Safety and Security Considerations for Component-Based Engineering of Software-Intensive Systems

1 February 2011

Karen Mercedes Goertzel, CISSP[1]

Theodore Winograd, CISSP

Booz Allen Hamilton

Jointly funded by:

Assistant Secretary of the Navy Chief System Engineer

and

Naval Ordnance Safety and Security Activity (NOSSA)

---

[1] With assistance from Anthony Gottlieb, Ph.D.

TABLE OF CONTENTS

1. INTRODUCTION

1.1. PURPOSE

Software engineering has become a significant part of the overall engineering of modern safety-critical systems, ranging from weapons systems, to avionic and other vehicle control systems, to industrial control systems, to medical devices. In these systems, Commercial Off-The-Shelf (COTS), open source, and other non-developmental software is used to monitor and control physical processes the failure of which could lead to loss of human life or other catastrophic mishaps.

Control of weapon systems requires the Navy system and software developers to fully understand and develop new techniques for assessing and mitigating the safety hazards and security risks to Navy weapon systems imposed by these imperatives. The purpose of this white paper is to discuss the safety hazards that can arise in safety-critical component-based software-intensive systems (also known as "software-reliant systems") such as weapons systems, as well as the security risks that can result in safety mishaps (*i.e.*, "safety-impacting security"). The paper also discusses assessment and analysis techniques that can be used to pinpoint and assess such hazards and risks, and architectural engineering countermeasures that can be used mitigate those that cannot be avoided or eliminated.

Specifically, the paper discusses:

- The types of anomalous, unsafe, and non-secure behaviors that can emerge when components interact in component-based systems;
- Analysis and assessment techniques that can be used to predict where and how such anomalous behaviors are likely to occur;
- Architectural engineering countermeasures that can be used by the system's developer to either prevent such behaviors or to contain and minimize their impact, thereby mitigating the risk they pose to the safe, secure operation of the system.

1.2. SCOPE

1.2.1. In Scope

This paper's focus is limited to safety and safety-impacting security issues that arise from assembly of software components, including unsafe, non-secure, or otherwise anomalous behaviors that arise within a software component due to its interaction with another component.

This paper discusses technical analysis and testing methods to predict safety hazards and safety-impacting security risks (defined in Section 2) that are likely to arise in software-intensive safety-critical and safety-relevant component-based software-intensive systems.[2] Such hazards and risks emerge from:

- interactions between the software components of the system;
- interactions between the system's software components and any hardware components from which the software components obtain data of some sort (*e.g.*, mechanical sensors).

The paper also discusses software architectural engineering techniques and tools that can be used to mitigate those emergent risks and hazards.

Finally, this paper discusses testing and analysis techniques and tools for assessing the safety and safety-impacting security properties of individual components, pairs of interacting components, and whole component-based systems, and for assessing the effectiveness of controls and countermeasures implemented to mitigate safety hazards and security risks in such systems.

1.2.2. Out of Scope

This paper does not address:

- Techniques for assessing or mitigating hazards and risks emerging from interactions exclusively between hardware components, with no software involved.
- Required or desirable properties in software-intensive systems other than safety and security, such as quality, reliability, performance, and usability.
- System-of-Systems (SoS) issues, except those challenges and considerations that pertain equally to the interactions among components in a single system and the interactions among systems-as-components in a larger SoS.
- Information assurance and cyber security as they pertain to safety-critical software-intensive component-based systems.
- Operational safety and security of component-based systems in deployment/production.
- Software engineering and assurance practices and tools for use in the original development of software components, or general information on how to develop individual software components to be safe and secure when operating in isolation. However, the authors recognize that such information may be of interest, and so a number of useful resources about safe and secure software engineering are identified below.

INFORMATION RESOURCES ON DEVELOPING SAFE SOFTWARE

- DOD, *Joint Software Systems Safety Engineering Handbook*, Draft Version 0.95, 30 September 2009. http://www.acq.osd.mil/atptf/guidance/Draft_Software_Safety_Handbook_2009-11-11.pdf
- DOD, *Software System Safety Handbook*, December 1999. http://www.system-safety.org/Documents/Software_System_Safety_Handbook.pdf

- North Atlantic Treaty Organization (NATO) Conference of National Armaments Directors (CNAD) Ammunition Safety Group, *Guidance on Software Safety Design and Assessment of Munition-Related Computing Systems*, AOP-52 (Edition 1), 9 December 2008. https://acc.dau.mil/GetAttachment.aspx?id=272390&pname=file&aid=42035&lang=en-US

- National Aeronautics and Space Administration (NASA), *NASA Software Safety Guidebook*, NASA-GB-8719.13, 31 March 2004. http://www.hq.nasa.gov/office/codeq/doctree/871913.pdf

- Federal Aviation Administration (FAA), *FAA System Safety Handbook*, 30 December 2000. Specifically Chapter 10: "System Software Safety". Also: Appendix J: "Software Safety". http://www.faa.gov/library/manuals/aviation/risk_management/ss_handbook/media/Chap10_1200.pdf. Also: http://www.faa.gov/library/manuals/aviation/risk_management/ss_handbook/media/app_j_1200.pdf

- Department of Energy (DOE), *Safety Software Guide for Use with 10 CFR 830, Subpart A, Quality Assurance Requirements, and DOE O 414.1C, Quality Assurance,* DOE G 414.1-4. https://www.directives.doe.gov/directives/current-directives/414.1-EGuide-4/at_download/file. Also: Related DOE software quality assurance directives and manuals. https://www.directives.doe.gov/search?Title=%22DOE+o+414.1c%22&Subject:list=status:+current&submit=Search

- Massachusetts Institute of Technology (MIT) Aero/Astro Software Engineering Research Laboratory Papers. http://sunnyday.mit.edu/papers.html

- Leveson, Nancy G., *Safeware: System Safety and Computers* (Reading, Mass.: Addison-Wesley, 1995).

- Herrmann, Debra S., *Software Safety and Reliability* (Los Alamitos, CA: Wiley-IEEE Computer Society Press, 2000).

- Wichmann, Brian A. (editor), *Software in Safety-Related Systems: Special Report* (Chichester, UK: John Wiley and Sons/BCS, 1992).

- Fowler, Kim (editor), *Mission-Critical and Safety-Critical Systems Handbook: Design and Development for Embedded Applications* (Burlington, MA: Newnes Publishing, 2009)

INFORMATION RESOURCES ON DEVELOPING SECURE SOFTWARE

- Goertzel, Karen Mercedes, Theodore Winograd, *et al.*, for the Department of Homeland Security (DHS) and DOD Data and Analysis Center for Software, *Enhancing the Development Life Cycle to Produce Secure Software*, October 2008.

- DHS, Build Security In portal. https://BuildSecurityIn.US-CERT.gov

- Broy, Manfred, Johannes Grunbauer, and Tony Hoare (editors), *Software Systems Reliability and Security* (Amsterdam, The Netherlands: IOS Press, 2007)

- Daswani, Neil, Christoph Kern, and Anita Kesavan, *Foundations of Security: What Every Programmer Needs to Know* (New York, NY: Springer-Verlag, 2007).

1.3. INTENDED AUDIENCE

The primary intended audience for this document includes architects, developers, and engineers responsible for the software elements within safety-critical and/or safety-relevant systems.

The secondary intended audience for this document includes:

- Engineers responsible for the overall architecture (hardware + software) of safety-critical/safety-relevant systems: This document should increase their understanding of the safety and security issues that can arise in component-based system development;

- Developers and other technical advisors to those responsible for acquisition/selection for reuse of the software components that will be used in such systems: This document should increase their understanding and ability to account for the hazards and risks that it may not be possible to mitigate solely at the software architecture level, but which may require adjustments to the hardware and whole system architectures as well;

- Software developers of individual software components: This document should increase their understanding of the issues associated with assembly of such components, and should inspire them to adopt software engineering and assurance and practices to render their components more robust "out of the box", and thus less likely to manifest hazardous or risky behaviors when assembled with other components.

1.4. ASSUMPTIONS

The reader is presumed to be well-versed in general system and software engineering concepts and terminology, and in system safety concepts and terminology.

The reader is also presumed to be somewhat familiar with basic component-based software development concepts and terminology.

---

[2] Such systems include weapons systems, avionic systems, automotive computer systems, some medical devices, control systems for nuclear power plants, air and ground traffic control systems, *etc.*

2. DEFINITIONS

Before proceeding, the reader should familiarize himself/herself with the following terms, as they are used in this document. Note that additional definitions of interest can be found in the Glossary in Appendix A:A.2.

2.1. SYSTEM

MIL-STD-882D defines *system* as "a composite, at any level of complexity, of personnel, procedures, materials, tools, equipment, facilities, and software. The elements of this composite entity are used together in the intended operational or support environment to perform a given task or achieve a specific purpose, support, or mission requirement". Similarly, the Defense Acquisition University defines system as "the combination of two or more hardware, software, and/or data components in such as way that they can interoperate to satisfy one or more functional requirements".[3] While this definition is certainly valid, in the context of component-based software systems, the definition[4] posited by the International Council on Systems Engineering (INCOSE) may be more helpful: "A system is a construct or collection of different elements that together produce results not obtainable by the elements alone. The elements, or parts, can include people, hardware, software, facilities, policies, and documents; that is, all things required to produce systems-level results. The results include system level qualities, properties, characteristics, functions, behavior and performance. The value added by the system as a whole, beyond that contributed independently by the parts, is primarily created by the relationship among the parts; that is, how they are interconnected."

2.2. SOFTWARE-INTENSIVE SYSTEM

According to International Organization for Standards and International Electrotechnical Commission (ISO/IEC),[5] a software-intensive system is any system in which software contributes essential influences to the design, construction, deployment, and evolution of the system as a whole. More simply, a software-intensive system, is any "system that is heavily dependent, or reliant, upon software" for its functionality.[6] A third definition asserts that to be considered software-intensive, a system need not necessarily have software implement the largest part of its functionality as long as software represents largest portion of the system's development costs, development risk, and/or development time.[7]

There is no question that safety-critical and safety-relevant systems are becoming increasingly, if not exclusively, software-intensive. "Systems in general are becoming increasingly software-intensive. Processors are embedded everywhere and networked computing is ubiquitous—in vehicles, buildings, appliances, workplaces, and other entities in which humans participate or with which they interact."[8] Moreover, "software-intensive systems include large-scale heterogeneous systems, embedded systems for automotive applications, telecommunications, wireless ad hoc systems, business applications with an emphasis on web services *etc.* Our daily lives depend on complex software-intensive systems, from banking to communications to transportation to medicine."[9]

Software-intensive systems clearly include weapons systems. "Modern weapon systems increasingly depend on large amounts of real-time, safety-critical, embedded software to achieve their mission objectives."[10] It is, in fact, Navy policy to consider any system to be software-intensive, and to be governed by acquisition and engineering policy that applies to software-intensive systems, unless a compelling case can be made for why it is not.[11]

2.3. COMPONENT

A component is a stand-alone piece of software that includes enough functionality to be useful on its own, *i.e.*, without being assembled into a larger system, but can also be used as a building block to create larger, more complex software systems. The feature of a component that enables it to be assembled with other components into a system is its built-in interface(s). Component interfaces are usually based on standards (*de facto* or *de jure*). [12] Indeed, along with autonomy, built-in interface(s) is one of the features that distinguishes components from other software entities (*e.g.*, compilation units).[13] A component may be a software module, *i.e.*, the smallest *autonomous* unit of decomposition in a software-based system, or it may be a larger entity which is itself assembled from smaller components. The difference between a module and a component is that a module need not be self-contained, while a component must.

The following is a summary of the required features and properties of components:

- **Component service specification:**[14] Interactions between components can be characterized in terms of provision by one component and consumption by another of a service.[15] [16] The service (function or computation) one component provides to another, the services it requests from the other component, and the details of the interface(s) by which these provisions and requests are made can be specified as a *contract* between the consumer component and the provider component (see below).[17] The assumptions a provider component has about the contractual obligations they are expected to fulfill in order to consume its service are also explicitly stated in the service specification, as are the preconditions or dependencies that it expects those component to impose on it.

  The service specification includes the *usage definition*, or "interaction scheme", that describes how the service-providing component is to be located, how its control flows are to be synchronized with that of the consumer component, which communications protocol(s) it will use to communicate with the consumer component, how the data it generates as output and receives as input are to be encoded, how different "qualities of service" (*e.g.*, security, safety, transaction control, *etc.*) are to be achieved, *etc.* This usage description should be sufficiently detailed that the developer of another component can implement that component with the ability to successfully consume the service provided by the first component.

- **Precisely defined, contractually specified interfaces:**[18] [19] [20] By "contractually-specified", we mean that the interfaces satisfy certain obligations, which can be characterized as a *contract* between the service-providing component and the service-consuming component. These obligations ensure that independently-developed components recognize and obey the same set of rules that govern how they predictably interact within standard build-time and run-time environments.

The interface used between two components can be thought of as defining the set of intercomponent dependencies that flow from the service-providing component to the consuming component, *i.e.*, what the service-provider expects/requires from the consumer in order to provide its service to that consumer. The components' shared interface also provides the mechanism by which each component can communicate its assumptions to the other, and can determine whether there is a match or mismatch between its assumptions and those of the other component.[21] Such mismatches can inhibit or prevent the successful assembly of two components because components can interact successfully only when they share consistent assumptions about what each will provide to or require from the other.

While some assumptions necessarily pertain to the unique service (function or computation) each component brings to the interaction, most of the assumptions can be standardized across all components, and can thus be expressed as a standard component interface that can be replicated across multiple components. Components are often categorized according to the standard interface(s) they implement, *e.g.*, Web service components, Java Enterprise Edition (EE)/ Java Bean components, Eclipse components, .NET components, Distributed Component Object Model (DCOM) components, Common Object Request Broker Architecture (CORBA) components..

- **Independence of any dependencies on a fixed set of clients, services, or resources:**[22] The component can provide its service to any client(s) that provides the requisite interface and satisfies the specified usage assumptions of the first component. While co-dependencies are found between pairs of modules in all types of systems—*e.g.*, client/server, peer-to-peer—in component-based systems the detailed specification of standard interfaces enables a key benefit of component-based software engineering: the ability of components to be easily "swapped out" or substituted/replaced (at the architectural level) by other components with comparable functionality and the same standard interfaces.[23] In terms of software development life cycle, any given component can be developed and delivered independently of the release of all other components in the system.[24]

- **Encapsulation and abstraction:**[25] Because components are addressable only through their explicit interfaces, they can be seen as (1) hiding the details of their internal implementations; in practical implementation, this form of information hiding is known as *encapsulation*; (2) *abstracting* the component by limiting the information revealed about it to what is conveyed in the component's interfaces. Both encapsulation and abstraction simplify the component's external representation, limiting it to only the information a developer needs to produce another component that can interact successfully with the first. In essence, then, components can be seen as "black boxes".[26]

Component developers often build their components in accordance with a predefined *component model*.[27] This model defines all of the standard interfaces the component will need to function, and their associated contractual obligations. By building the component to this model, developers can work in isolation yet be fairly certain the components they produce will be able to interact with other components built according to the same model. The component model, then, provides a kind of standard template that specify the standards and conventions under which components

are to be designed and implemented, thereby reducing the chance of assumption mismatches between components built to the model. Compliance with a standard component model is one of the features that distinguishes a component from other types of packaged software.

Interactions among pairs of components can be viewed in terms of provision and consumption of services (that is, with one component acting as "server" and the other as "client", with the possibility of the components interchanging roles as needed)[28] in accordance with the specified interfaces and obligations defined for each component.[29]

2.4. COMPONENT-BASED SYSTEM

A component-based system is a system in which components are integrated together using their built-in interfaces to collectively provide a specified set of system services or functions. Component-based systems share several characteristics with SoS:[30]

1. **Operational Independence of the Elements:** The SoS is composed of systems which are independent and useful in their own right. If the SoS were to be disassembled into its component systems, those components would be able to usefully operate independently. While individual components in a component-based system may not provide as extensive a set of functions as systems within a SoS, like those systems, components can be independently functional, even if that functionality is limited and may not be particularly useful on its own.

2. **Evolutionary Development:** The SoS does not appear fully formed. Its development and existence is evolutionary with functions and purposes added, removed, and modified based on lessons learned during its development and use. Component-based systems share this characteristic to some extent, because the vast majority of such systems are built, at least in part, from non-developmental components, each of which has its own development life cycle and release schedule. Moreover, an innate characteristic of a component is its ability to be replaced at any time by a different functionally comparable component. These aspects of components render most component-based system developments at least somewhat evolutionary.

3. **Emergent Behavior:** The system performs functions and carries out purposes that do not reside in any single component of the SoS. These behaviors are emergent properties of the system-of-systems. The principal purposes of the systems-of-systems are fulfilled by these behaviors. It is because these emergent behaviors may have safety or security implications or impacts that their behaviors in the assemble system need to be predicted, assessed, and if necessary constrained to mitigate risk to the greatest extent possible.

A significant difference between a component-based system and a SoS is that the presence of compatible built-in interfaces cannot be presumed in the systems from which a SoS is composed, while such interfaces can be presumed in components that have been successfully assembled into a component-based system. In an SoS, various gateways, frontends, and other workarounds are often necessary to enable two independently developed systems to interoperate to the extent required by the SoS requirements.

ADDITIONAL SUGGESTED RESOURCES

- Kozaczynski, Wojtek, and Grady Booch, "Component-Based Software Engineering", in IEEE *Software*, Volume 15 Issue 5, October 1998, pages 37-46. http://www.idi.ntnu.no/emner/dt8100/extra-papers/P-r10-kozaczynski98.pdf (accessed 8 April 2010).
- Bachmann, Felix, Len Bass, Robert Seacord, et al., Carnegie Mellon University Software Engineering Institute (CMU SEI), *Volume II: Technical Concepts of Component-Based Software Engineering,* 2nd Edition, Technical Report CMU/SEI-2000-TR-008 / ESC-TR-2000-007, May 2000. http://www.sei.cmu.edu/reports/00tr008.pdf (accessed 14 January 2010).
- Cai, Xia, Michael R. Lyu, and Kam-Fai Wong, Chinese University of Hong Kong, and Roy Ko, Hong Kong Productivity Council, "Component-Based Software Engineering: Technologies, Development Frameworks, and Quality Assurance Schemes", in *Proceedings of the Seventh Asia-Pacific Software Engineering Conference*, Singapore, 5-8 December 2000. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.8.6911&rep=rep1&type=pdf (accessed 14 January 2010).
- Kaur, Iqbaldeep, Parvinder S. Sandhu, and Vandana Saini, Rayat and Bahra Institute of Engineering and Bio-Technology, and Hardeep Singh, Guru Nnak Dev University, "Analytical Study of Component Based Software Engineering", in *Proceedings of the World Congress on Science, Engineering and Technology*, Penang, Malaysia, 25-27 February 2009. http://www.waset.org/journals/waset/v50/v50-78.pdf (accessed 14 January 2010).
- Crnkovic, Ivica, and Magnus Larsson (editors), *Building reliable component-based software systems*; see specifically Chapter 13: "Components in real-time systems", and Chapter 16: "Component-based embedded systems" (Norwood, MA: Artech House, 2002).

2.5. SAFETY

MIL-STD-882D defines *safety* as: "Freedom from those conditions that can cause death, injury, occupational illness, damage to or loss of equipment or property, or damage to the environment". Such a "condition" is termed a *hazard*, which is defined "Any real or potential condition that can cause injury, illness, or death to personnel; damage to or loss of a system, equipment or property; or damage to the environment." A hazard, if realized, results in a *mishap*, which is defined as "An unplanned event or series of events resulting in death, injury, occupational illness, damage to or loss of equipment or property, or damage to the environment."

MIL-STD-882D defines *system safety* as "the application of engineering and management principles, criteria and techniques to achieve acceptable mishap risk within the constraints of operational effectiveness, time and cost throughout the system's life cycle." The NAVSEA *Weapon System Safety Guidelines Handbook* defines[31] *software safety* as "the process that is

applied to ensure that the software will execute in the system context and operational environment with an acceptable level of safety mishap risk."

As Desmond Meacham has observed,[32] these definitions are based on the assumption that system and software safety rely on the processes by which the system and software are developed; that is, that safety must be part of the design of the system. System safety engineering focuses on early identification and analysis of hazards at the system level and root causes of those hazards at the software level. These analyses enable the system developer to mitigate those hazards and root causes through the system design. This is no less true when the design process includes the specification of a component-based architectural design.

Nancy Leveson has acknowledged that absolute "freedom from those conditions that can cause death, injury, occupational illness, damage to or loss of equipment or property, or damage to the environment" is not achievable for complex real-world systems, although absolute safety should always be the goal starting point from which judgments about acceptable levels of mishap risk are made. As a system property, safety has a contribution from software in software-intensive systems. While software itself is an abstraction, and thus has no substance and cannot directly harm people, property, or the environment—it is the nature of the software's sensing and control of physical components within a system and its environment that render that system safe or not. Safety of software, then, depends on the combined interactions of the software with hardware, system operator, and external factors. Leveson's definition of software system safety, while consistent with the NAVSEA definition of software safety, is more explicit about this dependency: "Software System Safety implies that the software will execute within a system context without contributing to hazards".[33]

2.6. SECURITY

Enhancing the Development Life Cycle to Produce Secure Software[34] defines security as it pertains to software and software-based systems as follows:

> *"To be considered secure, software must exhibit three properties: (1)* Dependability: *Dependable software executes predictably and operates correctly under all conditions, including hostile conditions, including when the software comes under attack or runs on a malicious host;*[35] *(2)* Trustworthiness: *Trustworthy software contains few if any vulnerabilities or weaknesses that can be intentionally exploited to subvert or sabotage the software's dependability. In addition, to be considered trustworthy, the software must contain no malicious logic that causes it to behave in a malicious manner; (3)* Survivability *(also referred to as "Resilience"): Survivable—or resilient—software is software that is resilient enough to (1) either resist (i.e., protect itself against) or tolerate (i.e., continue operating dependably in spite of) most known attacks plus as many novel attacks as possible, and (2) recover as quickly as possible, and with as little damage as possible, from those attacks that it can neither resist nor tolerate."*

The security counterpart to a safety hazard is termed a *risk*; the causal agent of a risk is termed a *threat*, and the mode through which that agent operates is termed an *attack*.

What is targeted by an attack is a *weakness* which, if it can be intentionally used—*exploited*—by the attacker to achieve his objectives, is termed a *vulnerability*; as its name implies, an attack that has the express purpose of exploiting a vulnerability is termed an *exploit*.

The outcome of a successful attack, which is the security counterpart of a safety mishap, is referred to as a *compromise*. There are several types of possible compromise, some of which can exist simultaneously:

- *subversion* in which the system's functionality is altered or co-opted by the attacker to achieve certain objective(s);
- *sabotage*, usually referred to as *denial of service*, in which intentional action(s) is undertaken to interrupt or disrupt either the system's ability to operate dependably, or the ability of its users to access its functions;
- *intrusion* or *penetration* (implying forced entry) in which the attacker gains access to the system's functions, resources, or data that would otherwise be denied to him/her;
- *violation*, in which a security policy predefined for and enforced by the system is broken, or a security control within the system is breached or bypassed.

2.6.1. Safety-Impacting Security Risk

Safety-impacting security risks are those security risks that, if realized, introduce safety hazards to a system. Specifically, safety-impacting security risks are risks to the *integrity* and *availability* properties of the system.

Integrity is the security property closely linked to dependability and trustworthiness: the system's dependability relies on it remaining intact, and not being altered unexpectedly, either accidentally or intentionally or by an unauthorized party or in an unauthorized way. The system's trustworthiness relies on it not being altered in ways that introduce vulnerabilities or malicious logic. The reason integrity is important, then, is that unauthorized and unexpected modifications can not only impede the system's ability to operate dependably, but can render the system vulnerable to additional compromises, and/or can add unauthorized functions. In all cases, such alterations also renders incorrect all assumptions under which the system operates—assumptions based on painstaking analysis and testing of the system prior to its deployment. Assurance of system integrity, then, is assurance of system intactness and the correctness of assumptions about the system.

Availability is another security property closely linked to system dependability. To be considered available, the system must be operational and accessible by its intended users to the extent specified in its requirements (*e.g.*, 99% of the time, 95% of the time, *etc.*). Availability is similar to "required uptime" and "quality of service", except that it addresses not only the system's continuity of operation but also the continuity of access to the system by those that require it. The main threat to availability is denial of service.

It is impossible for the safety of a system to be assured unless these two security properties can be assured against intentional compromise. This is why the system must be able to avoid, resist, or withstand threats to its integrity and availability, and errors and weaknesses that make the system vulnerable to such threats must be eliminated or mitigated sufficiently, as part of the overall safety engineering of the system.

ADDITIONAL SUGGESTED RESOURCES

- Nordland, Odd, "Some Security Aspects in Safety Related Systems", presented at the Safety-Critical Systems Club/European Workshop on Industrial Computer Systems Reliability Safety and Security Security of Safety-Critical Computer Systems Workshop on the Relationship between Safety and Security in Software-Based Systems, Newcastle, UK, 25 September 2008.
- Rushby, John, SRI International, *Critical System Properties: Survey and Taxonomy*, SRI Technical Report CSL-93-01, May 1993 / revised February 1994. http://www.csl.sri.com/papers/csl-93-1/ (accessed 8 April 2010). Also: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.72.6265&rep=rep1&type=pdf (accessed 8 April 2010).

---

[3] Defense Acquisition University (DAU), *Glossary of Defense Acquisition Acronyms and Terms,* 13th Edition, November 2009. https://acc.dau.mil/GetAttachment.aspx?id=17650&pname=file&aid=48053 (accessed 27 March 2010). Also: https://acc.dau.mil/CommunityBrowser.aspx?id=17650 (accessed 31 March 2010).

[4] Originally articulated by Eberhardt Rechtin and Mark W. Maier in *The Art of Systems Architecting,* Second Edition (Boca Raton, FL: CRC Press LLC, 2000), page 6.

[5] In ISO/IEC 42010:2007, *Systems and Software Engineering—Recommended practice for architectural description of software-intensive systems*.

[6] Goldenson, Dennis R., and Matthew J. Fisher, Carnegie Mellon University Software Engineering Institute, *Improving the Acquisition of Software Intensive Systems*, CMU/SEI-2000-TR-003—ESC-TR-2000-003, August 2000. http://www.sei.cmu.edu/library/abstracts/reports/00tr003.cfm (accessed 24 March 2010).

[7] Op. cit., DAU *Glossary*.

[8] Goldin, Dina, and David Keil, University of Connecticut, "Interactive Models for Design of Software-Intensive Systems", in *Proceedings of the Workshop on the Foundations of Interactive Computation*, Edinburgh, Scotland, 9 April 2005. http://www.engr.uconn.edu/~dqg/papers/sod.pdf (accessed 24 March 2010).

[9] Wirsing, Martin, and Rémi Ronchaud, "Report on the EU/NSF [European Commission/National Science Foundation] Strategic Workshop on Engineering Software-Intensive Systems, Edinburgh, Scotland, 22-23 May 2004". http://www.ercim.eu/EU-NSF/sis.pdf (accessed 24 March 2010).

[10] Feiler, Peter H., and Dionisio de Niz, CMU SEI, *ASSIP Study of Real-Time Safety-Critical Embedded Software-Intensive System Engineering Practices*, CMU/SEI-2008-SR-001, February 2008. http://www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&doc=GetTRDoc.pdf&AD=ADA480129 (accessed 24 March 2010).

[11] "[W]ith the vast majority of system functionality now being implemented via software (vice hardware) in our Naval weapons and information systems, this focus team believes that all current systems should be considered to be

'software intensive' unless the Program/Project Manager can explain why they are not. Therefore no definition of a 'software intensive system' need be provided." Assistant Secretary of the Navy for Research, Development, and Acquisition (ASN[RD&A]) Software Acquisition Management Focus Team, *Software Acquisition Management "As-ls State" Report*, Version 2.0, 17 April 2007. https://acquisition.navy.mil/rda/content/download/4631/20848/version/4/file/AS+Is+REPORT+011.pdf (accessed 24 March 2010).

[12] De Panfilis, Stefano, SINTEF, and Arne J. Berre, Engineering Ingegneria Informatica S.p.A., Open issues and concerns in Component Based Software Engineering. http://research.microsoft.com/~cszypers/events/wcop2004/20%20Panfilis%20Berre.pdf (accessed 29 March 2010). Also

[13] Hepner, M., R. Gamble, M. Kelkar, L. Davis, and D. Flagg, University of Tulsa, "Patterns of Conflict among Software Components", in *The Journal of Systems and Software*, Volume 79 Issue 4, April 2006, pages 537-551. http://www.seat.utulsa.edu/papers/JSS05-Hepner.pdf (accessed 31 March 2010).

[14] Lau, Kung-Kiu, Vladyslav Ukis, and Zheng Wang, University of Manchester, "Predictable Assembly: Towards a Definition", presented at the Second Workshop on Predictable Software Component Assembly, Manchester, UK, 12 September 2005. http://www.cs.man.ac.uk/~wangz0/program.html (accessed 7 March 2010).

[15] Broy, Manfred, Technical University, Anton Deimel, SAP AG, Juergen Henn, IBM, Kai Koskimies, Nokia Research, František Plášil, Charles University, Gustav Pomberger, University of Linz, Wolfgang Pree, University of Constance, Michael Stal, Siemens AG, and Clemens Szyperski, Queensland University of Technology, "What characterizes a (software) component?", in *Software—Concepts and Tools*, Volume 19 Number 1, June 1998, pages 49-56. http://www.exciton.cs.rice.edu/comp410/frameworks/Pree/J010.pdf (accessed 12 March 2010).

[16] Bachmann, Felix, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau, Carnegie Mellon University Software Engineering Institute, *Volume II: Technical Concepts of Component-Based Software Engineering*, 2nd Edition, Technical Report CMU/SEI-2000-TR-008 | ESC-TR-2000-007, May 2000. http://www.sei.cmu.edu/reports/00tr008.pdf (accessed 7 March 2010).

[17] Han, Jun, Ryszard Kowalczyk, Swinburne University of Technology (Melbourne, Australia) and Khaled M. Khan, Qatar University, "Security-Oriented Service Composition and Evolution", in *Proceedings of the 13th Asia Pacific Software Engineering Conference*, Bangalore, India, 6-8 December 2006. http://www.ict.swin.edu.au/personal/jhan/jhanPapers/apsec06sec-comp.pdf (accessed 14 January 2010).

[18] *Ibid.*

[19] *Op. cit.* Bachmann, Bass, *et al.*, *Technical Concepts*.

[20] *Op. cit.*, Broy, Deimel, *et al.*, "What characterizes".

[21] Khan, Khaled M., University of Western Sydney, and Jun Han, Swinburne University of Technology, "A Process Framework for Characterising Security Properties of Component-Based Software Systems", in *Proceedings of the 2004 Australian Software Engineering Conference*, Melbourne, Australia, 13-16 April 2004. http://www.ict.swin.edu.au/personal/jhan/jhanPapers/aswec04.pdf (accessed 29 March 2010).

[22] *Op. cit.*, Lau, Ukis, *et al.*, "Predictable Assembly".

[23] In fact, as noted by Broy, Deimel, *et al.*, it is the component architectural framework that provides the enabling technology of "plug and play" software whereby most adaptations to the component-based system can be achieved by simply swapping components in and out.

[24] *Op. cit.*, Broy, Deimel, *et al.*, "What characterizes".

[25] Meyer, Bertrand, ETH Zürich, "The Grand Challenge of Trusted Components", in *Proceedings of the 25th International Conference on Software Engineering*, Portland, Oregon, 3-10 May 2003. http://se.ethz.ch/~meyer/publications/ieee/trusted-icse.pdf (accessed 31 May 2010).

[26] Broy, Manfred, Technische Universität München, "Towards a Mathematical Concept of a Component and Its Use", keynote speech at the 1996 Components Users' Conference, Munich, Germany, 15-19 July 1996. http://www4.informatik.tu-muenchen.de/papers/Broy_CUC1996_klein_1996_Publication.html (accessed 12 March 2010).

[27] *Op. cit.*, Bachmann, Bass, *et al.*, *Technical Concepts*.

[28] *Op. cit.*, Broy, Manfred, Anton Deimel, *et al.*, "What characterizes a (software) component?".

[29] *Op. cit.*, Bachmann, Bass, *et al.*, *Technical Concepts*.

[30] Maier, Mark W., The Aerospace Corporation, "Architecting Principles for Systems-of-Systems", in *Systems Engineering*, Volume 1 Issue 4, 1988, pages 267-284. http://www.infoed.com/Open/PAPERS/systems.htm (accessed 31 March 2010).

[31] NAVSEA, *Weapon System Safety Guidelines Handbook*, Section II Chapter 14 "Software Safety", NAVSEA SW020-AH-SAF-010, 2005. http://www.lejeune.usmc.mil/eso/orders/sw020-af-hbk-010.pdf (accessed 14 January 2010).

[32] Meacham, Desmond J., Flight Lieutenant, Royal Australian Air Force, *Standards Interoperability: Application of Contemporary Software Safety Assurance Standards to the Evolution of Legacy Software,* Naval Postgraduate School Master of Science Thesis, March 2006. http://www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&doc=GetTRDoc.pdf&AD=ADA445887 (accessed 14 January 2010).

[33] Leveson, Nancy G., *Safeware: System Safety and Computers* (Boston, MA: Addison-Wesley, 1995).

[34] Goertzel, Karen Mercedes, Ted Winograd, *et al.* for the Department of Homeland Security, *Enhancing the Development Life Cycle to Produce Secure Software,* Version 2.0, published for DHS by the DOD Data and Analysis Center for Software (DACS), October 2008. https://www.thedacs.com/techs/enhanced_life_cycles/ (accessed 31 March 2010; download requires free registration on DACS site). Also available online: https://wiki.thedacs.com/Enhancing_the_Development_Life_Cycle_to_Produce_Secure_Software (accessed 31 March 2010).

[35] This document does not discuss the characteristics, requirements, or techniques for achieving dependability. Instead, we recommend the reader consult an excellent in-depth discussion of software dependability issues, requirements, and techniques, in Johns, Lionel S., Peter Sharfman, Thomas H. Karas, Anthony Fainberg, C.E. "Sandy" Thomas, and David Weiss, U.S. Congress, Office of Technology Assessment, *SDI: Technology, Survivability, and Software*, OTA-ISC-353 (Washington, DC: U.S. Government Printing Office, May 1988). http://govinfo.library.unt.edu/ota/Ota_3/DATA/1988/8837.PDF (accessed 14 January 2010). Also: http://handle.dtic.mil/100.2/ADA339517 (accessed 14 January 2010).

3. COMPONENT-BASED SYSTEM DEVELOPMENT

> *While COTS components have been widely expected to provide significant cost savings, this has not always proved to be the case. Hidden costs include market research that must be conducted to find suitable components, extensive component evaluation and testing, licensing problems, and costs due to continual changes and upgrading of commercial components…. On the operating system level [as an example], fundamental issues must be addressed at the evaluation stage of system development, including access to source code, choice of a real-time operating system versus non-real-time operating system, and operating system functionality. Once an operating system is chosen, it may even be necessary to tailor it or otherwise protect against unwanted extra functionality. Development of a fault-tolerant architecture could provide an extra level of reliability and safety for safety-critical applications.*[36]

Assembly of components into composed systems can be greatly aided by use of an architectural component *framework*. A component framework is an implementation of services that support or enforce a component model. The component model will also define how components are to be bound to the resources, or services, provided by the component framework other by other components in that framework. A component framework can be seen as a kind of "mini-operating system" in which components are assembled together, with the framework's set of services in some cases augmenting those provided by the components within the framework, and in other cases facilitating the consumer-provider interactions between components.[37] The integration of components within a framework is commonly referred to as *composition* or *assembly*. The component model used to develop the component may specify how components are to interact directly with each other, how they are to interact within a component framework, or both.

Because properties such as safety and security are not intrinsic to individual components in isolation—these properties *emerge* from the interactions between components or the interactions between a component and its environment or a human user—individual component testing and analysis (*e.g.*, through static and dynamic analysis, fault injection, fuzzing, *etc.*) can only provide incomplete and indirect evidence of how the component might behave when interoperating with other components within a component-based system. Emergent properties can only be demonstrated through testing that involves component interactions, *e.g.*, pair-wise component testing and testing of whole component assemblies.

Received wisdom suggests that a particular safety or security property will hold true in a composed system as long as its individual components satisfy that property.[38] Because of this view, it is also commonly believed that the properties of the composed system are necessarily derived from the safety and security properties of its least safe and least secure components. In reality, it is not necessary for the least robust components to dictate the safety and security properties of the composed system as a whole. Component assembly techniques, such as virtual machine (VM) "sandboxing" of unsafe/non-secure components during system operation, can be used to isolate less trustworthy components and prevent their flaws from impacting the safe and secure operation of the system overall.

For example, if one component suffers from a buffer overrun while a second maintains safe array bounds, the flawed component can be isolated and contained through use of fine-grained, intricate component assembly techniques. One such technique is the implementation of a small module that associates the system's sandboxing techniques with the individual interfaces between interacting components—these interactions being manifested by the nature of those components' inputs and outputs. The enhanced sandboxing capability would be used to isolate the execution of the components whose interfaces have sandboxing associated with them.

Further, use of autonomic and biologically-inspired survivability techniques (see Appendix C) can help ensure that the converse of "common wisdom" is true, *i.e.*, that the system as a whole is not only more robust than its weakest component, but is even stronger than its strongest component.[39]

Trustworthy and untrustworthy components are arranged, and controls and countermeasures to safety hazards and security risks posed by the untrustworthy components are added, so that the components can all interoperate with confidence that the system as a whole can operate safely and securely.

> *NOTE: This document presumes a close dependency between security of a component or system and the ability of that component or system to be deemed safe.*[40] *"Security" in this context refers to the behavior of the software itself, not its ability to protect information. To be secure, software must be (1) dependable—correct and predictable in its operation, (2) trustworthy—non-malicious, and not vulnerable to subversion or sabotage of its operation, and(3) survivable—able to tolerate intentional attempts at subversion/sabotage as well as unintentional faults. Non-secure software is inherently unpredictable in its behavior due to either (1) the inclusion of malicious logic which, if executed, will cause the software to perform unexpected, destructive function(s), or (2) the presence of vulnerabilities that, if exploited, will cause the software to behave in ways it is not expected to. In either case, this unpredictability renders the component unsafe.*

There are standard and commodity frameworks for software component models such as Eclipse, Java EE, and .NET, as well as earlier framework technologies such as CORBA and DCOM.[41] These frameworks, however, are all limited to the specification and matching of structural interface definitions. Interface description languages (IDLs) enable the expression of the syntactic structure of each interface, including its attributes, operations, and events. To publish the safety and security properties of a component, the component's interface description would have to be extended beyond the operations and attributes that constitute its functionalities to also express the constraints that must be enforced for those operations and attributes, and their associated functionalities, to manifest safely and securely.

ADDITIONAL SUGGESTED RESOURCES

- Ellison, Robert J., Carnegie Mellon University Software Engineering Institute, "Trustworthy Composition: The System Is Not Always the Sum of Its Parts", on DHS Build Security In portal and dated 28 September 2005. https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/assembly/50-BSI.html (accessed 8 April 2010).

- Caffall, S. "Butch", *Conceptual Framework Approach for Systems-of-Systems Software Developments,* Naval Postgraduate School Master of Science Thesis, March 2003. http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA418550&Location=U2&doc=GetTRDoc.pdf (accessed 14 January 2010).
- Xie, Fei, and James C. Browne, "Verified systems by composition from verified components", in *Proceedings of Fourth Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering,* Helsinki, Finland, 1-5 September 2003. http://web.cecs.pdx.edu/~xie/pubs/composition-tr.pdf (accessed 14 January 2010).
- Fairbanks, George, *Design Fragments*, Ph.D. thesis for Carnegie Mellon University, CMU-ISRI-07-108, 19 April 2007. http://reports-archive.adm.cs.cmu.edu/anon/isri2007/CMU-ISRI-07-108.pdf (accessed 8 April 2010).

3.1. COMPONENT EVALUATION AND SELECTION

> *If reused software components do not already implement the requirements resulting from the safety contract, it is unlikely that the safety team will be able to affect the design sufficiently to adequately mitigate the associated risk. However, it is possible to specify certain attributes of components... assuming that there are multiple components available from which to select. The safety team can specify those attributes required to maintain an acceptable level of risk in the interaction of a component or system in the new system. If there are available components or packages that have those attributes and also achieve the attributes required by the other "ilities," the designers can incorporate those into the resultant system. However, it is difficult to anticipate what attributes a system may require when the system is undefined. This occurs during the development of a library of reusable components when no system is under development.*[42]

The following considerations should inform the identification and evaluation of components to be used in the assembled system.[43] Given the need to choose between two or more interchangeable components, these considerations should also inform the determination of which is best suited for use in the system:

1. Identifying the functional, safety, and security design constraints that should be used to narrow the range of possible components to be evaluated;
2. Selecting a methodology by which the design tradeoffs can be quantified so that a measure of a given component's fitness for safe, secure use can be determined;
3. Determining, through various analyses and tests, which components best satisfy the requirements of their intended role in the composed system;

4. Determining, through various analyses and tests, which components exhibit the most consistently dependable, trustworthy behavior when combined with other components;

5. Selecting the components to be used in the application.

Before component assessment can begin, a set of criteria against which the components are to be assessed needs to be established.[44] These criteria may include product criteria, for assessing the components themselves, as well as process criteria, for assessing the processes used to develop the components (when such information is available). The product assessment criteria should include the following considerations:

- **Product acceptance:** Known safety and security properties and issues may be revealed by pre-existing evidence available about other organizations' experiences using the component in similar/comparable systems, in published product evaluations, in safety or security incident reports involving the product, in vulnerability and malicious code reports for the product, *etc.*

- **Constraints and invariants:** To assess the safety and security impact of any preconditions, post-conditions, and other constraints and invariants under which the component is required to operate, *e.g.*, imposed by performance requirements, safety and security requirements, execution environment specifications, *etc.*

- **Behavior:** Examples of component behavior can be derived from design documentation, test results, documented past experiences using the product, and direct observation. A behavior assessment should consider all known and assumed constraints and invariants, to understand the conditions under which the product is expected to operate (in the case of design document review), and the conditions under which it was operating when the behaviors were observed (in the case of test results and past experience reviews).

- **Design:** Evaluation of documented dependencies, interface specifications, test cases, proofs of safety/security properties (if any), proofs of preconditions, post-conditions, invariants

- **Extensibility:** Built-in mechanisms available for adding logic, redefining/modifying logic.

The process assessment criteria should include the following considerations:

- **Methodology:** Was a known good methodology used to develop the component, *e.g.*, Rational Unified Process, Trusted Software Process (Carnegie Mellon University Software Engineering Institute), Security Development Lifecycle (Microsoft)? In some cases, this information can be discovered by simply asking the suppliers. In other cases, such as Microsoft's use of its Security Development Lifecycle methodology, the company has published extensively on the content of the methodology, and on which Microsoft products are developed using that method, and which are not.

- **Process Standard Certification:** Is the development organization certified against a relevant process quality, safety, or security standard, *e.g.*, CMMI, SSE-CMM, ISO 9003?

- **Reputation:** Is the development organization recognized as a trustworthy supplier of safety-critical and/or security-critical software?

- **Incident Response:** Has the development organization previously released software known to have been directly involved in a significant safety mishap or security incidents? What measures were taken by the development organization to prevent a recurrence?

Evaluation of an non-developmental component's safety and security properties is best accomplished in stages: first, an initial assessment to disqualify obviously-unsuitable components, followed by in-depth hazard, threat, and risk assessments of the "down-selected" shortlist of components. Pre-selection evaluation of candidate components should not only confirm that the component does what it will be required to do when assembled into the system, but must also confirm that the component will not introduce potential hazards or vulnerabilities to the system.

As much pre-existing evidence as possible should be gathered in support of the initial assessment of component fitness-for-use in the intended system.[45] Such evidence should include at least one, and ideally more than one, of the following:

- Results of direct tests, including safety and security tests, of the component;

- Results of analyses of the component and its development process;

- Documented experience with the component in other similar/comparable systems;

- "Trusted third party" evidence, such as independent quality, safety, or security certifications of the component and its development process. Examples: results of IEC 61508 assessments, Common Criteria evaluations reports, *etc.*

It is likely that the initial component assessment can be based predominantly, if not exclusively, on a safety/security evidence review, with direct analysis and testing limited to those components that appear to be the best candidates for fulfilling certain roles in the system. However, when there is insufficient or inadequate pre-existing evidence available for even an initial fitness assessment of the component, it will be necessary to produce additional evidence through testing and analysis. Techniques for direct component analysis and testing are described in Section 7. Note that the more critical the safety and security imperatives of the system, the more likely additional assessments and tests will be needed. Also, such tests greatly increase the cost of using NDI components; such costs may in fact exceed the cost of custom-developing components that are designed from the outset to comply with all relevant engineering, safety, and security requirements.[46]

The results of all analyses of pre-existing evidence as well as any additional direct analyses or tests, must be evaluated to determine whether evidence gathered provides a sufficient basis on which to determine the component's fitness for use, and to make the necessary risk management

decisions regarding additional controls and countermeasures that may be needed to mitigate any non-resolvable safety or security issues presented by use of the component.

ADDITIONAL SUGGESTED RESOURCES

- Aas, Andreas, Norwegian University of Science and Technology, "Use of COTS Software in Safety-Critical Systems", 2007.http://www.idi.ntnu.no/emner/dt8100/Essay2007/aas-essay07.pdf (accessed 8 April 2010).

- Dowling, Ted, UK Defence Evaluation and Research Agency (DERA), "Software COTS Components—Problems, and Solutions?", in *Proceedings of the NATO Research and Technology Organisation Systems Concepts and Integration (SCI) Panel Symposium on Strategies to Mitigate Obsolescence in Defense Systems Using Commercial Components,* Budapest, Hungary, 23-25 October 2000 (RTO-MP-072 AC/323(SCI-084)TP/31).

- Ellis, Richard, Stratum Management Ltd., "Management Issues in the Use of Commercial Components in Military Systems", presented at the NATO Research and Technology Organisation Systems Concepts and Integration Symposium on "Strategies to Mitigate Obsolescence in Defense Systems Using Commercial Components, Budapest, Hungary, 23-25 October 2000. http://handle.dtic.mil/100.2/ADP010981 (accessed 8 April 2010).

- Khan, Khaled M., University of Western Sydney (Sydney, Australia), and Jun Han, Swinburne University of Techology (Melbourne, Australia), "Assessing Security Properties of Software Components: A Software Engineer's Perspective", in *Proceedings of the 2006 Australian Software Engineering Conference*, Sydney, Australia, 18-21 April 2006. http://www.ict.swin.edu.au/personal/jhan/jhanPapers/aswec06-security.pdf (accessed 29 March 2010).

- Leveson, Nancy, and Kathryn Anne Weiss, "Making Embedded Software Reuse Practical and Safe", in *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, Newport Beach, CA, November 2004. http://sunnyday.mit.edu/papers/fse04.pdf (accessed 31 March 2010).

- Minkiewicz, Arlene F., "Security in a COTS-Based Software System", in *CrossTalk: The Journal of Defense Software Engineering*, November 2005. http://www.stsc.hill.af.mil/crosstalk/2005/11/0511Minkiewicz.html (accessed 8 April 2010).

- Schuster, Jens Uwe, infoteam Software GmbH, "COTS-Software in Safety-Related Systems", in *Proceedings of the 8th International Symposium on Programmable Electronic Systems in Safety-Related Applications*, Köln (Cologne), Germany, 2-3 September 2008.

- Ye, Fan, and Tim Kelly, "Contract-Based Justification for COTS Component within Safety-Critical Applications", in *Proceedings of the 9th Australian Workshop on*

Safety Related Programmable Systems, Brisbane, Queensland, Australia, August 2004.

3.2. PREPARING COMPONENTS FOR ASSEMBLY

Untrustworthy components may need to be prepared for assembly into a composed application through reconfiguration, modification, or wrapping. To determine whether such modifications are needed, the engineer needs to determine which component features must be available, which features must be disabled or blocked, and which components need to be prevented from interfacing with one or more other components, and how that prevention can occur (*e.g.*, through VM or trusted processor module [TPM] isolation of either or both of the components, blocking of outputs from or inputs to one or both, *etc.*).

3.2.1. Custom-Developed Elements for Component-Based Systems

As noted above, component-based system development shifts the developer's efforts away from custom coding of software components to custom coding of controls and countermeasures that will mitigate the hazards and risks posed by less trustworthy and untrustworthy COTS and NDI software components. In addition, it may be necessary to write "glue code" for those components that don't contain their own "plug-and-play" interfaces (these are not strictly "components" in the component-based engineering sense—but it is highly unlikely that enough "plug-and-play" components exist to safely provide the functionality required in a safety-critical system or a high-confidence security-critical system.

Custom development for component-based systems may include:

- "Add-on" logic to add functionality to, or modify the "out of the box" functionality of, a COTS/NDI component;
- Entire components needed to perform functions that cannot be performed by any COTS/NDI component, either due to obscurity of the functionality or lack of sufficiently safe/secure options among COTS/NDI components;
- Interface software to enable components to support standard interface technologies not supported "out of the box", or to map proprietary interfaces of legacy components to standard interfaces. Such interface software may include custom-developed APIs, proxies, gateways, filters, applets, and scripts;
- Safety and security controls and countermeasures (*e.g.*, input validation modules, wrappers, proxies, *etc.*).[47] It should be possible to determine the need for such controls/countermeasures by observing (through analysis and testing, as described in Section 6) components from their unsafe/non-secure behaviors in response to inputs and the dangerous nature of their outputs, and to estimate the impact of such behaviors and dangerous outputs on whole system's behavior (in terms of its safe and secure operation). The effectiveness of controls/countermeasures should be sufficient to minimize such estimated impact to an acceptable level.

3.4. COMPONENT ASSEMBLY ARCHITECTURE

In most systems, the components could be assembled in multiple ways to satisfy the system's functional objectives. However, there will likely be different safety and security, as well as performance and usability, effects from one assembly architecture vs. another.

Key factors that differentiate functionally equivalent assembly architectures are:

- **Quantity of components:** Architectures with fewer components are preferable *if* fewer components can be used without significantly increasing the complexity of each component. Also note that more components there are, the more complex the analyses required to determine not just the behaviors resulting from direct interactions between component pairs, but to trace the "chain of events" that may cause a distant upstream interaction to eventually impact a remote downstream component's behavior. Finally, depending on the number of untrustworthy components among the total components, the number of countermeasures and controls that need to be added to the system to use those untrustworthy components safely and securely may multiply significantly. The need to analyze and test the system will also increase, both to make sure not only that the many component interactions do not threaten safe/secure operation, and that the many controls/countermeasures do not interfere with or negate each other's effects;

- **Complexity of components:** This is related to the previous factor: a greater number of simple components will be easier to analyze individually, but more complex and time-consuming to analyze collectively, and to maintain. Code review and testing will probably be similar to what is needed for fewer, more complex components, given that the same amount of functionality and code needs to be reviewed/tested;

- **Sequence in which components perform their functions:** This may also have a bearing on the amount of time it takes to achieve an objective for which multiple components' functions are required;

- **Whether components are contiguous:** This is of particular concern in terms of exposure of trustworthy components to untrustworthy components;

- **Whether a component is exposed, to external users, processes, or other systems;**

- **Given multiple components able to perform the same function, which of those components will perform that function:** For example, in a Web server application, the operating system, backend database management system, and Web server application may all be capable of authenticating users. However, it is highly unlikely that the system will be designed to authenticate the user three times. Instead, a fourth component—a single sign-on system—may be added to the architecture to authenticate users on behalf of all the other components that require user authentication.

The best possible assembly architecture will:

- minimize, and preferably eliminate any external exposure of untrustworthy components, as such components are far more likely to fail as a result of a malicious user or other externally-originated input;

- rely only on trustworthy components to perform critical functions, including functions to achieve system fault-tolerance and security;

- isolate trustworthy components that perform critical functions to prevent any direct access to them /interaction with them by untrustworthy components;

- allow for the interpolation at component "boundaries" of safety and security controls/countermeasures (1) between two components, and (2) between components and the system's external interface(s);

- be the most easily adaptable, *i.e.*, to allow for the direct substitution of existing components with new components that have the same standard interfaces and comparable functionality, but which may be less trustworthy than the components they replace (thus requiring the addition of controls/countermeasures),[48] or the replacement of a group of components with a group that collectively provides the same functionality (in which case major architectural adjustment may be needed).

ADDITIONAL SUGGESTED RESOURCES

- Wallnau, Kurt, Judith Stafford, Scott Hissam, and Mark Klein, Carnegie Mellon University Software Engineering Institute, "On the Relationship of Software Architecture to Software Component Technology", in *Proceedings of the Sixth International Workshop on Component-Oriented Programming*, Budapest, Hungary, 19 June 2001. http://research.microsoft.com/en-us/um/people/cszypers/events/wcop2001/wallnaustaffordhissamklein.pdf

- Kalinsky, David, "Architecture of safety-critical systems", in *Embedded Systems Design*, 23 October 2005. http://www.embedded.com/columns/technicalinsights/169600396?_requestid=142048 (accessed 25 March 2010).

- Schmidt, Heinz, Monash University, and Iman Poernomo and Ralf Reussner, DSTC Pty. Ltd., "Trust-by-Contract: Modelling, Analysing and Predicting Behaviour of Software Architectures", in *Journal of Integrated Design and Process Science*, Volume 5 Issue 3, August 2001, Pages 25-51. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.10.6507&rep=rep1&type=pdf

[36] Clough, Anne, Charles Stark Draper Laboratory, for U.S. Department of Transportation Federal Railroad Administration, *Commercial-Off-The-Shelf (COTS) Hardware and Software for Train Control Applications: System Safety Considerations*, Final Report DOT/FRA/ORD-03/14, April 2003. http://www.fra.dot.gov/downloads/Research/ord0314.pdf (accessed 12 March 2010).

[37] *Op. cit.* Bachman, *et al.*

[38] Barnum, Sean, and Michael Gegick, "Securing the Weakest Link", 19 September 2005. https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/principles/356-BSI.html (accessed 14 January 2010).

[39] Vorobiev, Artem, and Jun Han, Swinburne University of Technology (Melbourne, Australia), "Secrobat: Secure and Robust Component-based Architectures", in *Proceedings of the 13th Asia Pacific Software Engineering Conference*, Bangalore, India, 6-8 December 2006. http://www.ict.swin.edu.au/personal/jhan/jhanPapers/apsec06secrobat.pdf (accessed 29 March 2010).

[40] For further examination of the dependency of safety on a presumption of security in safety-critical embedded systems, see: Michael Ellims, "Is Security Necessary for Safety?", presented at 4th Embedded Security in Cars Conference, Berlin, Germany, 14-15 November 2006. http://www.pi-shurlok.com/Shared/Uploads/NewsArticles/Files/security_and_safety.pdf (accessed 12 March 2010).

[41] It should be noted that committing to a commodity framework or "software stack" can result in the unintended isolation (or exclusion) of certain candidate components due to the inherent assumption that all components used with the framework will include the interfaces supported by that framework. If these are proprietary (*i.e.*, non-standard) interfaces, any component that does not support them will be inhibited from interacting with those that do. A more insidious problem is that of commodity frameworks that purport to provide a standard interface mechanism, such as Simple Object Access Protocol (SOAP), Lightweight Directory Access Protocol (LDAP), *etc.*, when in fact what they provide is a *version* of that protocol that has been augmented with vendor-proprietary extensions or "enhancements"; in many cases, the standard protocol specification explicitly allows for such extensions (*e.g.*, the American National Standards Institute's Standard Query Language specification). In such cases, conflicts between the vendor version of a standard protocol and those that speak the "pure" version can introduce errors into the interactions between those components. See: Gamble, M.T., and R.F. Gamble, University of Tulsa, "Isolating Mechanisms in COTS-based Systems", in *Proceedings of the Sixth International IEEE Conference on COTS-Based Software Systems*, Banff, Alberta, Canada, 26 February-2 March 2007, pages 33-40. http://www.seat.utulsa.edu/papers/ICCBSS07-Todd.pdf (accessed 31 March 2010). For other issues that may arise from use of commodity component frameworks see: Lau, Kung-Kiu, and Zheng Wang, "Software Component Models", in *IEEE Transactions on Software Engineering*, Volume 33 Number 10, October 2007. http://www.cs.man.ac.uk/~kung-kiu/pub/tse07.pdf (accessed 31 May 2010); also: Kung-Kiu Lau and Vladyslav Ukis, University of Manchester, "On Characteristics and Differences of Component Execution Environments", University of Manchester Preprint CSPP-41, February 2007. http://www.cs.man.ac.uk/~kung-kiu/pub/cspp41.pdf (accessed 31 May 2010).

[42] Navy Joint Software Systems Safety Engineering Workgroup, *Joint Software Systems Safety Engineering Handbook*, DRAFT Version 0.95, 30 September 2009.

[43] Defense Information Systems Agency (DISA) Applications Security Project, *Developer's Guide to Secure Use of Software Components*, Draft Version 3.0, 7 October 2004.

[44] *Op. cit.*, Meyer, "The Grand Challenge".

[45] Bishop, Peter, Robin Bloomfield, and Peter Froome, Adelard, "Justifying the use of software of uncertain pedigree (SOUP) in safety-related applications", Health and Safety Executive (United Kingdom) Contract Research Report 336/2001, 2001. http://www.hse.gov.uk/research/crr_pdf/2001/crr01336.pdf; also: http://www.sipi61508.com/ciks/bishop1.pdf

[46] This includes costs for maintenance, since such assessments/tests will need to be repeated for all new releases of the NDI components.

[47] For particularly critical interfaces, it may make sense to implement not one but two controls: a control for output of the "upstream" component—the one providing data to the "downstream" component. This will reduce the heavy dependency on the single control, and may improve performance. For example, an output validation filter for data sent by the upstream component could catch—and possibly sanitize—most instances of unacceptable (due to length, format, *etc.*) output. Then the input validation logic added to the downstream component would deal with only the

exceptional cases that were not caught by the output validation logic of the upstream component. Performance could be improved due to a reduced quantity of unacceptable input sent to the downstream component: an important consideration if that component's operation were more time-critical than that of the upstream component.

[48] By the same token, if the new component is more trustworthy, it may be possible to eliminate a control/countermeasure that was required when using the previous, less trustworthy component.

4. HAZARDS AND RISKS THAT ARISE FROM COMPOSITION

> *[T]o use a COTS component with confidence, it is necessary to ensure that, the impact of component behaviour on system safety is fully understood and managed. More specifically, it should be ensured that the potential failures of the COTS component cannot contribute to system level hazards.*[49]

Ideally, organizations should be able to rely on software developers with the familiarity with the safety- and security best-practices to generate all source code and components that will be used in the software-intensive system. With this level of control, the development organization would be able to ensure safety and security are considered during every phase of software development: including requirements, design, implementation, testing and deployment. This would allow the organization's software to perform all of its safety and security functions correctly.

However, developers are human, and thus inherently imperfect. Even if they are well-versed in safe software engineering practices, they are prone to make mistakes, some of which may lead directly—or indirectly—to safety or security flaws in the software-intensive system.

Developers are even more likely to make mistakes under pressure of last-minute addition of requirements or unrealistic deadlines. Moreover, lack of emphasis by management or customer on all but functional (and perhaps performance) requirements may lead to designs that are incomplete in terms of addressing safety and security imperatives.

Even with safety- and security-oriented software development methodologies and testing practices (and sufficient training/education to understand and use them), a developer cannot be expected to create software that is completely safe or secure. To this end, organizations have begun relying on reuse of existing components with the expectation that these components are more robust than their custom developed counterparts. While taking advantage of these components will increase the likelihood that the software will be more robust (*i.e.*, it will be tested and used by a larger number of users) there is no guarantee that the resulting software-intensive system will be as safe or secure as its custom-developed counterpart. When composing software-intensive system, it is important to keep in mind the number of hazards that can introduce risk into the system.

It is important to note that the hazards associated with software development are not new. The Therac-25 radiation therapy machine used in the mid-to-late 1980s was a software-intensive component-based system that resulted in the deaths of several individuals due to flaws within the system software.[50] One of the primary flaws associated with this system was that it relied entirely on software to perform safety-critical functions without performing a stringent review of the software itself. By re-using software components developed for other devices that incorporated hardware fail-safes, the developers of the Therac-25 assumed that the original software was sufficient for safety-critical functions. In fact, the developers of the Therac-25 made the following assumptions in their safety analysis:

1. Programming errors have been reduced by extensive testing on a hardware simulator and under field conditions on teletherapy units. Any residual software errors are not included in the analysis.

2. Program software does not degrade due to wear, fatigue, or reproduction process.

3. Computer execution errors are caused by faulty hardware components and by "soft" (random) errors induced by alpha particles and electromagnetic noise.

While a modern system will assume the existence of residual software errors—and potentially those induced by "soft" errors, it is important to realize that invalid assumptions may themselves lead to an unsafe or insecure system.

In particular, the Therac-25 suffered from obscure race conditions in the software input. Some failures of the software occurred as a result of the operator typing input faster than the operation of the Therac-25, resulting in incorrect dosages to be applied based on the original incorrect input. The following sub-sections highlight a number of potential areas in which software components may introduce security risks or safety hazards into a multi-component system.[51]

ADDITIONAL SUGGESTED RESOURCES

- Ying, Robin, Stevens Institute of Technology, "Building Systems Using Software Components", in DoD *SoftwareTech News*, Volume 9 Number 1, March 2006. http://www.softwaretechnews.com/stn_view.php?stn_id=4&article_id=10 (accessed 8 April 2010).

- Børretzen, Jon Arvid, "The impact of component-based development on software quality attributes", 2005. http://www.idi.ntnu.no/emner/dt8100/Essay2005/boerretzen.pdf (accessed 8 April 2010).

4.1. PARAMETER-PASSING ISSUES

Most software components require glue- or interface-code to supply certain parameters in order to function correctly. In such situations, it is important for the documentation—and the component itself—to explicitly identify when it expects input and what format that input should be in. However, there is always a possibility that component-based software will result in unexpected behavior, resulting in parameters being sent outside of predetermined boundaries. For example, components may send input that another component does not expect. There are several possible events that could occur:

- The components disagree on when to interact based on some external condition at run-time or based on a flawed policy;

- The specifications regarding a component's coupling with another component results in sending wrong data type;

- The sender receives unexpected input and passing it directly to the receiving component;
- Data is supplied to the wrong component, based on flaws in the sending component and/or interface code;
- Component does not receive input it does expect;

Conversely, a component may neglect to send input when another component expects input. Again, there are several possible events that could occur:

- A component "hangs" indefinitely while waiting for data to arrive from a second component that has crashed.
- The component expects certain data from another specific component but instead, the sender is now being represented by a third component who sends different data than what the receiver expects.
- The sender, based on a false positive, doesn't send.
- Sender or receiver not configured to interact with each other.

In general, most component-based software-intensive systems rely on some form of external coupling, where a software component has a dependency on software written by a third party. Upon invocation of a software component, problems can arise when the component accepts and processes parameterized data not of the length or format expected. The causes for component acceptance of bad data include:

- Component documentation that is unclear as to what data it expects to receive upon integration with other components or a larger system.
- Software developers may assume to know what data to pass based on the functionality of the component or based on what previous components of like functionality expected.
- The invoking software may have converted the data before actually making the invocation. However, this behavior might be expected if the programmer failed to re-cast the data before invoking the component.

Sometimes, interaction across more than two components may introduce inconsistencies in the system that lead to a loss of data. For example, a component may be unable to receive requested input or access service from a second component due to architectural interpolation of a third component (or other barrier) that severs the interface between the first component and the second component.

Often, one of the primary components within the system may be a software entity designed to be standalone—or to interact only with limited external interfaces (*e.g.*, physical sensors). In such cases, the component does not have the necessary interfaces to interact safely/securely with other components. Rewriting a standalone computer program such that it can be invoked as a

component from a system or other component requires a redesign of variable complexity. That complexity will depend on what language the program is written in, the modularity of the original design, as well as the goals of the component-based system as a whole.

The same is true with a computer program that has interfaced with hardware sensors. Here, in addition to the work required for a standalone program, the existing interfaces must be replaced with software interfaces. This will have to involve a deep analysis of the program's data flows.

As always, when integrating standalone components into a larger system, developers must be keenly aware of the resource expectations and consumptions of the individual components. If the collective expectations/consumptions of all components exceed the available resources, performance may be detrimentally affected, which in some safety-critical systems can have hazardous results. For this reason, a thorough analysis of component resource requirements vs. available resources should be undertaken when hosting multiple components on the same physical platform.

4.2. TIMING AND SEQUENCING ISSUES

While parameters are an important aspect to ensuring proper communication between components, developers must be keenly aware of timing and sequencing issues—particularly in real-time and/or distributed environments. With components increasingly being deployed in large-scale and small-scale software systems, ensuring the proper timing and sequencing of components becomes ever more important.

With large-scale component-based systems relying on input from one another, there is always the potential that one component will retrieve the "wrong data at the right time," leading to a situation where data was retrieved at the appropriate point in execution, but it is incorrect and may lead to safety or security defects within the system. For example, a weapon system may rely on an external component to provide position information for both aiming the turret and moving a targeting camera—with the same formatting supplied for both components. If the turret and camera receive the wrong information, it can be expected that the system will not function properly.

Conversely, one component may receive the "right data at the wrong time". In the turret example above, the targeting component may receive the correct data, but only after the weapon has received firing orders—causing the turret to fire *prior to* being positioned.

Increasingly, with components sharing the same networks, hardware, and interfaces, software systems have to deal with the possibility that high-priority functions may be interrupted by low-priority functions. For example, if a clock component is hosted on the same operating system as a targeting component, it is possible that a high number of requests to the clock component may cause resource exhaustion of the targeting component—causing the low-priority component to adversely affect the high-priority component. Regardless of how the components are deployed within the architecture, any architecture that allows multi-threaded execution of shared resources may eventually run into some form of race condition and—in so doing—directly impact the safety and security of the software intensive system as a whole.

4.3. RESOURCE CONFLICTS

Any architecture that relies on multiple independent components can be susceptible to resource deadlocks. In fact, *E.G.* Coffman identified four preconditions to introducing deadlocks into any system:[52]

- **Mutual exclusion condition:** a resource that cannot be used by more than one process at a time;
- **Hold and wait condition:** processes already holding resources may request new resources;
- **No preemption condition:** No resource can be forcibly removed from a process holding it, resources can be released only by the explicit action of the process;
- **Circular wait condition:** two or more processes form a circular chain where each process waits for a resource that the next process in the chain holds.

Removing any one of these conditions will prevent deadlocks from occurring. However, in practice, it becomes increasingly complex to address any one of these conditions in a large-scale system. To this end, large-scale databases actively monitor for resource deadlocks, forcefully resolving them at run-time.

If the component requesting the resource is authorized to make such a request at the time of the request, the resource should be released or shared if the resource's management algorithm allows sharing. If not, the request is queued and waits for the resource to be released. There are operating systems that permit the request to repeatedly ask for the resource until it is released; this is called spinning on a lock. Regardless of what mode a request waits in however, the resource must be released eventually; even if the task currently in possession of the resource has a high priority. In such cases, as a high priority task consumes additional time slices, either the priority or the length of the time slices is lowered. Still, is it possible for a component to fail to release a resource when operating normally.

When the functionality of a component is critical, the apparent failure of a component to release a resource may, in fact, be intended by the component's designer for safety reasons, *i.e.*, the resource is held in order to preempt the processing of any new request from another component. For example, suppose a surgeon operating a software-controlled surgical laser has reached the phase of an operation at which an operating system-controlled resource in the surgical laser's software system must remain exclusively available to a single specific application throughout the next phase of the operation. Through the application's graphical user interface (GUI), the surgeon will instruct the OS to lock the resource, which will cause all attempts by other components to access or claim the locked resource to fail. While technically a failure at the individual component level—*i.e.*, the requesting component failed to gain access to the requested resource—at the system level, this was not a failure at all; It was a resource conflict "by design": the resource was locked intentionally, to prevent a safety hazard that might have escalated into a mishap.

Another good example of the failure of a reasonable request for service by one component can occur when an on-line weapons system has identified an unfriendly target and needs to perform the calculation of a targeting position and launch trajectory followed by its weapon's launch sequence, without any interruption between the steps between calculation and launch. Any interruption could delay the calculation of target position and launch trajectory, which would cause the weapon to aim inaccurately, and not only miss its intended target, but to hit an unintended object (possibly causing the unintended casualty of one or more civilians). Were such an application multi-threaded, the attempted parallel activities of two or more processes could confuse the ability to establish this clear, simple, uninterrupted sequence of events.

Moreover, it would complicate the ability to determine whether a component's seemingly reasonable request for service was, in fact, maliciously intended (*e.g.*, through coding by a rogue developer and/or insertion of a logic bomb or Trojan horse) to cause exactly this type conflict that could lead to system failure…and safety mishap.

4.4. IMPROPER CONFIGURATION ISSUES

Even with appropriate component parameters, timing and resource allocation, component-based architectures must be configured to operate properly within the environment. In fact, one of the primary reasons for using COTS/NDI components within a software architecture are that these components can be configured to operate in a variety of ways. As such, organizations must ensure that the required configurations do not introduce potential safety or security defects into the system.

Of particular concern for security-related defects are the software's access privileges:

- **Component access privileges set too high:** This enables the component to potentially interfere with other components it is not intended to interact with, or access data, resources, or devices it is not intended to access.

- **Component access privileges set too low:** This prevents the component from interacting with other components it is supposed to interact with, or accessing data, resources, or devices it needs.

- **Component role set incorrectly:** Equivalent to setting some access privileges too low and others too high.

Any component that does not have appropriate access privileges may directly affect the safety of the system as a whole by introducing the potential for an attacker to inject an intentional safety hazard into the system. In addition, the *safe* operation of the component may have its own set of configuration controls, including:

- **Component fail-safe settings:** This determines how the component will behave if certain error conditions are satisfied.

- **Component safety thresholds:** This would determine what the safe operating parameters for the component are.

- **Component priority:** This would determine how the component must behave in concert with other aspects of the system (*e.g.*, setting real-time priority within the OS).

After a thorough safety and security analysis, those fixes and patches that will not detrimentally affect the safety (and safety-impacting security) of the system should be applied to the component prior to system deployment. To the extent possible, the version of each component deployed should reflect its most recent code-base, which is presumed to be that in which the most errors, flaws, and vulnerabilities have been remediated by the component's developers or integrators.

4.5. IMPROPER FUNCTIONALITY ISSUES

No matter how much discipline and testing a component undergoes, there is always the possibility that it may contain an undetected or uncorrected error that may result in an incorrect computation. If the component itself, or another component to which it passes the computation results, acts on those incorrect results, the result may trigger a fault with safety or safety-impacting security implications.

Concerns over improper functionality are not limited to the components themselves, but also to their execution environment—an environment level component (hardware sensor, operating system process) may malfunction, providing the component with incorrect data which, if the component acts on that data, may lead to an error, fault, and ultimately mishap. For example, a component that does not validate input from another component at the application or operating system level or from a hardware sensor, may inadvertently act upon unvalidated input that turns out to be incorrect. As a result, the component may behave in an unexpected, possibly harmful, manner.

There is also the possibility of a component calling a library routine or function that it expects to be present in the operating system, only to find that the library/function is unavailable, not present, or that an incorrect version has been installed. In any of these cases, the component may behave in an anomalous, possibly dangerous, way as a result.

4.6. INADEQUATE FUNCTIONALITY ISSUES

In some cases, a component may not have all of the functionality required to adequately perform its intended role in the system. For example, the component may be unable to ensure that communication between it and another component is properly authenticated, verified and validated. The component may also not be able to handle parameters of sufficient precision for the software's operation. In either case, developing to the component's limitations may lead to safety or security defects within the system. Use of functionally deficient components is often inspired by desired cost savings. The result is either that the system stakeholders accept a system that does not satisfy all of its requirements (or the requirements need to be adjusted).

Furthermore, use of inadequate components makes it more difficult to isolate problems that may be caused by the inadequate functionality during system operation.

Alternately, wrappers or externally called mechanisms may be implemented, if possible, to augment the deficient component. In some cases, this may be accomplished by locating yet another component to fill in the missing functionality. More often, it will require the development of custom software—which is exactly what use of the pre-existing component was supposed to avoid—thus eliminating the cost savings of using the component. The result, whether another component or custom code is added, is a system that is less elegant, more complex, and more difficult to analyze and maintain. It is better to use components that meet all of their specifications in the first place, as long as those components do not introduce unmitigatable hazards or vulnerabilities.

4.7. UNANTICIPATED EXECUTION OF UNUSED/DORMANT CODE

NDI components are often designed to provide a large variety of functions, so they can support a wide range of user applications. Such components typically provide functions that are not needed in the specific system in which they are used. Generally, these unused functions can be disabled using configuration mechanisms—or possibly left enabled with the expectation that they will not be triggered (called or executed) in the course of the expected operation of the system. Such unused, or "dormant", functionality can, however, be triggered by an unanticipated event, such as an unexpected input that is (mis)interpreted by the system as a command. This could result in the unanticipated, potentially dangerous, behavior of the system as a whole due to the unanticipated execution of a function that was not intended to execute at that time (if ever).

For example, a large software-intensive system that has a generic installation of a commodity operating system may have support for running processes at pre-determined times. In a sensitive environment, it is possible that some of the activated processes may interfere with the behavior of the system (*e.g.*, modifying files or starving safety-critical functions of resources).

ADDITIONAL SUGGESTED RESOURCES

- Rieger, William B., Boeing Defense and Space Group, "Safety Considerations of Unused Code: Its Causes and Implications", in *CrossTalk: The Journal of Defense Software Engineering*, January 1997. http://www.stsc.hill.af.mil/crosstalk/frames.asp?uri=1997/01/saftyconsider.asp (accessed 8 April 2010).

4.8. DATA PROTECTION ISSUES

When relying on components to process and store data generated within the software-intensive system it is important to note that different components may not provide equivalent levels of data protection—especially when they are written in different programming languages. For example, a component written in the Perl scripting language does not include inherent checks that ensure that data being processed is numerical rather than textual (the danger being that a Perl process

may fail due to inappropriately-formatted data being accepted by the application). For Perl components, such checks would have to be explicitly programmed, whereas comparable components written in C or Java would have these sorts of checks built in through their implementation language.

Component interface code that assumes that other components perform certain checks will introduce errors into the system when such checks do not occur. For this reason, the developer needs to:

1. determine all of the checks that each component's interface expects;
2. determine whether the expected checks are built into all other components with which the first component may interact/communicate;
3. implement wrapper code or another mechanism to add expected checks to any components in which they are not already present but may at some point be needed.

Components may also have different internal requirements for ensuring that the data is valid or securely stored. For example, security- or safety-critical data should only be stored in trusted locations on the computing platform with appropriate data-level safeguards in place (*e.g.*, file permissions, data encryption or digital signatures). Without a process in place for ensuring that an individual component performs these required activities and provides the required safeguards, it becomes possible for external entities to either inadvertently or maliciously affect the outcome of the system by taking advantage of the component's data protection profile.

4.9. INHERENT COMPONENT ISSUES

Finally, it is important for organizations to be aware of security or safety issues that may inherently exist within the component itself. We mention these briefly to aid the developer in defining criteria for selection of components.

- A component may contain malicious logic that affects its interactions with other components.
- A component may contain faulty logic that affects its interactions with other components.
- The assembly architecture for the system may expose the component's weaknesses to access by other components, system- or middleware-level entities, or human users.

In each of these cases, once the components are assembled, it will be difficult to isolate the root cause of any unsafe or non-secure behavior or fault down to the individual component level. For this reason, it is important that the individual components' behaviors and vulnerabilities all be well-understood before assembly begins. This understanding can be gained through individual component testing and analysis, to identify errors, anomalous or malicious logic, and exploitable vulnerabilities.

The level of analysis/testing performed should be commensurate with the level of assurance desired that the component behaves safely and securely in and of itself (*i.e.*, when not interacting with other components, as single-component testing/analysis will not demonstrate this). The level of assurance should reflect the quantity and intensity of safety hazards (determined through hazard analysis) and security risks (determined through threat modeling) to which the component (and system) is expected to be exposed on the one hand, and the safety criticality of the system on the other. The more safety critical the system, the more assurance will be needed that its components will not compromise that safety. Similarly, the more hazards/threats to which the component is expected to be exposed, the greater the assurance needed that those hazards/threats will not compromise its safe operation.

As with any testing process, it is infeasible to develop tests that will guarantee the security of safety of the component. Nevertheless, organizations should determine what level of assurance is necessary when performing an analysis of individual components. After each component is deemed internally sound, pairs of components that will interact during system operation should be tested, then larger combinations, until ultimately the whole system is tested, with each component's behavior within the system both fully exercised and carefully monitored to ensure that component-level logic errors, malicious behaviors, or vulnerabilities are identified (as early in the development life cycle as possible) and mitigated prior to deployment of the system as a whole.

ADDITIONAL SUGGESTED RESOURCES

- Lipson, Howard F., Nancy R. Mead, and Andrew P. Moore, CMU SEI, "Can We Ever Build Survivable Systems from COTS Components?", Technical Note CMU/SEI-2001-TN-030, December 2001. http://www.cert.org/archive/pdf/01tn030.pdf (accessed 14 January 2010).
- Voas, Jeffrey, "Mitigating the Potential for Damage Caused by COTS and Third-Party Software Failures", in *Proceedings of the 15th International Conference and Exposition on Testing Computer Software*, Washington, DC, 12-18 June 1998 http://www.cigital.com/papers/download/ven.pdf
- Voas, Jeffrey, F. Charron, and Keith Miller, "Tolerant Software Interfaces: Can COTS-based Systems be Trusted Without Them?" in *Proceedings of the 15th Int'l. Conference on Computer Safety, Reliability, and Security*, Vienna, Austria, October 1996. http://www.cigital.com/papers/download/safecomp96.pdf (accessed 31 March 2010).
- Voas, Jeffrey, Anup Ghosh, Gary McGraw, and Keith Miller, "Glueing Together Software Components: How Good is Your Glue?" in *Proceedings of Pacific Northwest Software Quality Conference*, Portland, OR, October 1996. http://www.cigital.com/papers/download/pnsqc96.pdf (accessed 31 March 2010).

[49] Ye, Fan, and Tim Kelly, University of York, "Contract-Based Justification for COTS Component within Safety-Critical Applications", in *Proceedings of the the 9th Australian Workshop on Safety RelatedProgrammable Systems*, Brisbane, Australia, October 2004. http://www.acs.org.au/documents/public/crpit/CRPITV47Ye.pdf (accessed 12 March 2010).

[50] Leveson, Nancy G., and Turner, Clark, "An Investigation of the Therac-25 Accidents", in *IEEE Computer*, Volume 26, Number 7, July 1993, pages 18-41. http://cs.ucla.edu/~kohler/class/05f-osp/ref/leveson93investigation.pdf (accessed 18 December 2009).

[51] Section 5 of *COTS Hardware and Software for Train Control Applications* elaborates on safety considerations discussed here specifically as they pertain to COTS operating systems. *Op. cit.* Clough/Federal Railroad Administration.

[52] Coffman, Jr., *E.G.*, *et al.*, "System Deadlocks", in *Computing Surveys*, Volume 3, Number. 2, June 1971, pp. 67-78.

5. MODELING AND PREDICTIVE ANALYSIS OF HAZARDS AND RISKS THAT EMERGE FROM COMPOSITION

> *There is debate within the software engineering community over the merits or otherwise of inferring the quality of fielded software code from the quality of the people, processes, and tools that an organization uses to develop software products. It is generally accepted that whilst quality processes are a necessary enabler to produce quality code, processes alone are not a guarantee of quality code. It is for this reason, that the bulk of software processes for safety-critical software are in fact the software product verification processes that include a wide range of activities from requirements verification through unit testing to final qualification testing. Given the practical impossibility of performing exhaustive testing, an important consideration for the verification of safety-critical software is the measure of coverage that is provided by the verification effort. The quantity and type of verification that is performed on software will determine the level of assurance that can be ascribed to the fielded product.*[53]

When planning to introduce components into an architecture, organizations can use a number of tools and methods to determine—ahead of time—the likely hazards and risks that will emerge in the overall system once multiple components are incorporated into a single system.

Evaluators, analysts, and testers of software components and component-based systems may find the information and descriptions of tools capabilities in the NAVSEA NOSSA *Software Security Assessment Tools Review*[54] of help in planning of analyses and tests and identification of appropriate tools to support those analyses/tests.

ADDITIONAL SUGGESTED RESOURCES

- Esker, Linda J., "Gaining Early Insight Into Software Safety: Measures of Potential Problems and Risks", presented at the 19th Systems and Software Technology Conference, Tampa Bay, FL, 18-21 June 2007. http://sstc-online.org/2007/pdfs/LJE1669.pdf (accessed 8 April 2010).

5.1. TECHNIQUES AND TOOLS FOR MODELING, SIMULATION, AND ANALYSIS IN AID OF HAZARD, RISK, AND VULNERABILITY PREDICTION

Developers of safety- and security-critical systems often use a number of modeling and simulation tools to simulate and observe the types of unsafe behaviors that might be encountered in a production system. Balestrini-Robinson *et al.* suggest that agent-based models are the best suited for modeling intercomponent interactions and dynamic and emergent behaviors of components in complex systems. They warn, however, that of all modeling techniques, agent-based models are the most difficult to implement and validate.[55] McDermott, *et al.*, have described the benefits of using the Air Force Modeling and Simulation Training Toolkit

(AFMSTT) for modeling and simulation (M&S) of SoS;[56] the advantages they describe may also hold true when using the AFMSTT for M&S of large, complex component-based systems.

Binary fault injection and source code fault injection tools can provide insight into how the software will function in certain scenarios, the results of which can directly be applied to a safety and security assurance case.[57] In addition, organizations can use a number of simulation tools to provide a realistic run-time environment in which different scenarios can be played out, showing how well the system and its individual components behave in a hostile or hazardous environment.

As noted in the DOD *Joint Software Systems Safety Engineering Handbook*:

> *Modeling and simulation are often the only means available to analyze the interaction of components in a system.... The models and simulators must possess a sufficient pedigree and proven accuracy to allow the safety team to achieve their objectives. The models and simulators must also be accurate enough to react in the same manner as the actual component or system. The models and simulators, because they are used to validate safety, must be classified as safety-critical themselves and be under configuration control, and ultimately a deliverable.*

Safety code reviews and formal methods can also be useful for verifying the safety and security of individual components. Safety and security code reviews differ from traditional code reviews in that their goal is not simply to find errors or defects within the software. Instead, the goals of a safety and security code review are to identify the safety-critical and safety-impacting security functions and parameters of the software and ensure they satisfy the requirements that specify how the software should prevent and respond to safety hazards and security errors during runtime.[58] As such, a code review clarifies whether the software satisfies its requirements for each safety-relevant and safety-impacting security-relevant parameter and function. The specific techniques for such reviews are static and dynamic code analysis. These are discussed further in Section 7.3.

Formal methods provide a means by which assurance of software safety can be extrapolated from the mathematical proof of the formally (*i.e.*, mathematically) specified requirements for the software and a formally-specified model of the software as it would operate if implemented correctly and in strict conformance with its formal specification.

Also helpful may be the analysis of indirect evidence of a component's "goodness". Such evidence will demonstrate, for example that the component was developed by a highly-regarded, trustworthy software vendor (*i.e.*, "good" pedigree), that the component has not been tampered with or corrupted over its life time (*i.e.*, "good" provenance), that the component has attained relevant safety or security certifications, that the component conforms with certain standards, *etc.* None of this evidence on its own proves that the component is safe or secure; it can, however, tip the scale in favor of trusting a component whose safety/security cannot be adequately determined from direct analysis alone.

Developers of security-critical systems may also rely on risk/threat modeling and analysis tools to provide an understanding of the threats that may directly affect the system. These tools allow for the modeling and identification of threats.

According to Donald Firesmith,[59] safety risk analysis is performed using one of four techniques, captured in Table 5-1. These techniques are similar to those defined by Firesmith for performing security risk analysis, which are also captured alongside their safety risk analysis counterparts, in Table 5-1.

Table 5-1. Safety Risk Analysis compared with Security Risk Analysis

Technique	Safety analysis connotation	Security analysis connotation
Asset analysis	Determine which assets are valuable to legitimate stakeholders and how valuable they are.	Determine the assets that are sufficiently valuable to legitimate stakeholders to be worth protecting from attackers.
Harm analysis	Determine the credible types and severities of the accidental harm that can occur to these assets, whereby harm severities are typically categories of the magnitude of harm.	Determine the types and severities of the malicious harm that can occur to these assets so that the level of investment in security countermeasures will be commensurate with the value of the assets being protected.
Incident analysis	Determine the kinds of accidents that can cause harm to the assets as well as the kinds of near misses that need to be addressed.	Determine the kinds of security attacks that could cause malicious harm as well as the kinds of probes to be addressed.
Hazard analysis	Determine the hazards (*i.e.*, hazardous conditions) that can lead to safety incidents as well as their causes and consequences.	
Threat analysis		Determine the threats (*i.e.*, threatening conditions) that can lead to security incidents. Because security involves attacks, these threats can be considered to be the existence of attackers with specific profiles (*e.g.*, means, motive, and opportunity) or the proactive tools of their trade (*e.g.*, viruses and worms). Misuse and abuse cases can be useful tools as part of threat analysis, as can threat models, attack trees, and/or attack graphs.
Risk analysis		Categorize the security incidents and threats by levels of security risk,

		such as intolerable, undesirable, as low as reasonably practical (ALARP), and acceptable.

By merging each of these techniques, organizations can develop a streamlined safety and security risk analysis process. In particular, the asset and harm analysis steps of both types of analysis are complementary—as any security defects within a safety critical system necessarily become safety hazards.

In 2002, Arslan Brömme[60] identified several different types of hazard analysis that organizations may take advantage of. Of note, each of these has a security risk analysis counterpart:

- Preliminary Hazard Analysis (PHA),
- System Hazard Analysis (SHA),
- Subsystem Hazard Analysis (SSHA),
- Software Hazard Analysis (SHA),
- Operating Hazard Analysis (OHA),
- Maintenance Hazard Analysis (MHA),
- Fault Hazard Analysis (FHA).

Each of these hazard analyses can be performed in concert with a risk analysis at the associated level. For example, organizations can categorize the risks facing an application into several different families based on the type of hazard analysis performed: preliminary analysis, system-level analysis, software-level analysis, operating and maintenance analysis and analysis of *intentional* faults that may be induced by an attacker.

ADDITIONAL SUGGESTED RESOURCES

- Dahll, Gustav, Organisation for Economic Co-operation and Development, "Combining disparate sources of information in the safety assessment of software-based systems", in *Nuclear Engineering and Design*, Volume 195 Issue 3, 2 February 2000, pages 307-319.

5.1.1. FMEA, FMECA, and Hazard Analysis for Component-Based Software

As noted in the *Joint Software Systems Safety Engineering Handbook*, because software has no physical failure modes, Failure Modes and Effects Analysis (FMEA), Failure Mode, Effects, and Criticality Analysis (FMECA),[61] and related analyses can be difficult to apply to software-intensive systems. This said, software does have functions that can be implemented incorrectly, operate erroneously, or fail to operate at all for various reasons. For component-based software, FMEA/FMECA that strives to identify the causes and potential severity (or criticality, in

FMECA parlance) of failures of software functions needs to consider not just errors within individual software components, but errors that may arise from "mismatches", such as expected-but-not-received, unexpected, or incorrectly-formatted input from or output to other components.

Hazards are best identified at the system or subsystem level. In the case of component-based software systems, hazards will begin to emerge as multiple software components interact with each other and with the system's hardware components. Researchers at the Advanced Physics Laboratory of Johns Hopkins University developed an extended version of existing FMEA and FMECA methodologies to specifically address hazard severity analysis for SoS interfaces.[62] The new methodology, Interface FMECA (iFMECA) enables engineers to perform a risk-based prioritization of interfaces in SoS, and could be applied to intercomponent interfaces in component-based systems.

Code review is useful for revealing coding errors (*e.g.*, syntax errors, exceeding array boundaries, *etc.*) in individual components. Code review can also reveal causal factors that contribute to hazards that were identified during hazard analysis. In analyzing component-based software code, particular attention should be paid to identifying errors at the component's interface boundaries. Pair-wise analyses should also be performed to ensure that the interface code of one component matches the expectations of the interface code of the second component with which it interacts. Such analyses should also look for explicit input validation code associated with each interface, to ensure that input received from other components is always checked before use by that component.

ADDITIONAL SUGGESTED RESOURCES

- Ippolito, Laura M., and Dolores R. Wallace, National Institute of Standards and Technology (NIST), *A Study on Hazard Analysis in High Integrity Software Standards and Guidelines*, NISTIR 5589, January 1995. http://hissa.nist.gov/HHRFdata/Artifacts/ITLdoc/5589/ (accessed 31 March 2010).
- Guzie, Gary L., Army Research Laboratory (ARL), *Vulnerability Risk Assessment*, Technical Report ARL-TR-1045, June 2000. http://www.arl.army.mil/arlreports/2000/ARL-TR-1045.pdf (accessed 31 March 2010).
- Hecht, Myron, "Failure Modes and Effects Analyses for Large Software Intensive Systems", presented at 20th Systems and Software Technology Conference, Las Vegas, NV, 29 April-02 May 2008. http://sstc-online.org/2008/pdfs/MH2009.pdf (accessed 8 April 2010).

5.1.2. Techniques and Tools for Analyzing Components for Potential Compositional Issues

Developers of security-critical systems may also rely on risk modeling and analysis tools to provide an understanding of the hazards and threats that may directly affect the system. These tools allow for the modeling and identification of threats and hazards.

Static analysis tools provide the automated review of software source code, the results of which can identify potential vulnerabilities or safety hazards within the software occurring as a direct result of coding errors. While these tools are often used to perform a review of the software at the end of its development life cycle, taking advantage of static analysis tools throughout the life cycle—not just for review of custom-developed code, but also for open source code and other reused source code (*e.g.*, legacy code)—will enable problems to be identified, and mitigated, earlier—and at lower cost. In some cases, organizations may prevent potentially insecure or unsafe code from entering into the code base at all. Nevertheless, it is important to note that most static analysis tools are developed either for security reviews or safety reviews, requiring a security- and safety-critical system to leverage multiple tools to achieve their goals.

Unit testing is a software development technique that allows developers to test individual units of source code within the software as they are being developed. In particular, unit testing is increasingly becoming an important aspect of many software development methods—as a primary goal of many agile methods is to fully integrate testing into the software development cycle. Due to the flexibility of unit-testing, organizations can introduce safety and security tests into the unit-testing structure (which is often integrated with the compilation phase of a development environment). By merging the test and development phases of the traditional waterfall model, developers can ensure that any functional, safety or security defects are discovered as soon as they are introduced into the code, reducing the effort and resources necessary to remediate potential hazards.

Fuzz testing is the introduction of randomly generated tests into the software testing environment. By randomizing either the inputs, or various aspects of inputs to different functions or components of the software, organizations have a higher chance of discovering potential corner-cases within the code that may not behave as expected—potentially discovering safety or security defects in the process. For example, organizations might introduce a string of random values into a component that expects only certain types of data (*e.g.*, only capital letters), resulting in a test of the input validation techniques developed for the component. Fuzz testing can also be used to simulate the types of faults that might occur in an operational environment as a result of either intentional attacks or random safety hazards.

ADDITIONAL SUGGESTED RESOURCES

- Schmidt, Heinz, Monash University, "Trustworthy components—compositionality and prediction", in *Journal of Systems and Software*, Volume 65 Issue 3, 15 March 2003, Pages 215-225.

5.1.3. Techniques and Tools for Inter-Component Behavioral Analysis

Inter-component behavioral analysis provides developers with an understanding of how different components within the system interact with one another—with the goal of identifying any security-relevant or safety-relevant risks introduced by composition. Using dynamic analysis, integrators run a client or harness application that performs a sequence of commands on the software, analyzing how the software reacts to these activities. Dynamic analysis can range from

integrated tool suites that aim to discover vulnerabilities or defects within the software to test scripts designed to assess how the overall application behaves as users perform intentional and unintentional operations on the software. Unlike static analysis, which only looks at the code-level interaction between software, dynamic analysis has the potential to identify defects or hazard that occur as a direct result of the run-time configuration of components. For example, dynamic analysis should be able to determine if deadlocks occur as a result of component configuration—rather than as a result of code-level deadlocks. By integrating with fuzz testing and binary fault injection, organizations can put the multi-component system into a series of stress tests aimed to simulate the potential hazardous or malicious environment in which the software will be deployed—allowing for the discovery of safety- or security-critical defects that occur as a direct result of inter-component behavior.

One of the most commonly used techniques for performing inter-component behavioral analysis is the use of integration testing, where organizations take components that have been previously unit-tested and test the system as a whole with the goal of identifying any potential integration-level defects within the software. In many cases, integration testing simply involves ensuring that the expected behavior of the software as a whole occurs, but integration testing can be expanded to include inter-component tests that specifically target individual component interfaces. Using test harnesses to simulate the inputs to multiple components, integration testing can be performed on each component interface separately, including tests for security defects and safety hazards as well as regression tests to ensure that the component behaves exactly as previous components have in the past. Depending on the level of integration testing performed, organizations can have a better understanding of individual component interfaces while simultaneously ensuring that the system as a whole operates as expected.

Because any security defects may lead to a situation where an attacker can introduce an intentional safety hazard, organizations can use penetration testing and vulnerability scanning to identify potential vulnerabilities that may arise as a result of inter-component behavior. In penetration testing, the tester attempts to gain unauthorized access to the system through any means available (*e.g.*, discovering buffer overflows, SQL injections and/or performing social engineering attacks). A comprehensive penetration test can provide a wealth of information that can then be used to increase system security. However, one important aspect of penetration testing is the requirement that the penetration tester have a full understanding of the system. While most attackers will not have full working knowledge of each of the components, providing this information to a penetration tester will increase the likelihood of finding vulnerabilities as well as improving the overall value of the analysis.

Specifically, penetration testers should be able to provide some insight into the general security of individual components and how they affect the system as a whole. For example, a penetration tester may identify vulnerabilities in the Java runtime environment that may allow for modification of running code. In addition to penetration testing, organizations may also use vulnerability analysis, which will scan the individual components of the system for known vulnerabilities (*e.g.*, scanning the operating system and COTS products installed on the system). Any vulnerabilities identified by this analysis indicate components that are not fully patched—which may server as valuable targets for the penetration tester.

Section 7.3 provides additional information on techniques discussed here.

ADDITIONAL SUGGESTED RESOURCES

- Möller, Anders, Mikael Nolin, Ian Peake, and Heinz W. Schmidt, "Probabilistic Analysis and Predictions of Component-Based Real-Time Systems", in *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, Palma de Mallorca, Balearic Islands, Spain, July 2005. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.105.3403&rep=rep1&type=pdf

- Khan, Khaled M., University of Western Sydney (Sydney, Australia), and Jun Han, Swinburne University of Technology (Melbourne, Australia), "A Security Characterisation Framework for Trustworthy Component Based Software Systems", in *Proceedings of the 27th Annual International Computer Software and Applications Conference*, Dallas, TX, 3-6 November 2003. http://www.ict.swin.edu.au/personal/jhan/jhanPapers/compsac03.pdf (accessed 29 March 2010).

---

[53] *Op. cit.* Meacham, *Standards Interoperability.*

[54] Winograd, Theodore, Holly Lynne McKinley Schmidt for NAVSEA NOSSA, *Software Security Assessment Tools Review*, Draft 2 March 2009.

[55] Balestrini-Robinson, S., J.M. Zentner, and T.R. Ender, Georgia Tech Research Institute, "On Modeling and Simulation Methods for Capturing Emergent Behaviors for Systems of Systems", presented at the National Defense Industrial Association 12th Annual Systems Engineering Conference, San Diego, CA, 26-29 October 2009. http://www.dtic.mil/ndia/2009systemengr/9041WednesdayTrack6Zentner.pdf (accessed 31 March 2010). Also: http://www.acq.osd.mil/sse/webinars/2010-03-09-SoSECIE-M&S-Emergent-Behaviors-Zentner-etal-brief.pdf (accessed 31 March 2010).

[56] McDermott, Edwin P., Sharam Sarkani, and Thomas A. Mazzuchi, George Washington University, "Air Force Modeling and Simulation Training Toolkit (AFMSTT)", presented at the National Defense Industrial Association 12th Annual Systems Engineering Conference, San Diego, CA, 26-29 October 2009. http://www.dtic.mil/ndia/2009systemengr/8825WednesdayTrack4McDermott.pdf (accessed 31 March 2010).

[57] *Op. cit.* Ye and Kelly, *Contract-Based Justification,* which describes a proposed framework and methodology for developing Compositional Safety Cases.

[58] An example of an automated version of such a code review can be found in Ewen Denney and Bernd Fischer, NASA, Generating Code Review Documentation for Auto-Generated Mission-Critical Software, in *Proceedings of the Third IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT),* Pasadena, CA, 19-23 July 2009. http://ti.arc.nasa.gov/m/pub/580/SMC-IT-Denney.pdf (accessed 14 January 2010).

[59] Firesmith, Donald. G., "Analyzing the Security Significance of System Requirements", *Proceedings of the Symposium on Requirements Engineering for Information Security,* Paris, France, 25 August 2005. http://crinfo.univ-paris1.fr/RE05/W06/short3_firesmith.pdf (accessed 18 December 2009) *or* http://www.sreis.org/SREIS_05_Program/short3_firesmith.pdf (accessed 14 January 2010).

[60] Brömme, Arslan, "Hazard Analysis of IT Systems", presented at the GI FB Sicherheit Schutz und Zuverlässigkeit Workshop on Joint Terminology for Safety and Security, Frankfurt a.M., Germany, 12 July 2002. http://www11.informatik.uni-erlangen.de/Kooperationen/Veranstaltungen/Sicherheit1/PositionPapers/Broemme.pdf (accessed 14 January 2010).

[61] Failure Mode, Effects, and Criticality Analysis (FMECA) is an extension of Failure Mode and Effects Analysis (FMEA). FMECA extends FMEA by including a criticality analysis, which is used to chart the probability of failure modes against the severity of their consequences. It is preferred over FEMA for analysis of hazards in space systems and military systems of the North Atlantic Treaty Organization (NATO). See: DOD, *Military Standard Procedures for Performing a Failure Mode, Effects, and Criticality Analysis*, MIL-STD-1629A (Cancelled), 24 November 1980. http://www.fmeainfocentre.com/handbooks/milstd1629.pdf (accessed 31 March 2010). Also MIL-STD-1629A Notice 1, 7 June 1983. http://www.sre.org/pubs/Mil-Std-1629A(1).pdf (accessed 31 March 2010). See also: Department of the Army, *Failure Modes, Effects and Criticality Analysis (FMECA) for Command, Control, Communications, Computer, Intelligence, Surveillance, and Reconnaissance (C4ISR) Facilities*, Technical Manual TM 5-698-4, 29 September 2006. http://www.army.mil/USAPA/eng/DR_pubs/dr_a/pdf/tm5_698_4.pdf (accessed 31 March 2010).

[62] Mayoral, Leo, and Clay Smith, Johns Hopkins University Advanced Physics Laboratory, "Extending FMECA to System of Systems (SoS) Interfaces", presented at the National Defense Industrial Association 12th Annual Systems Engineering Conference, San Diego, CA, 26-29 October 2009. http://www.dtic.mil/ndia/2009systemengr/8866WednesdayTrack6Mayoral.pdf (accessed 31 March 2010).

6. ARCHITECTURAL COUNTERMEASURES TO EMERGENT HAZARDS AND RISKS

Once the risk and hazard analysis is complete, the developer needs to determine whether it is possible to mitigate or counteract any security or safety issues identified in individual COTS, open source, or legacy components, or in the assembly architecture by which those components are to be integrated. In worst cases, such mitigations/countermeasures will not be possible, and a software component(s) will have to be replaced. In other cases, the functionality of the problematic component can be modified, either by directly modifying or extending the component itself (*e.g.*, adding a wrapper), or by adding into the architecture additional software and/or hardware components (*e.g.*, application level firewall, virtual machine sandbox). This section explores the various mitigations and countermeasures available, aligned with their main objectives.

ADDITIONAL SUGGESTED RESOURCES

- Chen, Luping, and John May, University of Bristol (UK), "Methods for Enhanced Safety Wrapper Design", in *Proceedings of Informatik 2004, the 34th Annual Meeting of the Society for Computer Science*, Ulm, Germany, 20-24 September 2004. http://subs.emis.de/LNI/Proceedings/Proceedings50/GI-Proceedings.50-13.pdf (accessed 29 March 2010).
- Anderson, Tom, Brian Randall, and Alexander Romanovsky, University of Newcastle-upon-Tyne, Wrapping the Future, in Proceedings of the International Federation for Information Processing, in IFIP International Federation for Information Processing 18th World Computer Congress, Toulouse, France, 22–27 August 2004.
- van der Meulen, Meine J.P., Steve Riddle, Lorenzo Strigini, and Nigel Jefferson, University of Newcastle upon Tyne, *Protective Wrapping of Off-the-Shelf Components*, Technical Report Series CS-TR-857, August 2004. http://www.cs.ncl.ac.uk/publications/trs/papers/857.pdf (accessed 29 March 2010).
- Papp, Zoltán, and Allard Zoutendijk, TNO Instistute of Applied Physics, "A Runtime Framework for System Safety", in *Proceedings of the 2003 Intelligent Vehicles Symposium*, Columbus, OH, 9-11 June 2003.
- Freeman, James W., "Defining and Employing a System of Systems COTS Strategy for Information Assurance (IA)", presented at the 19th Systems and Software Technology Conference, Tampa Bay, FL, 18-21 June 2007. http://sstc-online.org/2007/pdfs/JWF1709.pdf (accessed 8 April 2010).

6.1. CONSTRAINED EXECUTION ENVIRONMENTS

The purpose of execution environment constraints on component execution is to:

- Prevent unintended execution of unused functions and access to "dormant" and "dead" code;
- Isolation of components whose behaviors fall outside specified acceptable limits, or which are deemed unsafe or untrustworthy for any reason, to prevent their execution from affecting the safe/trustworthy components of the system;
- Enforce separation of "trusted" components from "untrusted" components;
- Constrain anomalous component behaviors.

Most tools that are used to isolate software components can be configured to impose specific constraints on the capabilities of components within those isolated environments. In this way, not only does the isolation of such components limit the "reach" of the results of any of their misbehaviors (so that the outcome does not affect the correct operation of any component outside of their isolated execution space), but the resources available to the component, the accesses they are allowed to perform, *etc.* can be limited to prevent certain types of misbehaviors from occurring at all.

For example, a virtual machine can be configured to allow only certain types of communication to occur between any entity within the virtual machine and any external entity; with such a constraint, the output generated by any misbehaving component within the virtual machine can be essentially "blocked" from exiting the virtual machine and reaching any component outside it.

Application frameworks that enable the enforcement of policies based on presence or absence of validated code signatures can also be used to limit the actions of components. For example, an unsigned component could be limited to performing only those functions known to be required by its role within a system as a whole, thereby preventing any unintended functions.

6.1.1. Virtual Machines and Trusted Processor Modules

In systems in which very high risk components may need to operate alongside very high-confidence "trusted" components, it may be desirable to combine these two approaches, *i.e.*, isolate the untrustworthy ("untrusted") components in a sandbox, and isolate the high-confidence components in a TPM. In this way, the architecture will interpose "two degrees of separation" between the high risk/untrusted components and the high-confidence/trusted components.

A less robust alternative to the TPM and VM isolation approaches is use of the permission-granting capabilities provided by application frameworks such as Java and .NET. These mechanisms not only enable the developer of the system to grant permissions to signed code, but can be configured to limit the functionality of unsigned code that executes within the framework.

Many UNIX/Linux-based and Real-Time OSs have security modules and separation kernels already integrated; in some cases, the operation of these modules/kernels can tailored by the developer/integrator, *e.g.*, to ensure isolation of running processes based on a pre-defined policy. For example, the Linux kernel supports the designation of hard and soft real-time processes,

whereby certain processes can be guaranteed to run at specific intervals in time. This allows organizations to limit communications between trusted and untrusted processes while ensuring that any required activities can be completed in a timely manner—preventing any safety hazards from arising due to resource starvation or misbehaving components.

For systems running in native UNIX or Linux without a VM, TPM, or application framework, the operating system's permission-granting capabilities enable the configuration of "*chroot* jails", in which untrusted code can be isolated to an extent by restricting the file system resources available to that running process.

6.1.2. Code Signature as the Basis for Component Isolation

A common mechanism for indicating "goodness" of a component is the digital code signature. This code signature can be applied after component analysis and testing, as a kind of "seal of approval" that indicates that the component has been found to be safe and trustworthy. At run time, the signature can be validated to ensure that the component has not been modified since the signature was applied.

If a component has no signature, or its signature cannot be validated at run time, the component can be moved into a virtual machine sandbox that will isolate it so that the results of its execution cannot affect the rest of the system.

The code signature can also be used to determine which components can be "trusted". The architecture can also isolate those components, *e.g.*, in a different virtual machine (VM) or in a hardware Trusted Platform Module (TPM), which ensure, with different levels of robustness[63] that no component operating outside the protected execution environment created by the VM or TPM can affect (alter or attack) them. Running components in VMs or TPMs also enables their execution to be monitored and their errors and exceptions/failures handled without harming the system as a whole.

6.2. FILTERING OF "DANGEROUS" INPUTS TO/OUTPUTS FROM COMPONENTS

Filtering of the inputs to a component or outputs from a component can be implemented by modifying the component itself (to add input or output validation logic), by applying a reused or custom-built wrapper to the component,[64] or by interposing a commodity filtering application (*e.g.*, firewall/gateway) between the component and other component(s).

Filtering enables the component to review its input or output for acceptability, based on pre-defined parameters, and to either reject (block) or modify ("sanitize") the input/output before sending it on—in the case of output—or storing or acting upon it (in the case of input).

Sanitization is the act of removing or altering some part of the input/output string. Sanitization can be simplistic—*e.g.*, a string that exceeds certain length limitation may simply be truncated; a string that is incorrectly formatted, *i.e.*, includes special characters when only alphanumeric characters are allowed, may simply be overwritten with zeroes. Or it can be more intelligently

altered, *e.g.*, an incorrectly formatted string may have only the non-allowed characters deleted or overwritten.

The criteria by which input or output is deemed unacceptable may be codified in a policy. For example, the policy may specify the acceptable length and format of input or output. Or it may go further to designate "white lists" or "black lists" of text strings that may or may not be allowed to appear in input or output. A white list indicates those strings that are allowed—the presumption being that any string not on the white list should be rejected or sanitized. A black list indicates those strings that are not allowed—the presumption being that any string not on the black list should be allowed. Black listing is less likely to result in the rejection of valid input/output that whoever compiled the white list failed to include on that list. White listing is less likely to result in dangerous inputs/outputs being accepted, because whoever compiled the black list forgot to include those strings on the list.

Use of white lists and black lists can also add unacceptable processing overheads. To mitigate this, developers of systems in which such lists are needed often implement them in combination, usually with strings checked first against a black list, to catch the obviously unacceptable strings, and the remaining reduced list of strings then checked against a white list, to catch any unknown strings.

The mechanism most often used to add filtering and checking logic to components is the *wrapper*, which is software interface code that is bound to the component in a way that interposes the wrapper interface between the wrapped component's built-in interface and the interfaces of other component(s) with which the component interacts. By filtering output from the component or input to the component, wrappers can restrict the ability of external components to request or invoke features in the wrapped component, can wholly block or sanitize (modify) potentially problematic requests or computation results before releasing them. Decisions to filter or sanitize may be based on safety and validity checks implemented by the wrapper logic.

A number of reusable software wrappers are available to perform validation and filtering of input. Java and .NET include filtering APIs that can be used to ensure that filtering functions are adequately performed.

In embedded systems, it may be difficult to add filtering logic or wrappers because of the overhead and added code they impose. However, the increasing prevalence of UNIX/Linux- and Windows-based embedded operating systems is making it less burdensome to add filtering logic to real-time safety-critical systems.

ADDITIONAL SUGGESTED RESOURCES

- Razmov, Valentin, University of Washington and Daniel R. Simon, Microsoft Research, "Practical Automated Filter Generation to Explicitly Enforce Implicit Input Assumptions", in *Proceedings of the 17th Annual Computer Security Applications Conference*, New Orleans, LA, 10-14 December 2001. http://www.acsac.org/2001/papers/53.pdf (accessed 8 April 2010).

6.3. SPECIFIC AND EXPLICIT ERROR AND EXCEPTION HANDLING

Safe, secure software requires significantly more error and exception-handling functionality than program functionality. This error and exception handling must be purpose-built, not generic, to minimize the possibility of faults escalating into failures. Regardless of whether a fault is accidental or intentionally-induced (as the result of an attack or exploit), rather than failing, the software should be able to keep running for as long as possible, if only at a degraded level of operation (*i.e.*, reduced performance, termination of lower-priority functions, rejection of new inputs/connections). For safety-relevant (vs. safety-critical) software, if it must fail, the exception handler and architecture should combine to prevent the failure from placing the software into an unsafe or insecure state. For safety-critical software there is little or no threshold of tolerance for the delays typically involved in post-failure recovery and restoration. In such software, survivability measures must be implemented to prevent failures, full stop.

As noted in Section 2, many of the inconsistencies that can arise in inter-component safety and security occur when one component sends data or commands in error to another component. As such, an important aspect of developing safe and secure component-based systems is in the development and maintenance of robust error handling procedures when implementing the connections between interfaces. Through an understanding of the expected inputs and outputs of various components—the glue code that provides a communication mechanism between multiple components can be written to implement additional error and exception handling logic beyond that provided in the individual components themselves. In these situations, the glue code can provide assurance that the output of one component will meet the expected input of the next component while appropriately modifying data or performing appropriate actions in the presence of unexpected or unallowable data or commands, and for reacting to faults/exceptions in ways that prevent them from escalating into failures (consistent with the fault tolerance requirements of the system).

See Appendix C for information on implementing robust error and exception handling.

6.4. REUSE OF VETTED COMPONENTS AND SOFTWARE LIBRARIES

Reusable vetted libraries components provide organizations with well-tested code that can be used to improve the safety or security of individual components. For example, the Enterprise Security Application Programming Interface (ESAPI)[65] offers well-tested security functionality that can be applied to individual components to improve their overall security framework. Other libraries, like the Spring Security Framework[66] can support aspect-oriented programming, where library functionality can be inserted into the component at run-time. Similar techniques can be applied in the safety field, where common safety-critical functions (*e.g.*, parameter checking) can be offloaded to a trusted aspect-oriented library or external component—precluding the need that a specific software component develop a custom implementation of the required functionality.

6.5. RISK-REDUCING MODIFICATION OF COMPONENTS

While wrapping provides a somewhat limited means of adding logic to a component without modifying that component, there is the potential to make more far reaching alterations to components acquired in source code form (*e.g.*, open source code, code from reuse repositories, legacy source code). Such modifications can have a number of different objectives:

- Correct errors in the component;
- Remove hazardous or vulnerable logic from the component;
- Add logic to the component, *e.g.*, code signature validation, user authentication, input validation and filtering, auditing/event logging, error handling and exception handling, interface protocols;
- Remove dormant (unused) code or dead code.

For some components there may be software development kits (SDKs) available that allow for the development of extensions to change or extend the component's functionality. Unlike wrappers and other external filtering mechanisms, SDK extensions are incorporated into the component itself, thereby modifying the component.

6.6. EXTERNAL SAFETY AND SECURITY CHECKS AND INTERLOCKS[67]

Additional safety-specific and security-specific logic components may added to the system architecture to perform safety checks, including dynamic checks, and implement interlocks to limit the consequences of failures in individual components or the system overall. For example, a software watchdog may be added in combination with dynamic test inputs intended to reveal any software lock-ups.

Increasingly, technology is being deployed that provides this level of control. Within VM-environments, organizations may deploy software external to the VM that monitors the status of processes and applications running within the VM. While this software is currently focused on maintaining a security posture, in many safety-critical environments, this type of information can provide insight into whether or not a safety-critical component is operating as expected.

At a higher level, organizations may leverage application firewalls or other gateway applications that monitor all interactions between components and have the capability to alter those components in a desired fashion. One of the most popular types tools in use are eXtensible Markup Language (XML) gateways, which can perform analyses of XML messages passing through an organization and serve as integration mediums (*e.g.*, enterprise service bus) to convert messages from one format to another to enable improved interoperability between separate components.

Further, many software development frameworks and APIs exist that allow for the integration of wrappers and "filters" to alter, audit and/or verify data that passes between various interfaces. At

the lower levels, organizations may take advantage of object-oriented APIs and extend them to support the required features without altering the internals of an individual component; at a higher level, organizations could leverage features like Java EE's support for "filter" objects through which all interaction pass. When these frameworks are not available, organizations may also work on custom-developing modules that will act as intermediaries between the various systems-taking advantage of network- or OS-level APIs to capture messages between individual components of the system.

6.7. SURVIVABILITY MEASURES

Survivability—the ability of a system to continue operating despite the presence of faults, failures, attacks, *etc.*—of software-based systems is achieved, predominantly, at the architecture level, through a combination of component redundancy and component diversity. Component redundancy is achieved by including multiple copies of the same component in the system, along with a mechanism that monitors the execution of the components to detect any event (*e.g.*, a fault or escalating series of faults) that may necessitate switching the burden of processing from a component that may soon fail to its replica. The number of redundant components in most systems is two: a primary and a "backup". In some systems, however (those with extremely high safety and availability requirements), it may be desirable to implement more than two copies of critical components—keeping in mind that with redundancy also comes complexity, not least in the software that must monitor and act to switch processing from failing to non-failing copies.[68]

Component diversity adds to the complexity, but also the survivability, of the system by not simply implementing exact replicas of the redundant components, but by implementing two components that are identical in functionality but different in their implementation. These components run simultaneously during system operation, under the assumption that an error or fault in one version of the component is unlikely to appear in the other version. Combined with redundancy, this ensures that not only is there an identical copy of a component that will keep running when its "twin" fails, but that the twins are fraternal rather than identical: both capable of the same functionality, but distinct enough to avoid sharing the same defects.

A fairly long-standing technique for achieving component diversity is *n*-version programming,[69] whereby different developers build components to the same specification, with modeling, analysis, and testing to prove that the diverse components are functionally identical to the extent that processing in the system is correct regardless of which component performs any given function.

More recently, in the research community biological and genetic models for diversity have been applied to software, *e.g.*, for implementing software that can autonomically diversify, as well as adapt to continue operating correctly despite the presence of anomalous or dangerous external stimuli (inputs) from its environment. An example of such research is the Genesis project at the University of Virginia.[70]

See Appendix C for further information on survivability measures for software-intensive systems.

ADDITIONAL SUGGESTED RESOURCES

- Guo, Wei, Yanyan Zheng, and Renzuo Xu, Wuhan University, "Survivability of components assembly in architecture", in *Wuhan University Journal of Natural Sciences*, Volume 13, Number 5, October 2008.
- Ellison, Robert J., and Carol Woody, "Survivability Challenges for Systems of Systems", 1 June 2007. http://www.sei.cmu.edu/library/abstracts/news-at-sei/securitymatters200706.cfm (accessed 14 January 2010).
- Ellison, Robert J., Carol Woody, et al., CMU SEI, "Survivability Assurance for System of Systems", Technical Report CMU/SEI-2008-TR-008 / ESC-TR-2008-008, May 2008.

---

[63] TPMs, being hardware-based, have been demonstrated to be more robust than software-based VMs, which may be vulnerable to side-channel attacks and other attacks to which TPMs are either not at all vulnerable or significantly less vulnerable.

[64] Chen, Luping, and John May, "Methods for Enhanced Safety Wrapper Design", in *Proceedings of Gesellschaft für Informatik e.V. Informatik 2004 Informatik verbindet Conference,* Ulm, Germany 20-24 September 2004, pp. 77-82. http://subs.emis.de/LNI/Proceedings/Proceedings50/GI-Proceedings.50-13.pdf (accessed 14 January 2010).

[65] OWASP Tools Project: Enterprise Security API (ESAPI) Web page. http://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API (accessed 14 January 2010).

[66] Spring Security Website. http://static.springsource.org/spring-security/site/ (accessed 14 January 2010).

[67] *Op. cit.*, Bishop, Bloomfield, *et al.*, "Justifying the use".

[68] Diaconescu, Ada, "A Framework for Using Component Redundancy for Self-Adapting and Self-Optimising Component-Based Enterprise Systems", in *Proceedings of the Association for Computing Machinery (ACM) International Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003)*, Anaheim, CA, 26-30 October 2003.

[69] Wikipedia: N-version programming. http://en.wikipedia.org/wiki/N-version_programming (accessed 14 January 2010).

[70] University of Virginia GENESIS Web site. http://www.cs.virginia.edu/genesis/ (accessed 14 January 2010).

7. ASSESSMENT AND TESTING OF COMPONENT-BASED SYSTEMS

As part of determining the overall safety or security level of a component-based system, an organization needs to develop a robust residual safety risk assessment report. Even more important, in light of the increasing interconnectedness of systems both within the DOD and the computing industry as a whole, security must be an important aspect of the safety risk assessment—for if an intentional safety fault can occur, the system is still unsafe. To this end, the organization needs to perform an assessment of the component-based system—particularly when the component-based system is built mostly on COTS products in which the organization has little insight or control over the development of the individual components. Regardless of the development or acquisition process associated with individual components within a system, a system-wide assessment provides organizations with an understanding of the emergent properties—including any emergent risks or hazards—introduced through composition of components.

In many cases, a system-level assessment is the closest an organization can come to matching the operating environment of the safety- or security-critical system prior to deployment. In fact, in some cases, these simulations can be more stringent than those tests performed with the entire system intact (for example, there are limits to the tests that can be performed on avionics software during test flights). Thus, the inclusion of these assessments within a residual safety risk assessment report is can greatly improve the assurance that the system is both safe and secure.

Independent Validation and Verification (IV&V) can be used to increase the level of confidence in a system-level review by relying on an independent and objective third party to perform the system-level safety and security risk assessment. As such, the resulting report is less likely to contain bias while benefitting from the expertise provided by a dedicated testing and analysis team. The use of certified reviewers allows for assurance that the team conducting the IV&V has met appropriate requirements as specified by the IV&V contract, allowing for the organization to ensure that the quality of reviewers within the IV&V team is as good as or better than an equivalent in-house team.

Independent Safety Assessments (ISAs) and Independent Safety Reviews (ISRs) are often required by mandate, for example see SECNAV Instruction 5100.10J and Army Regulations 70-1 and 385-16, FAA Order 1100.161. Similarly, independent security tests and evaluations, such as Common Criteria Evaluations and Security Test and Evaluation (ST&E) within systm Certification and Accreditation (C&A) regimes, are also required by mandate. Moreover, some government organizations such as NASA, DOD, and NIST have established their own software IV&V facilities for safety, security, and/or reliability testing.

While IV&V can serve as an important component in generating a residual risk assessment report, it is important to note that those performing the IV&V will not have as thorough an understanding of the system itself as an internal review team would otherwise have. As such, it may be necessary to augment IV&V with an internal review, allowing internal teams to

specifically test potentially known weak-spots and include those results in the overall risk assessment report.

ADDITIONAL SUGGESTED RESOURCES

- Voas, Jeffrey, "Testing Software for Characteristics Other than Correctness: Safety, Failure-tolerance, and Security", in *Proceedings of the 16th International Conference and Exposition on Testing Computer Software*, Bellevue, WA, March 1996. http://www.cigital.com/papers/download/ictcs96.pdf (accessed 31 March 2010).
- Wallin, Christina, ABB Corporate Research, "Verification and Validation of Software Components and Component Based Software Systems", in Chapter 5: "Component-Based Development Process", of Ivica Crnkovic and Magnus Larsson (editors), *Building Reliable Component-Based Systems* (Norwood, MA: Artech House, 2002). http://www.idt.mdh.se/cbse-book/extended-reports/05_Extended_Report.pdf (accessed 8 April 2010).

7.1. SAFETY ASSESSMENT AND TESTING

As identified by the Department of Transportation,[71] the same suite of tests performed on individual components may be performed on the system as a whole as an important aspect of a system-wide safety assessment. These tests include both system-level functional testing and system-level fault injection. In conjunction, organizations should aim to inject faults into the individual components as well as into the system as a whole, thereby introducing any system-level faults that may result from tests that would have been flagged by performing fault injection on the component itself.

In addition, organizations should continue to perform the safety verification and validation activities that have traditionally been included in the system development life cycle. However, where possible, safety verification of component requirements, design, coding, module testing, and integration testing should be included for individual components as well as for the system as a whole. In the case where COTS components are included in the system, "a rigorous component verification process must be adopted for all COTS components to attain a portion of the confidence level provided by the missing verification activities during requirements specification, software design, and software coding."[72]

Safety verification can be supported by numerous tools and techniques discussed in Section 5.1, including:[73]

- **Hazard Analysis:** Identifying potential hazards facing the system by considering the individual components of the system and its operating environment.
- **Cause Consequence Analysis:** Identifying the possible consequences associated with chosen events within the system.

- **Fault Tree Analysis:** Identifying the individual faults that lead to a specific safety hazard, allowing developers to calculate a probability that the safety hazard will occur.

Tools are available to perform fault injection against individual components or the system as a whole. In general, these tools will aid in performing fuzz tests against the system as well. In some cases, organizations may require analyzing the source code of COTS products and, where legally applicable, organizations may take advantage of a number of safety-oriented decompilers and binary analysis tools to gain some insight into the safety of individual components' code bases.

By performing safety assessments at both the component and system level, including equivalent rigorous verification processes for COTS components, the developer can develop robust documentation describing both how and why the component-based system satisfies its safety requirements.

ADDITIONAL SUGGESTED RESOURCES

- Skramstad, Torbjørn, Norwegian University of Science and Technology, "Assessment of Safety Critical Systems with COTS Software and Software of Uncertain Pedigree (SOUP)", in *Proceedings of the First European Research Consortium for Informatics and Mathematics Workshop on Software-Intensive Dependable Embedded Systems*, Porto, Portugal, 30 August-1 September 2005. http://www.itk.ntnu.no/misc/ercim/05/final_draft/skramstad_2005_Euromicro_ERCIM_WS_Camera_Ready.pdf (accessed 31 March 2010). Also: http://www.ercim.eu/publication/ws-proceedings/EMB-SYS2005.pdf (accessed 31 March 2010).
- Ericson, II, Clifton A., *Hazard Analysis Techniques for System Safety* (Hoboken, NJ: John Wiley & Sons, Inc., 2005).
- Wysocki, Joseph, HRL Laboratories, and Rami Debouk, General Motors, "Methodology for assessing safety-critical systems", in *International Journal of Modelling and Simulation*, Volume 27 Issue 2, March 2007, Pages 99-106.
- Lauritsen, Torgrim, "Safety analysis of Object-Oriented software", 2005, http://www.idi.ntnu.no/emner/dt8100/Essay2005/lauritsen.pdf (accessed 8 April 2010).
- Brosgol, Ph.D., Benjamin M., "Safety and Security: An Analysis of Certification Issues and Technologies", presented at 20th Systems and Software Technology Conference, Las Vegas, NV, 29 April-02 May 2008. http://sstc-online.org/2008/pdfs/BB2030.pdf (accessed 8 April 2010).

7.2. SECURITY ASSESSMENT AND TESTING

Much like safety assessments, the majority of security testing activities can be performed both at the component and system level. In particular, all security tests that do not rely on access to the

source code can be performed on COTS components. Where possible, organizations can also include information about the security practices used by the developer of the COTS components (for example, use of a well-documented security-enhanced development life cycle may be of interest when assessing the security of the COTS component).

When performing a security assessment of component-based systems, organization can leverage their usual techniques, applying them both to the individual components of the system as well as the system as a whole. In some cases, resources may be better spent by focusing on a specific component or on the system as a whole (for example, penetration testing resources may be better spent on the system as a whole rather than on each individual component). Other assessment techniques, including the use of automated tools can be performed throughout the security assessment process. Nevertheless, it is important that the security assessment process be a parallel of the safety assessment process, with the following security verifications being performed on requirements, design, coding, module testing and integration testing.

Security verification can be supported by a wide variety of tools and techniques, including:[74]

- **Manual Code Review (static analysis):** Identifying potential vulnerabilities by performing a manual review of security-critical portions of code (when available).
- **White Box Testing (static and dynamic analyses):** Identifying potential vulnerabilities by analyzing the source code for potential security defects or known-insecure libraries and code structures. In static analysis, only the source code is analyzed. In dynamic analysis, the source code is compiled and executed, with execution events traced to the original source code, to help determine the segments of the source code responsible for observed anomalies, faults, *etc.*
- **Black Box Testing:** Identifying potential vulnerabilities by performing dynamic analysis of the executing binary component/system and/or static analysis of the binary executable code.
- **Penetration Testing:** Identifying potential vulnerabilities by subjecting the system (or individual components) to attacks from a skilled professional.
- **System-level Analysis:** Identifying potential vulnerabilities at the design- or system-level of the application by performing a thorough review of system and component architecture and design documents.

By performing security assessments at both the component and system level, including equivalent processes for COTS components, the developer can develop robust documentation (*e.g.*, in the form of a test report) describing both how and why the component-based system meets its security requirements.

ADDITIONAL SUGGESTED RESOURCES

- Khan, Khaled M., University of Western Sydney (Sydney, Australia), and Jun Han, Swinburne University of Technology (Melbourne, Australia), “Deriving Systems Level

Security Properties of Component Based Composite Systems", in *Proceedings of the 2005 Australian Software Engineering Conference*, Brisbane, Australia, 29 March-1 April 2005. http://www.ict.swin.edu.au/personal/jhan/jhanPapers/aswec05KK-JH.pdf (accessed 29 March 2010).

- Wysopal, Chris, Lucas Nelson, Elfriede Dustin, and Dino Dai Zovi, *The Art of Software Security Testing: Identifying Software Security Flaws* (Boston, MA: Addison-Wesley Professional, 2006).
- Krsul, Ivan Victor, *Software Vulnerability Analysis*, PhD thesis, Purdue University, May 1998. http://ftp.cerias.purdue.edu/pub/papers/ivan-krsul/krsul-phd-thesis.pdf (accessed 29 March 2010).
- Dowd, Mark, John McDonald, and Justin Schuh, *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities* (Boston, MA: Addison-Wesley Professional, 2006).
- Ashbaugh, Douglas A., *Security Software Development: Assessing and Managing Security Risks* (Boca Raton, FL: Auerbach Publications, 2009).

[71] *Op. cit.*, Clough, Anne, *Train Control Applications*.

[72] Ibid, page 20

[73] *Op. cit.* Brömme, "Hazard Analysis of IT Systems".

[74] *Ibid.*

8. MAINTENANCE CONSIDERATIONS

The safety and security concerns described here can arise during maintenance of any safety-/security-critical system, but are particularly relevant for component-based systems.[75]

Either due to changes in system requirements or underlying technologies, or changes in the capabilities of or safety/security properties in later versions of NDI components, it is likely that at least some components will need to be replaced over the lifetime of the component-based system. In some cases, at the time the system was initially assembled, only partly-adequate components may have been available to fulfill certain roles. In other cases, after the system is deployed, a failure, mishap, or vulnerability exploit may occur that reveals a faults or vulnerabilities in certain components that were not caught during pre-deployment testing.

Even when a system failure can be isolated to a fault or exploited vulnerability in a specific NDI component, it may be extremely difficult to induce the component's developer (or, in the case of COTS, vendor) to acknowledge and fix the component to eliminate the problem. Even if a vendor does agree to make the change, the normal release schedule for COTS products may delay the fix for an unacceptable amount of time. In such cases, it may be necessary to substitute a functionally interchangeable but safer/more secure component, or to implement a mitigation such as a wrapper or execution constraint, at least until a more robust version of the flawed component becomes available.

For this reason, throughout the system's lifetime, the marketplace should be continuously monitored for current and emerging products and technologies. Both system development and certification/acceptance/accreditation processes need to be as flexible as possible, to allow for the substitution of functionally interchangeable but safer/more secure components, or addition of new components to satisfy new requirements.

It is imperative that immediately upon discovery of a fault or vulnerability that has the potential to cause a system failure every effort is taken to mitigate the increased risk through short-term countermeasures until the vendor releases a security patch/fix, or until other more permanent countermeasures and safeguards can be applied to isolate or replace the flawed component. Some effective short-term countermeasures, mainly operational, include strict auditing and monitoring of the application's usage and input and output data streams (with appropriate administrator response to detected potential mishaps or exploits), reconfiguration of the execution environment access control-type restraints to more tightly limit the exposure of the component, and/or the potential damage and impact of a fault or exploit in that component.

---

[75] *Op. cit.*, DISA, *Developer's Guide*.

9. RISK MITIGATION STRATEGY ROADMAP

All of the architectural engineering controls and countermeasures discussed in Section 6 of this document can help mitigate safety and safety-impacting security risks that arise in component-based software-intensive systems, there is a minimum "core set" that should be undertaken for every safety-critical system. Table 9-1 maps the mitigations in this "core set" to the phases of the system life cycle in which they should be implemented.

Table 9-1. Risk Mitigation Strategy Roadmap

Life Cycle Phase	Mitigation	Details
Acquisition	Avoiding SOUP Components.	Limiting acquisition and use of software of unknown pedigree/provenance (SOUP) to only system functions that are neither "trusted" nor exposed to direct access. Never use SOUP components to implement critical/trusted functions or functions at the system boundary (i.e., functions that can be easily found and targeted by an attacker).
Architecture	Architectural Risk Assessment.	This includes Safety Hazard Analysis and Security Risk Analysis, as basis for adjusting system software architecture to include necessary risk mitigation controls and countermeasures
	Constraints on Execution Environment.	Use of VMs, TPMs, wrappers, filters, etc., to reduce exposure of known-vulnerable and otherwise untrustworthy components, and to isolate such components from trustworthy components (and vice versa)
	Blocking Access to Unused Functions.	Use execution environment constraints, wrappers, filters, etc., to minimize external access to undocumented features, unused functions, dormant/dead code
Assembly	COTS SDKs.	Use vendor-provided SDKs whenever possible to extend COTS component functionality (e.g., to enable signature validation, add input validation logic or explicit error/exception handling logic, etc.), or to modify component functionality (e.g., to disable unused functions).
	Checksum Monitors.	Implement these monitors to observe system operation for evidence of system and data tampering.
	Code and Data Digital Signatures.	Digitally sign known-trustworthy code components and critical/sensitive data, to provide proof of authenticity and indications of tampering. This will also require implementation of signature validation logic in those other components that interact with the signed components/consume the signed data; such logic should prevent execution of components and use of data whose signatures cannot be validated.

	Specific, Explicit Error and Exception Handling Logic.	Addition of explicit error and exception handling logic (if necessary as wrappers) to prevent undefined execution of unused functions within individual components

10. CONCLUSION

The safe and secure operation of component-based systems relies in large part on the ability to trust the system as a whole, and all of its critical components (both NDI and custom) as well as the architecture according to which those components are assembled. As long ago as 1988, in its report on the Strategic Defense Initiative's Ballistic Missile Defense (BMD) technology,[76] the U.S. Congress Office of Technology Assessment made several observations that still hold true today for software components in complex, software-intensive weapons systems (especially COTS/NDI components). Some of the key points from that report should be considered before undertaking the assembly of components to produce a reliable, safe, survivable weapons system:

1. The nature of software and experience with large, complex software systems indicate that there will likely always be some irresolvable questions about how dependable a highly complex software-intensive weapon system assembled from components can be.

2. Existing large software systems, such as the long-distance telephone system, have become highly dependable only after extensive operational use and modification during which period errors that cause failures can be found and corrected.

3. Given the current state of the art in software engineering, complex systems cannot be trusted to be free of catastrophic failures before a period of extensive use.

4. A complex software-intensive weapon system assembled from components should probably not be trusted to work properly the first time due to the probable presence of errors that could cause catastrophic failures.

5. It is difficult to judge whether or not software, and particularly NDI software, is dependable and trustworthy. If evaluations of dependability and trustworthiness rely on the results of simulations of the weapon system's use in battle, a second critical question whether the battle simulations could be modeled closely and accurately enough to reveal all potential catastrophic system failures, and more importantly to enable those failures to be traced back to their root causes in individual software components or inter-component interfaces.

Attributes that most affect the ability to trust COTS/NDI components include the safety and security of the components' behaviors and interfaces, as observed during analysis and testing, and pedigree and provenance of those components (*i.e.*, origin of the component, how the component has been handled throughout its life cycle, reputation and track record of its developers).

Unless the integrator of the weapons system software discovers as many of these attributes as possible about each component to be used, he/she will be assuming unacceptable level of risk. Components about which such information cannot be discovered must be considered *un*trustworthy, and system safety and security engineering techniques must be applied to mitigate the hazards and risks posed by allowing the component to interoperate with other components in

the system, and allowing the component to be accessed by entities external to the system (human users, other systems).

In the integration of the system, rules—or contracts—governing the component's interactions with all other components need to be defined. These contracts will specify all inputs and services each component expects to receive from others, and all outputs and services the component will be expected to provide to others. Hazard analysis and risk analysis need to be undertaken to determine whether those expectations can be satisfied safely and securely, given the inability to trust one or more of the components to which the contract pertains.

For any interaction that cannot be assured to occur safely or securely, safety/security controls and countermeasures need to be interposed between the two components that are parties to the contract governing that interaction, to mitigate or eliminate (the latter may be required for safety, for which "room for error" is often less tolerable than it is for security) the safety hazards and security risks. These controls will either prevent the interaction from occurring (*e.g.*, virtual machines and TPMs are two controls that can be used for this purpose), or will change the nature of the interaction by altering the output provided by one component as input to the next (*e.g.*, filtering and input validation wrappers may be used to provide this kind of control). In addition, architectural-level mitigations may also be deployed (*e.g.*, using a minimized OS).

System developers and integrators should be reluctant to trust any component about which insufficient information about can be discovered about its properties. This information should be discovered primarily through direct analysis and testing of the component, in isolation and in combination with other components, using techniques described in this paper.

Another source of information is "indirect evidence", such as third-party reviews and analysis results, Common Criteria Security Targets, and other formal evaluation or certification documentation, developer/vendor claims, and knowledge of the component's pedigree and provenance, including its development process. However, a component should never be trusted based solely on indirect evidence, no matter how plentiful. While indirect evidence can be useful for helping flesh out the discoveries made during direct component analysis/testing, at a minimum some level of black box testing (if no source code is available) should always be performed to validate the claims stated or implied by the indirect evidence.

Further complicating matters is the inability for those analyzing and testing components to fully predict whether the safety and security properties those components exhibit in isolation will remain evident once those components interact with others in the system within a typical operational context, and whether they will further remain safe and secure when the system is exposed to hazards or security threats.

At the whole system level, then, the trustworthiness of components must be determined based not only on the component's direct interactions with other components, but on the component's observed behaviors in response to direct inputs and also "chain reaction" events. The best way to achieve this visibility into component behaviors is through component analysis and testing within an architectural framework. This framework should simulate, as closely as possible, the actual system architecture into which the component will be integrated.

Analysis of each component's behavior as it executes within the architectural framework will provide knowledge to the analyst/tester that single-component and pair-wise-component testing cannot. Such testing will enable the analyst/tester to observe all components simultaneously as they interact. The tester can observe components as they interact under normal conditions, and through instrumentation of inputs and sensor stimuli and also "tampering" with the system—*e.g.*, to remove a power source, disconnect a network cable, launch a denial of service attack pattern, *etc.*, through simulated safety mishaps or security attacks. Unfortunately, few systems can be built entirely from known-trustworthy (and thus justifiably trusted) components. Before accepting untrustworthy (and thus untrusted) components, however, a trade-off analysis should be performed that determines whether the perceived benefits, in terms of minimized development effort and time, do indeed outweigh the safety hazards and security risks introduced by the untrustworthy components: not just in terms of initial development cost, but whole lifecycle cost—through system de-commissioning and disposal. This trade-off analysis needs to consider not only development costs but costs associated with maintenance, patching, updating, change control, ongoing hazard and risk analyses, *etc.*

It is critical that the tradeoff analysis factor in not only the lifecycle costs of the untrustworthy component itself, but of all controls/countermeasures needed to mitigate the hazards and risks the component poses to the system. It may turn out that what appears to be a savings from use of an untrustworthy component is, in fact, only a savings in terms of the initial development cost: the component could well represent a loss in terms of its overall lifecycle cost.

ADDITIONAL SUGGESTED RESOURCES

- Wood, Bill J., "Software Risk Management for Medical Devices", in *Medical Device and Diagnostic Industry*, January 1999. http://www.bestechconsulting.biz/Software%20Risk%20Management%20for%20Medical%20Devices.pdf (accessed 29 March 2010).

[76] *Op. cit.*, Johns, Sharfman, *et al. SDI*.

APPENDIX A. ABBREVIATIONS, ACRONYMS, AND GLOSSARY

Section A.1 provides amplifications of the abbreviations and acronyms used in this document. Section A.2 is a glossary that defines terms used throughout this document which were not defined in Section 2.

A.1. ABBREVIATIONS AND ACRONYMS

AFMSTT	Air Force Modeling and Simulation Training Toolkit
ALARP	As Low As Reasonably Practical
API	Application Programming Interface
ARL	Army Research Laboratory
BMD	Ballistic Missile Defense
CNAD	Conference of National Armaments Directors
CMU	Carnegie Melon University
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off-The-Shelf
DACS	Data & Analysis Center for Software
DCOM	Distributed Component Object Model
DERA	Defence Evaluation and Research Agency
DHS	Department of Homeland Security
DOD	Department of Defense
DOE	Department of Energy
ESAPI	Enterprise Security Application Programming Interface
FAA	Federal Aviation Administration
FHA	Fault Hazard Analysis
FMEA	Failure Modes and Effects Analysis
FMECA	Failure Mode, Effects and Criticality Analysis
GUI	Graphical User Interface
IDL	Interface Description Language
IEC	International Electrotechnical Commission
IEEE	Institute for Electrical and Electronics Engineers
IFIP	International Federation for Information Processing
iFMECA	Interface FMECA
INCOSE	International Council on Systems Engineering

ISA	Independent Safety Assessment
ISO	International Organization for Standards
IV&V	Independent Validation and Verification
Java EE	Java Enterprise Edition
M&S	Modeling and Simulation
MHA	Maintenance Hazard Analysis
MIT	Massachusetts Institute of Technology
NASA	National Aeronautics and Space Administration
NATO	North Atlantic Treaty Organization
NAVSEA	Naval Sea Systems Command
NDI	Non-Developmental Item
NIST	National Institute of Standards and Technology
NOSSA	Naval Ordinance Safety and Security Activity
OHA	Operating Hazard Analysis
OS	Operating System
PHA	Preliminary Hazard Analysis
SCI	System Concepts and Integration
SDK	Software Development Kit
SHA	Software Hazard Analysis
SHA	System Hazard Analysis
SOUP	Software of Unknown Pedigree/Provenance
SSHA	Subsystem Hazard Analysis
SQL	Structured Query Language
TPM	Trusted Platform Module
VM	Virtual Machine
XML	eXtensible Markup Language

A.2. GLOSSARY

Legend of Sources for Definitions

Reference	Document
DAU	Defense Acquisition University, *Glossary of Defense Acquisition Acronyms and Terms,* 13th Edition, November 2009[77]
Risk	DAU, *Risk Management Guide for DOD Acquisitions*, Sixth Edition, August 2006.[78]

JCIDS	Chairman of the Joint Chiefs of Staff, *Manual for the Operation of the Joint Capabilities Integration and Development System*, 31 July 2009[79]
FAR	*Federal Acquisition Regulation* Subpart 2.101[80]
6212.01E	CJCS Instruction 6212.01E, I*nteroperability and Supportability of Information Technology and National Security Systems*, December 2008[81]
882D	DOD, *Department of Defense Standard Practice for System Safety*, MIL-STD-882D, 10 February 2000[82]
498	DOD, *Software Development and Documentation*, MIL-STD-498, 5 December 1994[83]
Maier	Maier, Mark W., The Aerospace Corporation, "Architecting Principles for Systems-of-Systems"[84]

Anti-Tamper (AT). Systems engineering activities intended to prevent and/or delay the exploitation of critical or sensitive technologies in U.S. systems by deterring efforts of an adversary or other individual to reverse-engineer, exploit, or develop countermeasures against a system or system component. [DAU]

Architecture. The structure of components, their interrelationships, and the principal guidelines governing their design and evolution over time. [JCIDS] (2) The organizational structure of a system that identifies its components, their interfaces, and the concept of execution among them. [498]

Availability. A measure of the degree to which an item is in an operable state and can be committed at the start of a mission when the mission is called for at an unknown (random) point in time. [DAU]

Commercial Off-The-Shelf (COTS). Any commercial item (excluding include bulk cargo items such as agricultural products and petroleum) sold in substantial quantities in the commercial marketplace and offered to the government under a contract or subcontract at any tier, without modification, in the same form in which it was sold in the marketplace. [FAR]

Configuration. A collection of an item's descriptive and governing characteristics, which can be expressed in functional terms, *i.e.*, what performance the item is expected to achieve; and in physical terms, *i.e.*, what the item should look like and consist of when it is built.

Embedded Software. Software physically incorporated into a physical system, device, or piece of equipment whose function is not purely data processing, or external but integral to that system/device/equipment. The functionality provided by embedded software is usually calculation in support of sensing, monitoring, or control of some physical element of the equipment, device, or system, *e.g.*, mathematical calculations of distances used in calibrating the targeting mechanisms of a weapon system; interpretation and comparison of heat sensor readings in a nuclear-powered submarine engine against a set of safe temperature thresholds.

Failure. An event in which any part of an item does not perform in conformance with its specification. A software failure is the inability, resulting from a fault in the software, to perform an intended logical operation as specified in the presence of the specified data or environment. [DAU]

Hazard. A condition that is a prerequisite to a mishap. [882]

Independent Verification and Validation (IV&V). Systematic evaluation (through review, assessment, and/or testing) of a product by an organization or individual that is not responsible for or involved in developing that product. [498 adapted]

Interchangeability. A condition that exists when two or more components possess capabilities and properties such that the components are equivalent in functionality, performance, and dependability. This equivalence enables the components to be exchanged one for the other without alteration to the components themselves or to any adjacent/adjoining components, except through reconfiguration or wrapping. [DAU adapted]

Interface. The functional, physical, and logical characteristics required at a common boundary or connection between components, systems, and/or people to enable those components, systems, and/or people to communicate, collaborate, exchange data and instructions, *etc.* [DAU]

Interoperability. The ability of systems, components, or units to provide data and services to and accept the same from other systems, components, or units, and to use the data and services so exchanged to enable them to operate effectively together. [6212.01E adapted]

Mishap. An unplanned event or series of events resulting in death, injury, occupational illness, or damage to or loss of equipment or property, or damage to the environment. An accident. [882]

Model. A representation of an actual or conceptual system that involves mathematics, logical expressions, or computer simulations that can be used to predict how the system might perform or survive under various conditions or in a range of hostile environments. [DAU]

Module. An independently-compilable unit of software made up of one or more procedures or routines or a combination of procedures and routines. [DAU]

Non-Developmental Item (NDI). Any previously developed item that requires only minor modifications or modifications of the type customarily available in the commercial marketplace in order to meet the requirements of the acquirer. [DAU]

Non-Developmental Software. A software product developed for either general use or use across multiple projects, instances, or uses, or developed for a specific project, instance, or use but reused for a different project/instance/use. Examples include, but are not limited to: COTS software, open source software, software in reuse libraries, or any other pre-existing software. See also Non-Developmental Item (NDI), Commercial Off-The-Shelf (COTS). [498 adapted]

Probability. The aggregate likelihood of the occurrence of a set of related or unrelated events. [882D adapted]

Program. A major, independent part of a software system. [DAU]. A combination of computer instructions and data definitions that enables computer hardware to perform computational or control functions. [498]

Dependability. The ability of a system and its parts to perform its mission without failure, degradation, or demand on the support system under a prescribed set of conditions. Software reliability is the probability that software will not cause a failure of a system for a specified time under specified conditions. Also referred to as Reliability. [DAU]

Design. Those characteristics of a system that are selected by the developer in response to the requirements. Some will directly satisfy the requirements; others will be elaborations of requirements, such as definitions of all error messages; others will be implementation-related, such as decisions, about what software units and logic to use to satisfy the requirements. [498 adapted]

Reuse. The use of NDI(s) to develop or update a system. [DAU adapted]

Risk. (1) A measure of future uncertainties in achieving system performance goals and objectives within defined functional and performance constraints. Risk can be associated with all aspects of a system and its environment (*e.g.*, threat, technology, maturity, supplier capability, design maturation, performance against plan). Risks have three components: (i) A future root cause (hazard or threat), which, if eliminated or corrected, would prevent a potential consequence from occurring, (ii) the probability (or likelihood), assessed at the present time, of that future root cause/threat being realized, and (iii) the anticipated consequence (or effect) of that root cause/threat being realized. [Risk] (2) An expression of the possibility/impact of a mishap or attack/exploit in terms of hazard or compromise severity and hazard or compromise probability. [882D adapted]

Risk Assessment. A comprehensive evaluation of risk and its associated impact. [882D]

Risk Management. The overarching process that encompasses identification, analysis, mitigation planning, implementation of mitigation(s), and tracking of future root causes/threats and their consequences. [Risk]

Safety-Critical. Refers to a condition, event, operation, process, or item of whose proper recognition, control, performance or tolerance is essential to safe system operation or use, *e.g.*, safety-critical function, safety critical path, safety critical component. [882D]

Safety-Critical Software Components. Those software components and units whose errors can result in a potential hazard, or loss of predictability or control of a system. [882D]

Severity. A measure of the worst possible consequence that could result from an event. [882D adapted]

Simulation. A method for implementing a model. It is the process of conducting experiments with a model for the purpose of understanding the behavior of the system modeled under selected conditions or of evaluating various strategies for the operation of the system within the limits imposed by developmental or operational criteria. Simulation may include the use of analog or digital devices, laboratory models, or "test bed" sites. [DAU]

Software System. A system consisting solely of software and possibly the computer equipment on which the software resides and operates. [498]

Subsystem. A component of a system that in itself may constitute a system. In a system of systems, a subsystem is always a system [882D adapted]

Survivability. The ability of a system and its crew to avoid or withstand a manmade hostile environment without suffering an abortive impairment of its ability to accomplish its designated mission.

System. The combination of two or more hardware, software, and/or data components in such as way that they can interoperate to satisfy one or more functional requirements. [DAU] (2) A composite, at any level of complexity, of personnel, procedures, materials, tools, equipment, facilities, and software. The elements of this composite entity are used together in the intended operational or support environment to perform a given task or achieve a specific purpose, support, or mission requirement. [882D]

System of Systems (SoS). (1) Two or more independent and useful systems integrated into a larger "super system" that delivers unique capabilities. [JCIDS] (2) A large widespread collection or network of systems functioning together to achieve a common purpose. [Maier] In such an arrangement, each system within a SoS can be seen as a *component* of that SoS.

System Safety. The application of engineering and management principles, criteria, and techniques to optimize all aspects of safety within the constraints of operational effectiveness, time, and cost throughout all phases of the system life cycle. [882D]

Trade-off. Selection among alternatives with the intent of obtaining the optimal, achievable system configuration.

Vulnerability. (1) An inherent weakness(s) in a component or system that can be exploited to effect an attack. (2) The characteristics of a component or system that cause it to suffer a definite degradation (loss or reduction of capability to perform the designated mission) as a result of having been subjected to a certain (defined) level of effects in an unnatural (man-made) hostile environment. [DAU adapted]

---

[77] Download from: https://acc.dau.mil/GetAttachment.aspx?id=17650&pname=file&aid=48053 (accessed 27 March 2010).

[78] Download from: http://www.dau.mil/pubs/gdbks/docs/RMG%206Ed%20Aug06.pdf, (accessed 30 March 2010).

[79] Download from: https://acc.dau.mil/communitybrowser.aspx?id=267116 (accessed 30 March 2010).

[80] Download from: http://farsite.hill.af.mil/reghtml/regs/far2afmcfars/fardfars/far/02.htm#P10_633 (accessed 30 March 2010).

[81] Download from: http://www.dtic.mil/cjcs_directives/cdata/unlimit/6212_01.pdf (accessed 30 March 2010).

[82] Download from: http://www.safetycenter.navy.mil/instructions/osh/milstd882d.pdf (accessed 30 March 2010).

[83] Download from: http://wwwedms.redstone.army.mil/edrd/ref498/498std.pdf (accessed 30 March 2010).

[84] Download from: http://www.infoed.com/Open/PAPERS/systems.htm (accessed 31 March 2010).

APPENDIX B. BIBLIOGRAPHY

The following information resources were consulted in the development of this document.

Assistant Secretary of the Navy for Research, Development, and Acquisition (ASN[RD&A]) Software Acquisition Management Focus Team, *Software Acquisition Management "As-ls State" Report*, Version 2.0, 17 April 2007. https://acquisition.navy.mil/rda/content/download/4631/20848/version/4/file/AS+Is+REPORT+011.pdf (accessed 24 March 2010).

Bachmann, Felix, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau, Carnegie Mellon University Software Engineering Institute, *Volume II: Technical Concepts of Component-Based Software Engineering*, 2nd Edition, Technical Report CMU/SEI-2000-TR-008 | ESC-TR-2000-007, May 2000. http://www.sei.cmu.edu/reports/00tr008.pdf (accessed 7 March 2010).

Balestrini-Robinson, S., J.M. Zentner, and T.R. Ender, Georgia Tech Research Institute, "On Modeling and Simulation Methods for Capturing Emergent Behaviors for Systems of Systems", presented at the National Defense Industrial Association 12th Annual Systems Engineering Conference, San Diego, CA, 26-29 October 2009. http://www.dtic.mil/ndia/2009systemengr/9041WednesdayTrack6Zentner.pdf (accessed 31 March 2010). Also: http://www.acq.osd.mil/sse/webinars/2010-03-09-SoSECIE-M&S-Emergent-Behaviors-Zentner-etal-brief.pdf (accessed 31 March 2010).

Barnum, Sean, and Michael Gegick, "Securing the Weakest Link", 19 September 2005. https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/principles/356-BSI.html (accessed 14 January 2010).

Bishop, Peter, Robin Bloomfield, and Peter Froome, Adelard, "Justifying the use of software of uncertain pedigree (SOUP) in safety-related applications", Health and Safety Executive (United Kingdom) Contract Research Report 336/2001, 2001. http://www.hse.gov.uk/research/crr_pdf/2001/crr01336.pdf; also: http://www.sipi61508.com/ciks/bishop1.pdf

Brömme, Arslan, "Hazard Analysis of IT Systems", presented at the GI FB Sicherheit Schutz und Zuverlässigkeit Workshop on Joint Terminology for Safety and Security, Frankfurt a.M., Germany, 12 July 2002. http://www11.informatik.uni-erlangen.de/Kooperationen/Veranstaltungen/Sicherheit1/PositionPapers/Broemme.pdf (accessed 14 January 2010).

Broy, Manfred, Technical University, Anton Deimel, SAP AG, Juergen Henn, IBM, Kai Koskimies, Nokia Research, František Plášil, Charles University, Gustav Pomberger, University of Linz, Wolfgang Pree, University of Constance, Michael Stal, Siemens AG, and Clemens Szyperski, Queensland University of Technology, "What characterizes a (software) component?", in *Software—Concepts and Tools*, Volume 19 Number 1, June 1998, pages 49-56. http://www.exciton.cs.rice.edu/comp410/frameworks/Pree/J010.pdf (accessed 12 March 2010).

Broy, Manfred, Technische Universität München, "Towards a Mathematical Concept of a Component and Its Use", keynote speech at the 1996 Components Users' Conference, Munich, Germany, 15-19 July 1996. http://www4.informatik.tu-muenchen.de/papers/Broy_CUC1996_klein_1996_Publication.html (accessed 12 March 2010).

Chen, Luping, and John May, "Methods for Enhanced Safety Wrapper Design", in *Proceedings of Gesellschaft für Informatik e.V. Informatik 2004 Informatik verbindet Conference,* Ulm, Germany 20-24 September 2004, pp. 77-82. http://subs.emis.de/LNI/Proceedings/Proceedings50/GI-Proceedings.50-13.pdf (accessed 14 January 2010).

Clough, Anne, Charles Stark Draper Laboratory, for U.S. Department of Transportation Federal Railroad Administration, *Commercial-Off-The-Shelf (COTS) Hardware and Software for Train Control Applications: System Safety Considerations*, Final Report DOT/FRA/ORD-03/14, April 2003. http://www.fra.dot.gov/downloads/Research/ord0314.pdf (accessed 12 March 2010).

Coffman, Jr., E.G., *et al.*, "System Deadlocks", in *Computing Surveys*, Volume 3, Number. 2, June 1971, pp. 67-78.

Committee for National Security Systems (CNSS) Instruction Number 4009, *National Information Assurance Glossary*, June 2006. http://www.cnss.gov/Assets/pdf/cnssi_4009.pdf (accessed 9 April 2010).

De Panfilis, Stefano, SINTEF, and Arne J. Berre, Engineering Ingegneria Informatica S.p.A., Open issues and concerns in Component Based Software Engineering. http://research.microsoft.com/~cszypers/events/wcop2004/20%20Panfilis%20Berre.pdf (accessed 29 March 2010). Also

Defense Acquisition University (DAU), *Glossary of Defense Acquisition Acronyms and Terms,* 13th Edition, November 2009. https://acc.dau.mil/GetAttachment.aspx?id=17650&pname=file&aid=48053 (accessed 27 March 2010). Also: https://acc.dau.mil/CommunityBrowser.aspx?id=17650 (accessed 31 March 2010).

Defense Information Systems Agency (DISA) Applications Security Project, *Developer's Guide to Secure Use of Software Components*, Draft Version 3.0, 7 October 2004.

Denney, Ewen, and Bernd Fischer, NASA, Generating Code Review Documentation for Auto-Generated Mission-Critical Software, in *Proceedings of the Third IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT),* Pasadena, CA, 19-23 July 2009. http://ti.arc.nasa.gov/m/pub/580/SMC-IT-Denney.pdf (accessed 14 January 2010).

Department of the Army, *Failure Modes, Effects and Criticality Analysis (FMECA) for Command, Control, Communications, Computer, Intelligence, Surveillance, and Reconnaissance (C4ISR) Facilities*, Technical Manual TM 5-698-4, 29 September 2006. http://www.army.mil/USAPA/eng/DR_pubs/dr_a/pdf/tm5_698_4.pdf (accessed 31 March 2010).

Diaconescu, Ada, "A Framework for Using Component Redundancy for Self-Adapting and Self-Optimising Component-Based Enterprise Systems", in *Proceedings of the Association for Computing Machinery (ACM) International Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003)*, Anaheim, CA, 26-30 October 2003.

DOD, *Military Standard Procedures for Performing a Failure Mode, Effects, and Criticality Analysis*, MIL-STD-1629A (Cancelled), 24 November 1980. http://www.fmeainfocentre.com/handbooks/milstd1629.pdf (accessed 31 March 2010). Also MIL-STD-1629A Notice 1, 7 June 1983. http://www.sre.org/pubs/Mil-Std-1629A(1).pdf (accessed 31 March 2010).

Ellims, Michael, "Is Security Necessary for Safety?", presented at 4th Embedded Security in Cars Conference, Berlin, Germany, 14-15 November 2006. http://www.pi-shurlok.com/Shared/Uploads/NewsArticles/Files/security_and_safety.pdf (accessed 12 March 2010).

Feiler, Peter H., and Dionisio de Niz, CMU SEI, *ASSIP Study of Real-Time Safety-Critical Embedded Software-Intensive System Engineering Practices*, CMU/SEI-2008-SR-001, February 2008. http://www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&doc=GetTRDoc.pdf&AD=ADA480129 (accessed 24 March 2010).

Firesmith, Donald. G., "Analyzing the Security Significance of System Requirements", *Proceedings of the Symposium on Requirements Engineering for Information Security,* Paris, France, 25 August 2005. http://crinfo.univ-paris1.fr/RE05/W06/short3_firesmith.pdf (accessed 18 December 2009) *or* http://www.sreis.org/SREIS_05_Program/short3_firesmith.pdf (accessed 14 January 2010).

Gamble, M.T., and R.F. Gamble, University of Tulsa, "Isolating Mechanisms in COTS-based Systems", in *Proceedings of the Sixth International IEEE Conference on COTS-Based Software Systems*, Banff, Alberta, Canada, 26 February-2 March 2007, pages 33-40. http://www.seat.utulsa.edu/papers/ICCBSS07-Todd.pdf (accessed 31 March 2010).

Goertzel, Karen Mercedes, and Larry Feldman, "Software Survivability: Where Safety and Security Converge", in *Proceedings of the American Institute of Aeronautics and Astronautics (AIAA) Infotech@Aerospace Conference,* Seattle, WA, 6-9 April 2009.

Goertzel, Karen Mercedes, Ted Winograd, *et al.* for the Department of Homeland Security, *Enhancing the Development Life Cycle to Produce Secure Software,* Version 2.0, published for DHS by the DOD Data and Analysis Center for Software (DACS), October 2008. https://www.thedacs.com/techs/enhanced_life_cycles/ (accessed 31 March 2010; download requires free registration on DACS site). Also available online: https://wiki.thedacs.com/Enhancing_the_Development_Life_Cycle_to_Produce_Secure_Software (accessed 31 March 2010).

Goldenson, Dennis R., and Matthew J. Fisher, Carnegie Mellon University Software Engineering Institute, *Improving the Acquisition of Software Intensive Systems*, CMU/SEI-2000-TR-003—ESC-TR-2000-003, August 2000. http://www.sei.cmu.edu/library/abstracts/reports/00tr003.cfm (accessed 24 March 2010).

Goldin, Dina, and David Keil, University of Connecticut, "Interactive Models for Design of Software-Intensive Systems", in *Proceedings of the Workshop on the Foundations of Interactive Computation*, Edinburgh, Scotland, 9 April 2005. http://www.engr.uconn.edu/~dqg/papers/sod.pdf (accessed 24 March 2010).

Han, Jun, Ryszard Kowalczyk, Swinburne University of Technology (Melbourne, Australia) and Khaled M. Khan, Qatar University, "Security-Oriented Service Composition and Evolution", in *Proceedings of the 13th Asia Pacific Software Engineering Conference*, Bangalore, India, 6-8 December 2006. http://www.ict.swin.edu.au/personal/jhan/jhanPapers/apsec06sec-comp.pdf (accessed 14 January 2010).

Hepner, M., R. Gamble, M. Kelkar, L. Davis, and D. Flagg, University of Tulsa, "Patterns of Conflict among Software Components", in *The Journal of Systems and Software*, Volume 79 Issue 4, April 2006, pages 537-551. http://www.seat.utulsa.edu/papers/JSS05-Hepner.pdf (accessed 31 March 2010).

ISO/IEC 42010:2007, *Systems and Software Engineering—Recommended practice for architectural description of software-intensive systems*.

Johns, Lionel S., Peter Sharfman, Thomas H. Karas, Anthony Fainberg, C.E. "Sandy" Thomas, and David Weiss, U.S. Congress, Office of Technology Assessment, *SDI: Technology, Survivability, and Software*, OTA-ISC-353 (Washington, DC: U.S. Government Printing Office, May 1988). http://govinfo.library.unt.edu/ota/Ota_3/DATA/1988/8837.PDF (accessed 14 January 2010). Also: http://handle.dtic.mil/100.2/ADA339517 (accessed 14 January 2010).

Khan, Khaled M., University of Western Sydney, and Jun Han, Swinburne University of Technology, "A Process Framework for Characterising Security Properties of Component-Based Software Systems", in *Proceedings of the 2004 Australian Software Engineering Conference*, Melbourne, Australia, 13-16 April 2004. http://www.ict.swin.edu.au/personal/jhan/jhanPapers/aswec04.pdf (accessed 29 March 2010).

Lau, Kung-Kiu, and Zheng Wang, "Software Component Models", in *IEEE Transactions on Software Engineering*, Volume 33 Number 10, October 2007. http://www.cs.man.ac.uk/~kung-kiu/pub/tse07.pdf (accessed 31 May 2010).

Lau, Kung-Kiu, and Vladyslav Ukis, University of Manchester, "On Characteristics and Differences of Component Execution Environments", University of Manchester Preprint CSPP-41, February 2007. http://www.cs.man.ac.uk/~kung-kiu/pub/cspp41.pdf (accessed 31 May 2010).

Lau, Kung-Kiu, Vladyslav Ukis, and Zheng Wang, University of Manchester, "Predictable Assembly: Towards a Definition", presented at the Second Workshop on Predictable Software

Component Assembly, Manchester, UK, 12 September 2005. http://www.cs.man.ac.uk/~wangz0/program.html (accessed 7 March 2010).

Leveson, Nancy G., and Turner, Clark, "An Investigation of the Therac-25 Accidents", in *IEEE Computer*, Volume 26, Number 7, July 1993, pages 18-41. http://cs.ucla.edu/~kohler/class/05f-osp/ref/leveson93investigation.pdf (accessed 18 December 2009).

Leveson, Nancy G., *Safeware: System Safety and Computers* (Boston, MA: Addison-Wesley, 1995).

Maier, Mark W., The Aerospace Corporation, "Architecting Principles for Systems-of-Systems", in *Systems Engineering*, Volume 1 Issue 4, 1988, pages 267-284. http://www.infoed.com/Open/PAPERS/systems.htm (accessed 31 March 2010).

Mayoral, Leo, and Clay Smith, Johns Hopkins University Advanced Physics Laboratory, "Extending FMECA to System of Systems (SoS) Interfaces", presented at the National Defense Industrial Association 12th Annual Systems Engineering Conference, San Diego, CA, 26-29 October 2009. http://www.dtic.mil/ndia/2009systemengr/8866WednesdayTrack6Mayoral.pdf (accessed 31 March 2010).

McDermott, Edwin P., Sharam Sarkani, and Thomas A. Mazzuchi, George Washington University, "Air Force Modeling and Simulation Training Toolkit (AFMSTT)", presented at the National Defense Industrial Association 12th Annual Systems Engineering Conference, San Diego, CA, 26-29 October 2009. http://www.dtic.mil/ndia/2009systemengr/8825WednesdayTrack4McDermott.pdf (accessed 31 March 2010).

Meacham, Desmond J., Flight Lieutenant, Royal Australian Air Force, *Standards Interoperability: Application of Contemporary Software Safety Assurance Standards to the Evolution of Legacy Software,* Naval Postgraduate School Master of Science Thesis, March 2006. http://www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&doc=GetTRDoc.pdf&AD=ADA445887 (accessed 14 January 2010).

Meyer, Bertrand, ETH Zürich, "The Grand Challenge of Trusted Components", in *Proceedings of the 25th International Conference on Software Engineering*, Portland, Oregon, 3-10 May 2003. http://se.ethz.ch/~meyer/publications/ieee/trusted-icse.pdf (accessed 31 May 2010).

NAVSEA, *Weapon System Safety Guidelines Handbook*, Section II Chapter 14 "Software Safety", NAVSEA SW020-AH-SAF-010, 2005. http://www.lejeune.usmc.mil/eso/orders/sw020-af-hbk-010.pdf (accessed 14 January 2010).

Navy Joint Software Systems Safety Engineering Workgroup, *Joint Software Systems Safety Engineering Handbook*, DRAFT Version 0.95, 30 September 2009.

Rechtin, Eberhardt, and Mark W. Maier in *The Art of Systems Architecting,* Second Edition (Boca Raton, FL: CRC Press LLC, 2000), page 6.

Vorobiev, Artem, and Jun Han, Swinburne University of Technology (Melbourne, Australia), "Secrobat: Secure and Robust Component-based Architectures", in *Proceedings of the 13th Asia Pacific Software Engineering Conference*, Bangalore, India, 6-8 December 2006. http://www.ict.swin.edu.au/personal/jhan/jhanPapers/apsec06secrobat.pdf (accessed 29 March 2010).

Wikipedia: N-version programming. http://en.wikipedia.org/wiki/N-version_programming (accessed 14 January 2010).

Winograd, Theodore, Holly Lynne McKinley Schmidt for NAVSEA NOSSA, *Software Security Assessment Tools Review*, Draft 2 March 2009.

Wirsing, Martin, and Rémi Ronchaud, "Report on the EU/NSF [European Commission/National Science Foundation] Strategic Workshop on Engineering Software-Intensive Systems, Edinburgh, Scotland, 22-23 May 2004". http://www.ercim.eu/EU-NSF/sis.pdf (accessed 24 March 2010).

Ye, Fan, and Tim Kelly, University of York, "Contract-Based Justification for COTS Component within Safety-Critical Applications", in *Proceedings of the the 9th Australian Workshop on Safety RelatedProgrammable Systems*, Brisbane, Australia, October 2004. http://www.acs.org.au/documents/public/crpit/CRPITV47Ye.pdf (accessed 12 March 2010).

APPENDIX C. ENGINEERING SOFTWARE FOR SURVIVABILITY

The following is an excerpt from "Software Survivability: Where Safety and Security Converge".[85]

The software assurance[86] community recognizes the achievements of software reliability and safety practitioners in improving the engineering of software. The job now is to determine which of these techniques, methods, and tools can realistically be leveraged to achieve security objectives in larger, more complex software programs and applications, such as those typically used in information systems.

What is widely believed to be a more effective approach than "vulnerability avoidance" is the design of software that can recognize and avoid, resist, or tolerate most anticipated attacks, and can quickly and with minimal damage recover from those attacks (anticipated and unanticipated) that cannot be avoided, resisted, or tolerated.

Software needs to be designed to be survivable. This has always been true of safety-critical software, but the ever-increasing reliance of the public and private sectors on information technology has engendered availability requirements for software with very low thresholds of tolerance for service disruption. Less tolerance of downtime means a greater imperative for continuous availability. Survivability is the means to that end.

Designing for survivability enables the software to take advantage of available redundancy and rapid recovery features at the system level. For example, if the system supports automatic backups and hot-sparing of high-consequence components with automatic swap-over, the software's design should be modularized in such a way that its high-consequence components can be decoupled and replicated on the "hot spare" platforms.

Survivable software contains more error- and exception-handling functionality than program functionality. Error and exception handling can be considered to aid in survivability when the goal of all error and exception handling routines is to ensure that faults are handled in a way that minimizes software failures and prevents the software from entering an insecure state if it does fail.

The software's error and exception handling should be designed so that whenever possible, the software will be able to continue operating in a degraded manner (with reduction in performance, or stopping the execution of lower-priority functions, or accepting fewer [or no] new inputs/connections) until a threshold is reached that triggers an orderly, secure termination of the software's execution. The software should never throw exceptions that allow it to crash and dump core memory, or leave its caches, temporary files, and other transient data exposed.[87]

The exception handling block should include a log (which auditors can later review) to record when and why an exception was thrown and messaging code that can automatically send alerts to the system administrator when an exception requires human intervention.

The overall robustness of a well-designed system is partially predicated by the robustness of its security handling procedures at the code or programming language level. C++ and Java, for example, inherently provide convenient and extensible exception handling support that includes "catching" exceptions (faults that are not necessarily caused by flaws in externally-sourced inputs or violations of software-level constraints) and errors (faults that are caused by external inputs or constraint violations; errors may or may not trigger exceptions).

Exception handling should be proactively designed to the greatest extent possible, through careful examination of code constraints as they occur during the concept and implementation phases of the life cycle. The developer should list all predictable faults (exceptions and errors) that could occur during software execution, and define how the software will handle each of them. In addition, address how the software will behave if confronted with an unanticipated fault or error. In some cases, potential faults may be preempted in the design phase, particularly if the software has been subjected to sufficiently comprehensive threat modeling.

The software's error, anomaly, and exception handling also needs to support backward and forward recovery—backward recovery to allow the software to detect every anomaly and error before that error/anomaly manifests as an exploitable vulnerability or escalates to a failure.

The biggest challenge for secure software design is implementing the software to be able to detect and recognize anomalous and erroneous states in the first place. Watchdog timers that check for "I'm alive" signals from processes are a useful approach, with each watchdog timer set by a software process other than that which it is responsible for observing. Another approach is to check periodically the control sum of the executable code in the memory.

Effectiveness of event—error or anomaly—monitors depends on the correctness of assumptions about:

- The structure of the program being monitored;
- The anomalies and errors that are considered possible, and those that are considered unlikely.

As with all assumptions, these may be invalidated by certain conditions.

An anomaly or error should be detected as near in time to the causal event as possible. This will enable isolation and diagnosis before erroneous data are able to propagate to other components. The number of program self-checks that can be implemented is usually limited by the availability of time and memory. At a minimum, there should be checks for all security-critical states, with risk analysis as the basis for defining the optimal contents and locations of each check.

Monitor checks should be non-intrusive, *i.e.*, they should not corrupt the process or data being checked. In addition, the developer should take particular care when coding logic for monitoring and checking to avoid including exploitable flaws.

Error handling in the software should recognize and tolerate errors likely to originate with human mistakes, such as input mistakes. The designer needs to:

- Allow enough fault tolerance in the software to enable it to continue operating dependably in the presence of a fairly large number of user input mistakes;

- Determine just how much information to provide in error messages by weighing the benefit of helping human users correct their own mistakes against the threat of reconnaissance attackers being able to leverage the knowledge they gain from overly-informative error messages.

Forward recovery measures include the use of robust data structures, the dynamic alteration of flow controls, and the tolerance (*i.e.*, ignoring) of single-cycle errors.

In most distributed software systems, components maintain a high level of interaction with each other. Inaction (*i.e.*, lack of response) in a particular component for an extended period of time, or receipt from that component of messages that do not follow prescribed protocols, should be interpreted by the recipient as abnormal behavior. All components should be designed or retrofitted to recognize abnormal behavior patterns that indicate possible DoS attempts. This detection capability can be implemented within software developed from scratch, but must be retrofitted into acquired or reused components (*e.g.*, by adding anomaly detection wrappers to monitor the component's behavior and report detected anomalies).

Early detection of the anomalies that are typically associated with DoS can make containment, graceful degradation, automatic fail-over, and other availability techniques possible to invoke before full DoS occurs. Anomaly awareness alone cannot prevent a widespread DoS attack, although it can effectively handle isolated DoS events in individual components, as long as detected abnormal behavior patterns correlate with anomalies that can be handled by the software as a whole.

For failures that are unavoidable, the software should be designed to fail securely. The exception handling logic should always attempt corrective action before allowing a failure to occur. Thresholds in the exception handler should be set to indicate "points of no return" beyond which recovery from a fault, vulnerable state, or encroaching failure is recognized to be unlikely or infeasible. Only upon reaching this threshold, the exception handler should allow the software to enter a secure failure state, *i.e.*, a failure state in which none of the software's program data, control data, or other sensitive data, and no resources controlled by the software, are suddenly exposed, thereby minimizing any damage that might result from the failure.

The software should be designed to retain only the bare minimum of needed state information, and should frequently purge data written in cache memory and temporary files on disk. These measures will minimize the likelihood of undesired disclosure of sensitive information, including information about the software itself that can be leveraged by attackers, in case of the software's failure.

Upon failing, the exception handler should return the software to the last known good state (which should always be more secure than the failure state). This said, in safety-critical systems, the imperative for fault tolerance and other measures that enable the system to avoid failure is

paramount: such systems have no threshold of tolerance for the delays typically involved in restoring software after it fails.

---

[85] Goertzel, Karen Mercedes, and Larry Feldman, "Software Survivability: Where Safety and Security Converge", in *Proceedings of the American Institute of Aeronautics and Astronautics (AIAA) Infotech@Aerospace Conference,* Seattle, WA, 6-9 April 2009.

[86] "Software assurance" as used here implies software security assurance, as reflected by the definitions of "software assurance" published by Committee on National Security Systems (CNSS), the Department of Homeland Security (DHS), and the National Institute of Standards and Technology (NIST). By contrast, the National Aeronautics and Space Administration's (NASA) definition refers to software reliability assurance. See: Committee for National Security Systems (CNSS) Instruction Number 4009, *National Information Assurance Glossary*, June 2006. http://www.cnss.gov/Assets/pdf/cnssi_4009.pdf (accessed 9 April 2010).

[87] Core dumps are only acceptable as a diagnostic tool during testing. Programs should be implemented to be configurable upon deployment to turn off their ability to generate core dumps when they fail during operational use. Instead of dumping core when the program fails, the program's exception handler should log the appropriate problem before the program exits. In addition, if possible, configure the size of the core file to be 0 (zero) (*e.g.*, using *setrlimit* or *ulimit* in UNIX); this will further prevent the creation of core files.