Design Assurance Arguments for Intrusion Tolerance*

Steve Dawson Joshua Levy Bob Riemenschneider
Hassen Saïdi Victoria Stavridou Alfonso Valdes
System Design Laboratory
SRI International
Menlo Park, CA 94025, USA
{*dawson,levy,rar,saidi,stavridou,valdes*}*@sdl.sri.com*

Abstract

We introduce the notion of a design assurance argument *as a diverse assembly of design choices, evidence, and reasoning that makes a convincing case that the design of the system, from abstract architecture to the most concrete details of implementation and operation, meets appropriate operational and security requirements. We sketch an approach to forming design assurance arguments and discuss its advantages and applicability to intrusion tolerant systems, using an intrusion tolerant Web server as an illustration.*

1. Introduction

1.1. The challenge of intrusion tolerance

Despite broad acknowledgment that information security is a growing problem, and one no longer confined to the largest and traditionally most security-conscious organizations, there appears to be little agreement on how the problem should be addressed. However, there is general agreement, at least within the security research community, on the failure of some methods — such as retrofitting systems designed without security in mind — and the need to focus on more promising approaches. Since most systems are now built from low-cost, commercial-quality elements that almost surely contain security flaws, the importance of designing systems that continue to function in the presence of security breaches is now quite apparent. In addition to the promise of meeting more stringent security requirements at lower cost, intrusion tolerant systems have the ability to provide graceful degradation of their performance and eventually offer mechanisms for recovery.

*This research was partially sponsored by DARPA under contract number N66001-00-C-8058. The views herein are those of the authors and do not necessarily reflect the views of the supporting agency.

Unlike traditional approaches to secure system design, in which security requirements on components are typically as strong as the overall system requirements, intrusion tolerance focuses on building systems from less trustworthy components that have weaker requirements than the overall system. The wider availability of such components means more flexibility in the design process, and most likely would lead to more economical solutions. However, the intrusion tolerance approach imposes a greater challenge in establishing that overall security requirements are met, since it is necessary to show that the lower-security components are assembled so as to guarantee security, as well as correct functionality. We discuss a few ideas for meeting this challenge.

1.2. Codesign and the design assurance argument

As part of our ongoing Cyberscience project,[1] we have begun to develop a framework for development of secure systems. We call our approach *security codesign* [DDL+01, RD01], by analogy with hardware/software codesign and influenced by work in dependable software architecture and the development of safety-critical systems. The goal of this approach is to integrate security into the design and engineering process of computer systems and to provide evidence that the resulting system meets its security goals. The codesign approach provides two main advantages: it allows a loosening of the coupling between security design and functionality design that streamlines the design process (a feature we will not discuss here), and it enables the construction of an information assurance case showing that the implemented system meets its security requirements. The assurance case is designed for making a convincing, auditable, maintainable case that the implemented system meets its security requirements by covering all phases of system development such as design, implementation, test-

[1] `http://www.sdl.sri.com/projects/cyberscience`

ing, deployment, and maintenance and showing that security requirements are accounted for in all phases. A critical piece of this case is a *design assurance argument* (DAA), a diverse assembly of evidence, design details, and reasoning that makes a convincing case that the the design of the system, from abstract architecture to the most concrete details of implementation and operation, meets appropriate operational and security requirements.

We believe that by properly structuring and recording design information, it is possible to construct DAAs that give high assurance that an implemented system meets its goals in terms of operation, security, and intrusion tolerance. Because of their comprehensive scope, encompassing all layers of abstraction, DAAs include both formal and informal reasoning, with the amount of rigor determined by the needs of the application. Furthermore, these arguments can be constructed from information captured during the design process, and do not depend on any specific design methodology. In this paper, we sketch an approach to forming DAAs and discuss its advantages and applicability, using an intrusion tolerant Web server as an illustration.

2. The design assurance argument

A complete description of an implemented system, even a small system such as a simple server, is a complex assembly of many details, abstract and concrete, specifying, for instance, conceptual architecture, data flow, software and operating system modules, source code, machine code, hardware operation, and the numerous ways the different views of the system relate. Obviously, since requirements for a system often concern all these ingredients, a design assurance argument must also be a very complex entity. In general, a DAA could include each description, from most abstract to most concrete, and reasoning about each description and relating different descriptions. When complete, it would argue that the most abstract description is a faithful interpretation of the high-level requirements, and the most concrete descriptions of the system, which should be quite close to the actual implementation, is a refinement of the abstract description that preserves both the functional and the intrusion tolerance requirements.

It is important that the DAA include descriptions that are as concrete as possible, since it is essential to ensure that the system itself runs according to its higher-level requirements. Often, formal verification focuses on the relationship between two descriptions, such as between a specification and a piece of code. The DAA would include this verification, but should also give some reason, not necessarily formal, that the the system actually executes the code that was verified. In summary, a DAA is

- Concise and readable by human auditors

High-level design
(e.g. basic functionality)
↓
Abstract architecture
(e.g. major components, data flow)
↓
Implementation
(e.g. code, platforms)
↓
Operation
(e.g. application and OS operation, hardware operation)

Figure 1. Design of a system at different levels of abstraction.

- Sound, in that it has no contradictions or gaps
- Structured, so that it clearly assembles different aspects of the design
- Broad in scope, covering all layers of abstraction
- Semi-formal, in that it may contain informal or a combination of formal and informal arguments

Given the detail and comprehensive scope of a DAA, it is clear that creating a DAA for an existing system from scratch would be a formidable, if not impossible, task. However, we believe a DAA is essentially an explicit representation of the type of argument designers implicitly create when developing a system. The DAA is a structured assembly of all the design decisions, reasoning, and factual information that would be used to explain each phase of system design. In fact, by sufficiently documenting the design process, we may be able to capture both the content and structure of a DAA. In our Cyberscience investigation, we have explored a similar idea, advocating the creation of a codesign object base that records relevant data from the design process, and is used to generate an information assurance case.

These notions become clearer if we consider the design of a system at various levels of abstraction, as in Figure 1. In some cases, designers may progress in an orderly "top-down" manner through the layers of abstraction, with higher-level abstract design preceding lower-level implementation decisions, but in general, development can take place at different layers at different times, or at more than one layer at once. Regardless of the development methodology, designers are explicitly and implicitly making choices at each of these levels to meet the requirements of the system. If a sufficiently well-organized and detailed development record of this process were kept, we would have de-

scriptions — from architecture diagrams to code to hardware choices — at each level of the hierarchy, together with the evidence and reasoning explaining how the choices meet the needs of the system. Typically, development at a given moment focuses on a certain part (or all parts) of the system, at a certain level of abstraction. The captured reasoning at that moment — formal or informal — can be distilled into an argument, based on relevant evidence, that the design description for a certain part of the system at a certain level of abstraction meets all relevant requirements (in particular, requirements at this level of abstraction) of the system under specified assumptions. Properly expressed, the requirements, design choices, reasoning, and assumptions form a *design element* that captures that portion of the development (Figure 2).

The development process yields design elements at all levels of abstraction. The most abstract design element consists of the high-level requirements and basic form of the system. Slightly more concrete design elements could detail the architecture of the system, and of various components. The key idea is that each element would give an argument about why its contents satisfy its requirements, under specified assumptions. Often, assumptions become requirements for design elements of subsystems or of more concrete components. For instance, an architectural design element could decompose a system into several components and specify their interoperation, placing assumptions on the behavior of components. Lower-level design elements would take these assumptions as their own requirements. Other assumptions may impose constraints, such as on the environment, that cannot be directly met by other design elements.

The DAA is the organization of design elements into a concise and readable argument that the high-level requirements of the system are met, subject to an unestablished (but reasonable) set of assumptions. Depending on the high-level requirements, the DAA may not refer to design elements for all parts or aspects of the system, but only to relevant ones.

3. An example: design of an intrusion tolerant Web server

As an illustration of how a DAA can be assembled from elements of the design process, we give an overview of our own work in designing and building a dependable intrusion tolerant Web server (the *DIT system*) [VAC+01]. As the purpose is simply to give a feel for the approach to forming a DAA, we discuss a simple instance of the system and omit the details of a number of mechanisms.

The mission of the DIT system is to provide, at reasonable cost, a system for high-availability distribution of Web content by incorporating widely available, relatively low-assurance COTS software into a high-assurance intrusion tolerant design. The emphasis is on availability and integrity, not confidentiality, of the service.

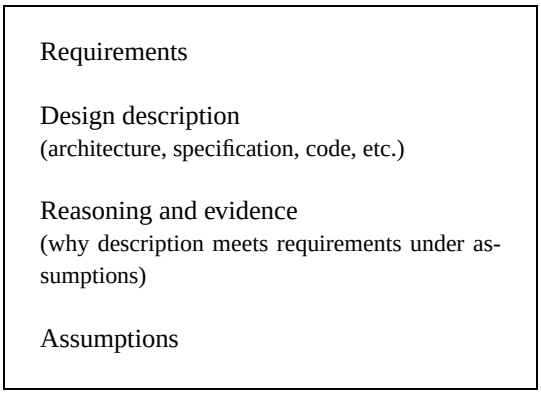

Figure 2. A design element.

3.1. Overview of the DIT system

The architecture of the system is based on the observation that widely available COTS Web server software is feature-filled and complex, and tends to contain security vulnerabilities (such as those exploited by last year's Code Red virus). However, different Web server programs and operating systems have different vulnerabilities, so a system with redundant diverse Web servers on diverse platforms should be able to provide a greater assurance of availability and integrity, provided we have a reliable mechanism for comparing and forwarding responses from the redundant servers to clients. The DIT system attempts to accomplish just this, using a *proxy* to forward client requests to a collection of diverse *application servers* running COTS software, and a monitoring subsystem that contains intrusions. The proxy is a hardened platform running a small amount of custom code. The simplicity and customized nature of the software on the proxy makes the proxy more amenable to hardening than the application servers, which are running more complex, harder-to-verify COTS software. The proxy accepts client requests, forwards them to a number of application servers, compares the content returned by the application servers, and, assuming enough agree, sends the corroborated answer back to the client. The proxy and application servers communicate over a private network that is monitored by an intrusion detection system (IDS). The IDS provides assurance that ill-behaving compromised application servers will be detected and corrected (by rebooting from read-only media), so that compromises are likely to remain limited to a small number of application servers. A *agreement policy* determines which and how many servers are queried by the proxy for each client request, and how sufficient agreement is determined.

3.2. Obtaining the DAA from the design process

Now let us consider how, as the DIT system is designed and built, relevant information from the design process can be captured. At the highest level, we have the mission goals (requirements) of the system (high-availability Web service at reasonable cost with a certain throughput capability), the approach (an intrusion tolerant system using COTS software), and an argument that the goals will be met given certain assumptions (the system will with high assurance tolerate expected attacks, and the cost constraints on COTS and non-COTS components will be met). Together, these form the highest-level design element, represented at the top of Figure 3.

Moving on to the architecture of the system, the next design element takes the above assumptions as its requirements and describes the topology and basic function of the system: one proxy that communicates with the client and forwards requests and responses between the client and each of multiple COTS application servers. From this design, it is clear that in order to meet the requirement that expected attacks be tolerated, we must make a number of new assumptions, including that

- A majority of the application servers are functioning properly at any one time;
- The proxy implements an agreement policy that serves correct content;
- It is very unlikely that the proxy is compromised by attacks.[2]

Design at more concrete levels naturally focuses on the three basic parts of the architecture: the proxy, the application servers, and the network components connecting them.[3] Design elements for each of these parts detail the more concrete internal design of the components and how it meets the architecture assumptions.

We must also look more concretely at how the components fit together. For example, we must specify the exact protocols used between proxy and application servers. We also must argue that the concrete interoperation follows the assumptions of the more abstract architecture design. For instance, once we know that the application servers are computers running COTS operating systems and software, and that they are connected by Ethernet, we realize the possibility of a single compromised application server attacking and compromising additional application servers. We can argue that this is unlikely if we assume that traffic between application servers is restricted and that compromised application servers will be detected and rebooted. To meet these assumptions we introduce various monitoring mechanisms including an intrusion detection system [PN97].

For examples of more concrete design elements, consider the proxy design. The most abstract design element for the proxy contains a specification of its operation, with explanation of how the specification meets the requirements for the proxy. The specification references the agreement policy, and of course assumes an implementation that follows its specification. The implementation element decides the software and operating system to be used, and another element contains the software code itself. Because of the strict security requirements on the proxy, it is not at all trivial to provide a good argument that the proxy code is correct. Our argument relies on two assumptions. One is that the code includes *online verification*, that is, that it contains an independently implemented mechanism that verifies that the code follows the agreement policy properly. The other is that the code is written and compiled in a secure way, to reduce chances for having typical security vulnerabilities (such as possible buffer overflows).

4. Discussion

Although in the above example, we described the design elements from the top down, in reality the various design elements have been elaborated in a different sequence. Indeed, some design stages can affect more than one element at a time, and two different elements can make assumptions about each other, so a perfectly ordered approach is likely to be impossible. Nonetheless, if all the design elements are eventually complete and span from the most abstract requirements to the concrete details, the assumptions for each element are met, and the reasoning within the elements is sound, then we have essentially created the DAA for the system. Thus, the DAA does not inherently constrain the choice of design methodology.

It may be possible to enrich the DAA with additional information that can not only provide assurance that the system meets its requirements, but can also help justify cost of the components and mechanisms used. For instance, in the DIT system, the DAA could justify not only that the redundancy of the application servers is sufficient to achieve intrusion tolerance goals, but that the inclusion of a certain number of servers is necessary. In this way, in addition to progressing up the abstraction hierarchy, showing that requirements are met, we can also progress down the hierarchy, making cases that design choices are economical.

The DAA also can aid in comparing the security of two systems. In our example, we could determine that a single

[2] This requirement, obviously, is hard to meet. The system we are developing actually includes intrusion tolerant redundancy in the proxy functionality as well.

[3] In this case, each design element seems to focus on a physical entity, but this is not the case in general. The elements simply reflect the natural or convenient points of view for the designers.

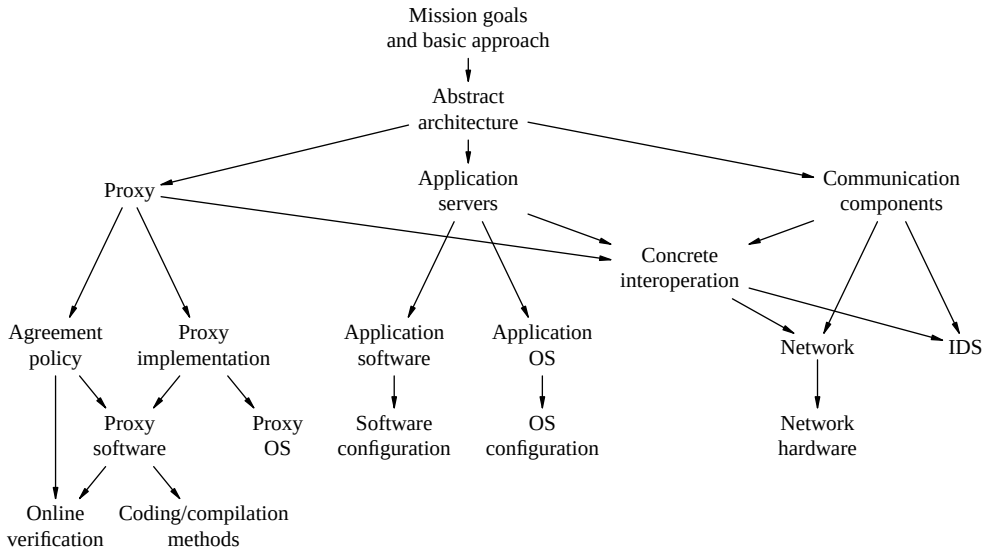

Figure 3. A summary of some of the design elements in the DIT system.

Web application server built from COTS products is less secure than our intrusion tolerant Web server if we establish that the single Web application server cannot reliably meet the high-level requirements of the DAA for the DIT system.

Although the DAA must contain some informal arguments, part of the design element assembly process can be formalized into logic. When the different levels of abstraction of the design are also expressed formally in terms of architectures and abstract models of computations that have to satisfy the requirements, it is possible to check the soundness of the DAA construction. In our example, requirements and assumptions carried out through the design process can be expressed in modal logic with a variety of temporal and knowledge modalities.

References

[DDL+01] Steve Dawson, Bruno Dutertre, Joshua Levy, Bob Riemenschneider, Hassen Saïdi, Victoria Stavridou, and Tomás E. Uribe. Security co-design. Technical report, System Design Laboratory, SRI International, 2001.

[PN97] P. Porras and P. Neumann. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In *National Information Security Conference*, October 1997.

[RD01] R. A. Riemenschneider and Steve Dawson. Dependability co-design. In *NSF Workshop on New Visions for Software Design and Productivity: Research and Applications*, Vanderbilt University, Nashville, TN, Dec 2001.

[VAC+01] Alfonso Valdes, Magnus Almgren, Steven Cheung, Yves Deswarte, Bruno Dutertre, Joshua Levy, Hassen Saïdi, Victoria Stavridou, and Tomás E. Uribe. An adaptive intrusion-tolerant server architecture. Technical report, System Design Laboratory, SRI International, 2001.