associated with that link. Thus, the central control represented by a firewall-protected perimeter has increasingly been replaced by multiple control points.

Multiple systems are integrated to support a work process with the assumption that those systems are trustworthy. By trustworthy, we mean that there is evidence that the systems have predictable behavior and are free of vulnerabilities. But, for a number of reasons, predicting system behavior and identifying potential vulnerabilities that could arise from component interactions are difficult for systems of any size and are more difficult when integrating multiple systems:

- Risks may not be observed until deployment.
  - Modeling cannot consider all the factors.
  - The effects of a system change or failure are difficult to contain and with external interdependencies may be unknown.
  - An insignificant local change or error can be magnified by unexpected or poorly understood interdependencies.
- The difficulty of predicting the behavior of an SoS is compounded by multiple and often autonomous controls.
- The sheer number and diversity of software and hardware components limits the capability to synchronize upgrades and deployments. Such synchronization is even less likely across multiple organizations. Interoperability is often required with legacy components and components with unapplied upgrades.
- There is reduced visibility of the state of the participants in an SoS. A successful exploit of a normally trusted system may not be recognized by the other systems.
- Subsystems and software components are designed for specific usage in terms of performance, reliability or security. With continuing changes in business usage, new business usage eventually will generate system or component usage that falls outside the design parameters.

As an example of the last item, consider Figure 1, in which the ovals on the right side represent geographically distributed systems and the blue and black lines are business processes that use those systems. The right side of the figure expands one of those systems. For a military example, an oval might be a specific Service system, whereas the work process might be joint activity that required coordination across the Services. The specific Service system receives both joint and Service-specific requests. A joint Service activity would likely generate a sequence of actions similar to the actions generated for a Service-specific request.

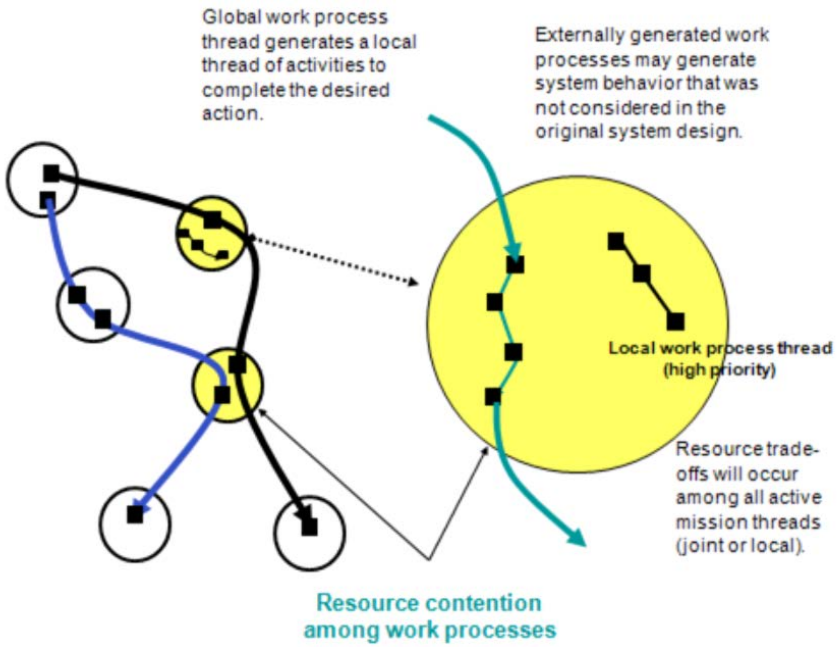

Figure 1. System-of-systems resource contention

AN APPROACH TO FAILURE ANALYSIS AND MANAGEMENT

Causation for Software Systems

The error analysis for a specific interface, for component-specific integration, or for architectural integration mechanisms is often similar to what is called event-chaining in safety engineering [Leveson 05]. The assumption for almost all causal analysis for engineered systems today is a model of accidents that assumes they result from a chain (or tree) of failure events and human errors. From an observed error, the analysis backward-chains and eventually stops at an event that is designated as the cause. Usually a root cause selected from the chain of events has one or more of the following characteristics: (1) it represents a type of event that is familiar and thus easily acceptable as an explanation for the accident, (2) it is a deviation from a standard, (3) it is the first event in the backward chain for which a "cure" is known, and (4) it is acceptable as the identified cause [Leveson 05].

Event-based models of accidents, with their relatively simple cause-effect links, were created in an era of mechanical systems and then adapted for electro-mechanical systems. The use of software in engineered systems has removed many of the physical constraints that limit complexity and has allowed engineers

to incorporate greatly increased complexity and coupling in systems containing large numbers of dynamically interacting components. In the simpler systems of the past, where all the interactions between components could be predicted and handled, component failure was the primary cause of accidents. In today's complex systems, made possible by the use of software, this is no longer the case. The same applies to security and other system properties: While some vulnerabilities are associated only with a single component, a more challenging class of vulnerability emerges in the interactions among multiple system components. Vulnerabilities of this type are system vulnerabilities and are much more difficult to locate and predict [Leveson 05]. Leveson's argument may be applicable to software assurance for complex systems. Attacker tactics evolve in response to changes in defensive strategies, system usage, and technologies deployed, and future tactics could exploit system interactions rather than just component vulnerabilities.

National Power Grid Failure in 2003

The 2003 Electric Power Grid blackout is a representative example of a system-of-systems failure. While this example deals with reliability rather than security, much of the analysis is applicable to security because both security and reliability must manage adverse events. While the failure was not the result of malicious activity, it provides a good example of how errors in failure management could be exploited. The failure did not have a single root cause but was the result of a combination of system, user, and operational errors. This example also demonstrates how risk mitigation is often a combination of system, user, and operator actions.

On August 14, 2003, approximately 50 million electricity consumers in Canada and the northeastern U.S. were subject to a cascading blackout, the largest in North America's history. There was not a single cause for this event. The final report by the U.S.-Canada Power System Outage Task Force identified a variety of contributing factors including poor communications, human error, mechanical breakdowns, inadequate training, software errors, poor performance for sophisticated computer modeling systems, and lack of attention given to simple tree trimming. The details for this example are not that critical. Rather the example demonstrates how the reliability of SoS can be affected by the interactions of computer systems, physical systems, and human operations [US-Canada 04].

The blackout was initiated when three high-voltage transmission lines operated by an Ohio utility short-circuited and went out of service when they came into contact with trees that were too close to the lines. The loss of the three lines resulted in too much electricity flowing onto other nearby lines, which caused those lines to overload and then be automatically shut down.

The control system included four subsystems, one of which generated alarms based on the severity of events. An operator has to monitor a significant amount of information—think in terms of having to browse through multiple sheets in a spreadsheet with no search function—and an alarm would direct the operator's attention to a particular item.

The control system is used by a number of utilities, and a race condition arose for the first time, which disabled the alarm subsystem. The alarm subsystem was replicated to mitigate hardware faults. The initial failure corrupted the data stream so that the second server also failed on startup. The subsystems were integrated in such a way that the alarm subsystem could be restored only by a cold restart of the full control system, a thirty to sixty minute operation. The alarm subsystem failure did not interrupt the data feeds from the power-grid sensors to the other system components. The control operators had access to all information, but without the alarms they would have had to identify abnormal conditions manually.

The power-grid operators were not automatically notified by the control system of the failure of the alarm subsystem. The initial symptoms of this failure were confusing, and IT operations did not immediately know that the alarm subsystem had failed. IT operations delayed notifying the control operators of the failure and did not appear to understand the operational impact of the alarm subsystem failure. The final report of power system outage task force raised issues with operator training with respect to managing emergency conditions. Hence, for a significant time, the power-grid operators were not aware of the loss of the three lines and hence took no action, such as shedding load, which could have kept the problem from growing and becoming too large to control.

A primary power grid monitoring tool is a state estimator, which takes measurements of quantities related to the system state as input and provides an estimate of the system state as output. It is used to confirm that the monitored electric power system is operating in a stable and reliable state by simulating the system both at the present time and one step ahead for a particular network topology and loading condition. With the use of a state estimator and its associated contingency analysis software, system operators can review each critical contingency to determine whether each possible future state is within reliability limits. A state estimator might run every five minutes. When the output from a state estimator is out of bounds (i.e., generates an impossible state), it usually means that there is an inconsistency in the input. To correct that situation, the state estimator is taken off-line and the input data verified.

There are independent monitoring facilities for the power grid. Midwest Independent System Operator (MISO) coordinates power transmission in the affected

region, and utilities exchange alerts on critical failures. MISO uses a state estimator that runs every five minutes as one of the tools to monitor the grid. On August 14, MISO state estimator produced output that was far outside the acceptable range. The failure was caused because the status of an out-of-service line had not been updated for the state estimator. After that state was corrected, MISO's IT operations erred initially by forgetting to re-enable the automatic five minute runs. When that operational error was observed and the five minute runs enabled, a second failure occurred caused by invalid status information on another line. The state estimator finally came back on line in an operational mode about two minutes before the unrecoverable cascade of failures started.

Observations

(These observations are those of the author and not of the task force that reviewed the blackout.)

The power failure shows the need to demonstrate assurance not just for a system but also for business operations, training, and IT operations. The failure of the alarm subsystem is a significant business continuity risk. The mitigation of that risk could have included

- automatic notification of grid controllers and IT system operations of a subsystem failure
- incorporation of multiple restart options for the alarm service in the design. The only restart available assumed the validity of the data queued for processing by the alarm system. Are there alternatives that would provide partial functionality for the grid controllers?

This example reflects a failure with service availability. The task force final report noted that none of the events leading to blackout had malicious intent. But the failure also demonstrates the sensitivity of the power grid's SoS to data integrity (MISO's state estimator) and the effects of a software failure on an essential system service (the alarm system race condition). Attackers, particularly those with some inside knowledge, might be able to stress the system with multiple independent external "accidents."

The monitoring of the grid requires that organizations like MISO have knowledge of the loading and available capacity of a utility. MISO has to recommend actions based on its confidence in the utility data feeds. That same information would be valuable to a utility's competitors, since it might give them an advantage in pricing electricity. These confidentiality and integrity requirements involve multiple organizations. A utility has to have confidence that MISO protects the information and uses it appropriately without the ability to monitor MISO's IT operations.

SYSTEMS: CONSOLIDATING ASPECTS OF SECURITY WITH GENERAL FAILURE MANAGEMENT

The Introduction to System Strategies article notes that the multiplicity of systems and increasing number of possible error states arising from system interactions can overwhelm security and reliability analysis. Separating the analysis for reliability, dependability, and security may compound the complexity. With respect to the power grid blackout, the mitigation for the alarm subsystem failure would have to have been chosen without knowledge of the actual cause of the failure. The use of only server redundancy suggested that the analysis concentrated on hardware failures.

The use of a second server consolidates the management of hardware failures. It does not matter if the failure was with a power supply or with memory. In either case the mitigation replaced the entire unit.

The consolidation of failure mitigations is a necessity for software also. The immediate diagnosis of a cause may be impossible, and the choice of mitigations is based initially on the effects. Threats, vulnerabilities, unexpected events, and changes of practice—for processes, people and technology—can have similar effects on business continuity. With respect to security, intrusion detection components may be able to identify an attack directed at a specific system. Intrusion detection is less effective when the effects are indirect, such as an attack that compromises a normally trusted data source. The effects of an indirect intrusion are often indistinguishable from the effects of a user or software error on that remote system. For example, was the failure of the alarm server in the power grid example caused by an internal failure (race condition in this instance), by extremely high volumes of data, or by corrupted data? Was a high volume of data or a corrupted data stream caused by external system or sensor failures, by malicious activity, or by a significant number of actual events?

The analysis of alarm service failures for the power grid might have considered causes of the server failures:

- Hardware failure and some software errors – Mitigate with a redundant server, which covers server hardware failures as well as some server software errors.
- Input data stream errors – Multiple mitigations: (1) Restart server without using the saved data queue. The data had been processed by other subsystems. Can the alarms be generated by analysis of the existing power grid state information? (2) Have a separate subsystem monitor and store the last N hours of the input stream, which can then be replayed.
- Poor diagnostic support for system management – The data stream from the external monitors to the control system appeared to be syntactically correct

because that data was processed without errors by other subsystems. That information would suggest the possibility of an error in the queued data used to restart the redundant server. IT operations require guidance on how to interpret such events so as to support recovery.

Failure Analysis Using Scenarios—Describing Stresses

Because identifying vulnerabilities associated with system interactions is difficult (as noted by Leveson and in the quote from Trust in Cyberspace), it is essential that subsystems and components are as vulnerability free as possible. Integration may depend on the ability to tailor components or subsystems. The analysis of any such extensibility or configuration mechanisms should have high priority. Web services are an example of mechanisms that can tailor the interface among systems.

System failure analysis should be part of overall risk analysis activities. An essential metric for this analysis is maintaining the traceability to business operations. That traceability can be used in cost-benefit analysis for establishing failure priorities and selection of mitigations.

The initial analysis identifies potential adverse system and operational stresses rather than explicit vulnerabilities. For example, in terms of availability, which kinds of failures could disrupt business operations? For the power grid example, such analysis should have identified the alarm system. The stresses should include those that may arise for the operational administration of the system.

The mitigation of system failures should consider the role of users and system administrative staff. For example, multisystem identity management may be required to meet authorization requirements, but such a security service also means that the effects of an operator error propagate to multiple systems. Increased dependencies among systems also means that it may be more difficult for an operator to know the effects of any actions taken in response to a system failure. An inappropriate operator response may induce additional failures that could be exploited.

Fault events, affects, and responses can be described by scenarios. Scenarios provide a way to document failures and structure the necessary analysis.

Scenario Components

The components of a general scenario are

- **a stimulus:** a condition or event that needs to be considered when it arrives at a system
- **a source of stimulus:** the entity that generated the stimulus

- **an environment:** the conditions under which the stimulus occurs, such as during high usage
- **the affected artifacts:** Examples of artifacts include data stores, services, and software monitoring sensors.
- **response:** the activity undertaken after the arrival of the stimulus
- **response measure:** the attribute-specific constraint that must be satisfied by the response
- **a set of architectural tactics:** the architectural tactics that could be used to achieve the desired response

We need to define some terms.

Failure: Occurs when the system does not deliver its expected service (as specified or desired). A failure is externally observable.

Error: An internal state leading to failure if the system does not handle the situation correctly.

Fault: The cause of an error, which may lead to a failure.

For security, input that exceeds buffer size is a fault. The error is the violation of the pre-conditions for the function that receives that input. A failure would be ignoring that error and permitting a buffer overflow that was used to increase user privileges.

The analysis of faults has to include the perspective

1. of a multisystem work process thread
2. of a system that contributes to the completion of a business work process

Scenario Guidance

Scenarios need to be sufficiently detailed to evaluate or refine a proposed system or software architecture. Consider modifiability. Measuring the cost of change in terms of development effort could lead to a very different approach than measuring the cost in terms of operational expenditures as downtime or installations.

The specificity of scenarios could also be a liability. A scenario examines a slice of a system. Evaluating a lot of slices does not necessarily validate the whole. The scenarios should be representative of the spectrum of desired behaviors. For qualities such as security and reliability, include scenarios that capture stresses for essential business processes. Such scenarios provide traceability between technical decisions and business usage.

Events

The kinds of events that can be analyzed depend on the visibility we have for the systems. An SoS implementation of a business work process is a multistep activity. The composition of two of those steps or a single step can be a source of faults. If the work process is considered as a transaction, what are the events that could cause a transaction to fail? For a directed SoS such as a military command and control system or an internal business SoS, knowledge of the internal systems and operating procedures can be used to identify a significant number of system and operational failures that might lead to a failure of a transaction that is associated with the work process. Where there is limited or no visibility for the supporting systems, the "owner" of a business process has to be more pessimistic (or maybe that is more paranoid) about expected behavior. In most instances, monitoring and logging activity is necessary to support analysis after the occurrence of adverse events.

Having visibility into internal systems and operations is also a liability. An objective for loose coupling is to avoid using knowledge of internal system structure so that an individual system can be changed without affecting system integration. Using knowledge of internal systems for failure management could limit the ability to make future changes.

Designers for systems that have dependencies on external systems could consider events such as

- events unanticipated in the initial design of the system interfaces. The analysis of interactions is rarely complete, and over time, changes in usage, software, and system management can introduce new behavior.
- events that challenge the trust assumed among systems. A challenging aspect of designing for multiple systems is specifying the trust among the parties. The trust might be reflected in the confidence that the suppliers of services will not exploit that contract, the confidence that the systems can perform the required action correctly and in the allotted time, or the confidence that a system has not been compromised. The trust associated with a particular party or system is dynamic. Events such as delayed responses, errors associated with data exchanges, or reports of internal processing errors could suggest that the trustworthiness of that service should be reviewed.

Designers for systems that support a multiple system work process could consider events such as

- violations of any preconditions for initiating a service request
  - input data not meeting specifications

    - invalid authentication or authorizations. These might be failures in a web service protocol.
    - frequency of usage
- errors that arise completing the requested service
    - Are existing error recovery procedures consistent with those expected by external usage?
    - Are there failure states that now require operator intervention?
- system usage patterns that differ from “normal” usage
    - resource contention
    - reduced capability to meet local service demand for service
    - high priority external requests
    - risks for system managed assets

Responses and response measures

Response objectives include prevention, containment of effects, restoration of the system state to time in advance of the event, and corrective changes in the existing system state. Response measures include cost, effects on current usage, scope of recovery actions in terms of components affected, downtime, and operational costs such as operator effort and training.

For multiple-system work process, there may also be requirements for undoing actions completed in the steps preceding the failure. Hence a multisystem work process may create new recovery requirements for individual systems. An SoS may support sufficient redundancy so that processing can continue. The challenge is to have a set of scenarios that represent a sufficient spectrum of failures. Hardware failures are typically one of the first ones to be considered. Software failures can be harder to predict. The probability a specific software failure may be quite small, which can make it difficult to justify the analysis and mitigation costs. But with the increased integration of business systems, the effects of an error can be widely propagated. As noted by The Burton Group in a client report, an organization that had to deal with a non-malicious denial-of-service event now requires all new applications to have detailed availability requirements.

The evolutionary nature of an SoS generates a number of important secondary measures. The dynamics associated with business processes are an ever present driver. The regulatory environment is changing, as are the associated practices expected by auditors. The risk tolerance of an organization may change in response to regulatory changes, contractual agreements, increased management awareness of the liabilities associated with an activity, or simply the awareness of events that may affect an organization’s reputation. The objective for design patterns such as service oriented architectures is to reduce costs and still enable

change by sharing business services among applications. Grady Booch, in a spring 2006 talk at the SEI, noted that the half-life of architectures he was studying for his Architecture Handbook was around five years. That rate of aging in systems probably reflects the continuing change in requirements. Whereas we can replace a single system when the gap between what it does and what it should do becomes too great, replacing an SoS would normally be too expensive and too disruptive. Yet an SoS is subject to same demands for change.

Organizations increasingly use flexibility as a way to provide resilience (e.g., business continuity) in response to threats, vulnerabilities, unexpected events, or changes in business practices. Flexibility is expensive, and the desired flexibility has to be carefully specified. For the business continuity example in Introduction to System Strategies, flexibility meant being able to easily change software so that if necessary a geographical location could be run as a stand-alone operation or so that the software could be executed at another location. Flexibility can be a security risk. An adaptable system may be changed in ways that now violate the original security assumptions. Extensibility mechanisms such as mobile code and extensible architectures can be exploited by attackers. Scenarios need to include the events that may arise because of the flexibility mechanisms used.

The error management software has to be analyzed for failures. The limited capability to monitor multisystem activities suggests increased likelihood of false positive or negative responses. An essential measure will be the ability to have predictable behavior in response to errors.

Finally, manageability should be a measure that is close to the top of the list. Error management may create a dependency among users, system operators, and the software systems. That dependency has to be reflected in change management and in configuration management. The most critical aspect of manageability may be with daily operations. The dynamics of the SoS system interchanges and the ability to easily create a new SoS work process thread complicates analyzing the cause of a system failure and identifying a recovery that is appropriate for the existing usage. An SoS recovery that involves the recovery of multiple systems or a failure in an automatic recovery can create a state that has not been previously observed by system operators. Manual interventions to recover from a failed system state may not be aware of the multiple dependencies among systems and create a configuration that is inconsistent with usage.

Environment

A scenario could be used to consider

- the criticality of the supported work process

- the context: a specific workflow, a class of users, a specific system configuration, or a specific combination of activities
- the resources available for a response (these could be computing resources or personnel)
- regulatory or legal constraints such as Sarbanes-Oxley that require the consistent enforcement of access and auditing policies across multiple systems

Characteristics of how systems were developed may influence how errors can be managed.

- commercial components: Commercial off-the-shelf software is designed for general usage. The risk assessment would have been general and the priorities more influenced by the vendor's reputational risks than a specific user's risks.
- multiparty developed system: General interoperability is not sufficient. Error management has to be consistent. For example, a system whose response to an adverse event is a shutdown creates an event (the loss of that service) that may not be well tolerated by all systems that use the provided service.
- multiparty cooperative systems: An architectural design for supporting business processes that involve independently developed and administrated systems raises a number of problems. We no longer have knowledge of the global system state. The systems do not necessarily share the same threat profile or risk analysis. A similar set of conditions could exist for loosely coupled internal systems where the global system knowledge may be theoretically possible but impractical to maintain. A response may have to be chosen with incomplete knowledge about the global state, including the cause of the event, with the increased likelihood of false positives and negatives. We may have inconsistent behavior as independent systems respond to the same event based on differing local knowledge and threat profiles.
- shared services: Distributed but shared infrastructure services may have multiple "owners" for each instance of a service and multiple points of control. Such an infrastructure has its own a set of threats and may be used by applications that do not share common threat and risk profiles.

Tactics

The limited experience with systems of systems and the diversity of such systems means that we have a scarcity of proven tactics. Advice is more often available on what not to do.

For example, it is not unusual to observe that early prototypes of web service applications used SSL for confidentiality. SSL is a point-to-point solution and not appropriate from an end-to-end perspective, since a transaction supported by web services may involve multiple hosts. The prototypes should have used the

capabilities provided by web services that were designed for end-to-end processing. The content of a web service message may be dominated by headers rather than data, but those headers have to describe how to manage data access and authentication across one or more sites. A similar technique can be used for an SoS. We could implement a quality of service attribute by labeling data. Such labeling might also provide guidance for error management.

Berg's High Assurance Design: Architecting Secure and Reliable Applications is a source of tactics [Berg 06]. That text has chapters on trust, compositional and transactional integrity, and failure response design and a manageability case study.

Today, loose coupling is often promoted as a means to enable interoperability across diverse platforms, but loose coupling is an essential tactic for maintaining system resilience. Errors in tightly coupled systems with synchronous communications are more difficult to mitigate and hence more likely to lead to a failure. Table 1 lists some of the differences between loose and tight system couplings [Perrow 99]. Error management should avoid techniques that would increase the coupling among systems.

Table 1. Differences between tight and loose system couplings

Tight Coupling	Loose Coupling
Delays in processing not possible	Processing delays possible
Invariant sequences	Order of sequences can be changed
Only one method to achieve goal	Alternate methods available
Little slack possible in supplies, equipment, and personnel	Slack in resources possible
Buffers and redundancies are designed in, deliberate	Buffers and redundancies are fortuitously available
Substitutions of supplies, equipment, and personnel are limited and designed in.	Substitutions are fortuitously available

The evolutionary character of an SoS influences tactics. The choice of tactics should consider future usage. Under what circumstances might that tactic have to

be revised or replaced? The demand for change will come both from the participating systems and from changes in the usage of an SoS.

The mitigation tactics for systems that participate in an SoS, particularly for an SoS that crosses organizational boundaries, may need to have a defensive perspective because of the limited visibility that exists for other participants and of the state of the SoS. Each system should have sufficient autonomy so that its own assets can be protected, but that autonomy may create conflicts between the objectives of a specific system and the SoS. Events that lead to the defensive operational mode for multiple systems could generate an SoS deadlock that requires operational procedures to resolve. The SoS scenarios should consider faults that might lead to deadlocks.

Composing system analysis

Although aircraft control systems are not networked in the manner being considered in this article, the effects of requirement changes for avionic systems on certification raise some of the problems that now confront the software architect for a networked system. The material in this subsection is drawn from an article by John Rushby [Rushby 2002]. While Rushby's approach has a research perspective and may not be immediately practical, his reasoning can be instructive for the practitioner.

The certification of avionic systems involves not only verifying that such systems work correctly but also showing that such systems cannot go badly wrong even when other things break. Aircraft have been certified as a whole. Components are certified only for their application for a specific aircraft. Software on board commercial aircraft has traditionally used federated architectures. Separate computer systems were used to implement functions such as such as the autopilot, flight management, or yaw damping, with redundancy done for each system. The limited interaction and the few shared resources among the separate systems served as a barrier to the propagation of faults and enabled essentially independent certification of each component. The duplication of resources is expensive, however, and the limited interaction among the modules could also limit the functionality that could be provided. Hence there is a move toward integrated and modular architectures in which several functions share a computing resource or where the functions can be composed from smaller components that could be used in different applications.

Rushby notes that we do construct systems from components by reasoning about the properties of the components and interactions among the interfaces. But with traditional design, we are reasoning about normal behavior. Certification has to deal with abnormal operation. How does the malfunction of a component affect the larger system? Unfortunately, the hazards may not respect the boundaries

that were defined for normal operation. For example, the requirements for aircraft tires for normal operation can be derived from the knowledge of the aircraft in terms of weight, landing speed, and length and from thermodynamics in terms of modeling the heat buildup. On the other hand, a tire failure can create new interfaces among aircraft components because tire fragments can puncture a fuel tank.

Rushby explores how we might certify individual components so that modular certifications could be composed to form a system certification. While certification may be a requirement for only a few systems, his approach may suggest ways to reason about the behavior of complex systems during abnormal conditions. The approach has three key elements.

- partitioning
- assume-guarantee reasoning
- separation of properties into normal and abnormal

The objective for partitioning is to avoid effects that are outside the defined interface by protecting the computational and communications environment perceived by non-faulty components. Assume-guarantee reasoning is the technique that allows one component to be verified in the presence of assumptions about another, and vice versa. Rushby extends the technique to consider abnormal properties. How does a component behave when components that it interacts with fail in some manner? Given a particular set of fault assumptions, can a component guarantee a specific level of service? The analysis needs to deal with the domino effect. If some set of faults reduces component A guaranteed service from level 1 to 2, then that change may lead to a reduction in the level guaranteed by component B. We have a domino effect if the latter reduction leads to a further reduction in the guaranteed service level for A.

CONCLUDING OBSERVATIONS

Need to Know

This article has discussed the importance of understanding the effects of the increasingly complex operating environment on system architecture. We have not discussed how widespread that knowledge has to be in the architecture. For example, an early strategy for distributed computing was to hide the networking faults from the application developer. The computing infrastructure can provide a generic response to faults, but usually only the end-points of a transaction are able to initiate a response that addresses the criticality of a function and how the

specific fault affects that functionality. The organization that had a self-inflicted denial of service added reliability requirements to the applications, the end points.

The need to know has to be balanced against maintenance costs. The operating environment, the business requirements, and the business and technical guidelines for responses to high-risk events will change. A frequently applied architectural tactic is to encapsulate the aspects of a design that are subject to continued change. Another approach is to bind such choices as late as possible in the development process so that they can be more easily changed. The organization that implemented the business continuity example in Introduction to System Strategies used the latter approach.

Monitoring

As the operating environment becomes less predictable, we need to continually monitor for abnormal behavior and provide appropriate diagnostics. The initial objective is not to support a real-time response or to share that information across systems. Rather, the later analysis of such data may observe the increasing frequency of conditions that had not been anticipated or support the analysis of a system after a failure.

REFERENCES

[Berg 06] Berg, Clifford J. High Assurance Design: Architecting Secure and Reliable Enterprise Applications. Upper Saddle River, NJ: Addison-Wesley, 2006.

[Boehm 06] Boehm, Barry. "Some Future Trends and Implications for Systems and Software Engineering Processes." Systems Engineering 9, 1 (Spring 2006): 1-19.

[Leveson 05] Leveson, Nancy G. "A Systems-Theoretic Approach to Safety in Software-Intensive Systems." IEEE Transactions on Dependable and Secure Computing 1, 1 (January-March 2004): 66-86.

[Maier 06] Maier, Mark W. "System and Software Architecture Reconciliation." Systems Engineering 9, 2 (Summer 2006): 146-158.

[Maier 98] Maier, Mark W. "Architecting Principles for Systems-of-Systems." Systems Engineering 1, 4 (Winter 1998): 267-284.

[Perrow 99] Perrow, Charles. Normal Accidents: Living with High Risk Technologies. Princeton, NJ: Princeton University Press, 1999 (ISBN 0-691-00412-9).

[Rushby 02] Rushby, John. Modular Certification (CSL Report). Menlo Park, CA: SRI International, 2002.

[Schneider 99] Schneider, Fred B., ed. Trust in Cyberspace. Washington, DC: National Academy Press, 1999.

[SEI 06] Software Engineering Institute. Ultra-Large-Scale Systems: The Software Challenge of the Future (2006).

[US-Canada 04] U.S.-Canada Power System Outage Task Force. Final Report on the August 14, 2003 Blackout in the United States and Canada: Causes and Recommendations. April 2004.

This material is based upon work funded and supported by Department of Homeland Security under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center sponsored by the United States Department of Defense.

DM-0001120