INSTITUTE FOR DEFENSE ANALYSES

A Sample Security Assurance Case Pattern

E. Kenneth Hong Fong, *Project Leader*
David A. Wheeler

December 2018

IDA Paper
P-9278

INSTITUTE FOR DEFENSE ANALYSES
4850 Mark Center Drive
Alexandria, Virginia 22311-1882

The Institute for Defense Analyses is a non-profit corporation that operates three federally funded research and development centers to provide objective analyses of national security issues, particularly those requiring scientific and technical expertise, and conduct related research on other national challenges.

About This Publication

This work was conducted by the Institute for Defense Analyses (IDA) under contract HQ0034-14-D-0001, Task AU-5-3856, "Enhancing Program Protection Through Effective Systems Assurance," for OUSD(R&E) Enterprise Engineering. The views, opinions, and findings should not be construed as representing the official position of either the Department of Defense or the sponsoring organization.

Acknowledgments

My thanks to Reginald N. Meeson, Jr. (for providing important feedback on the content), Ken Hong Fong (for providing important feedback on scope and the need for connections with other materials), Bob Martin (for useful early feedback about prioritizing CWEs), Carol Woody (for identifying some valuable supporting information about assurance cases), the Linux Foundation (for supporting and releasing the software and sample assurance case used here as a starting point), and Tom Hurt (for supporting this work).

For more information:

E. Kenneth Hong Fong, Project Leader
ehongfon@ida.org, 703-578-2753

Margaret E. Myers, Director, Information Technology and Systems Division
mmyers@ida.org, 703-578-2782

INSTITUTE FOR DEFENSE ANALYSES

IDA Paper P-9278

A Sample Security Assurance Case Pattern

E. Kenneth Hong Fong, *Project Leader*

David A. Wheeler

Executive Summary

Essentially, all systems with software should address security. However, there is no single "magic bullet" that makes software secure, because security is an emergent property of a system. Developing secure software requires consideration of security across its life cycle. In particular, security must be engineered in, not "bolted on" later. A related term is *software assurance* (SwA), which can be defined as "the level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at any time during its life cycle, and that the software functions in the intended manner" [CNSS 4009]. Unfortunately, considering security throughout the life cycle can be challenging, and the term "software assurance" is an objective—it does not prescribe any one technique. Tracking and managing the application of the various techniques across the software corpus and throughout the software life cycle can be overwhelming.

An *assurance case* is a widely recommended practical alternative to other approaches for managing the assurance activities (as opposed to an overwhelming list or other unstructured methods for recording what was done or not done). An assurance case "includes a top-level claim for a property of a system or product (or set of claims), systematic argumentation regarding this claim, and the evidence and explicit assumptions that underlie this argumentation" [ISO 15026-2:2011]. Because an assurance case is systematic, it is much easier for people to determine if important areas have been adequately covered and to understand the ramifications of different decisions. In this document, we focus on creating and using an assurance case to validate security properties (a "security assurance case"). The idea of a security assurance case is simple, but many have found it difficult to create a security assurance case because of the limited number of sample patterns and worked examples.

This document provides a sample security assurance case pattern, based on a publicly available assurance case of a real commercial system [Wheeler 2018a]. This document also shows how this pattern can be applied to a real system. We hope that many system/software developers and approving authorities will find this sample pattern and application to be a useful place to start when developing their own assurance cases. This document also discusses changes that could be made to deal with different kinds of applications, such as Internet of Things (IoT) or weapon systems. The sample security assurance case pattern provided here is for a system that only requires moderate assurance; higher levels of assurance would call for more rigor. This pattern can make it much easier to create a security assurance case.

Contents

Figures and Tables

1. Introduction

Essentially, all systems with software should address security. However, there is no single "magic bullet" that makes software secure, because security is an emergent property. Instead, developing secure software requires consideration of security across its life cycle. As Department of Defense (DoD) Instruction 5000.02 states, "Cybersecurity is a requirement for all DoD programs and must be fully considered and implemented in all aspects of acquisition programs across the life cycle." [DoDI 5000.02] In short, security must be engineered in, not "bolted on" later. A related term is *software assurance* (SwA), which can be defined as "the level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at any time during its life cycle, and that the software functions in the intended manner" [CNSS 4009].

However, addressing security throughout the life cycle can be challenging. A simple checklist fails to show the interrelationships of issues and easily becomes an overwhelming list that is impractical to prioritize. A simple list also does not assure that all aspects are adequately covered. The term *software assurance* describes an objective, but it is not prescriptive—how can someone counter vulnerabilities with some level of confidence?

An underlying reason that security assurance is difficult (in comparison with many other requirements) is that security properties, like safety properties, are generally emergent and negative properties:

- An emergent property is not implemented in any one component but instead arises from the totality of the system components, their interactions, and the system environment. As a result, security cannot be constrained to any one system component or any one stage of the life cycle.
- A negative property is a property that asserts the system "never does something." This means that simple testing is inadequate; merely showing that the system doesn't do something in one situation typically does not provide enough evidence to justify that a system will never do something.

An *assurance case* is a practical alternative to an overwhelming list or other unstructured methods for recording what was done or not done in a way that can lead to greater confidence in the result. An assurance case "includes a top-level claim for a property of a system or product (or set of claims), systematic argumentation regarding this claim, and the evidence and explicit assumptions that underlie this argumentation. Arguing through multiple levels of subordinate claims, this structured argumentation connects the

top-level claim to the evidence and assumptions." [ISO 15026-2:2011]. Because an assurance case is systematic, it is much easier for people to determine if important areas are adequately covered, and to understand the ramifications of different decisions. An assurance case can be used to justify any important property; historically, they were developed to validate safety properties (a "safety assurance case"), but in this document we focus on using an assurance case to validate security properties (a "security assurance case"). A security assurance case provides a clear and systematic view of why the system, as a whole, is adequately secure.

Many documents have stressed the value of creating an assurance case.[1] The National Defense Industrial Association (NDIA) document *Engineering for System Assurance* identified the assurance case as a key concept, stating that, "the purpose of an assurance case is to provide convincing justification to stakeholders that critical system assurance requirements are met in the system's expected environment(s)" [NDIA 2008]. The National Institute of Standards and Technology (NIST) also stresses their use [NIST SP 800-160]. The International Organization for Standardization (ISO) has developed a standard for assurance cases: ISO 15026. The Object Management Group (OMG) has developed the Structured Assurance Case Metamodel (SACM) to allow the "interchange of structured arguments [assurance cases] between diverse tools by different vendors." [OMG 2018] Also, The Open Group has developed a standard around use of the OMG SACM assurance case "for dependably architecting a system that has required characteristics." [Open Group 2013]. There are also several very helpful documents that provide guidance on how to create security assurance cases, including [Rhodes 2010], [Lipson 2014], and [Goodenough 2014].

The idea of an assurance case is simple, but many have found it difficult to create an assurance case because there are few public examples or patterns to follow. Many real security assurance cases are considered highly confidential, so publicly available worked examples are rare. One of the few public examples available is [Blanchette 2009a] [Blanchette 2009b], which is based on an unnamed defense system. However, although that paper is extremely instructive, it presents an assurance case that is very specific to a single system. [Goodenough 2014] is another valuable document, and it provides some high-level patterns, but it only provides a few fragments and relatively little detail on a specific pattern that could be followed.

This document provides a sample security assurance case pattern, based on a publicly available assurance case of a real commercial system [Wheeler 2018a]. This document also shows how this pattern can be applied to a real system. A video summarizing an earlier version of the assurance case is also available [Wheeler 2017]. We hope that many

[1] For brevity, in this document we will often use the term "assurance case" to mean "security assurance case"; we will occasionally use the full term to remind the reader of our focus.

system/software developers and approving authorities will find this sample pattern to be a useful place to start when developing their own assurance cases.

It is important to understand that having an assurance case document does not guarantee that a system is adequately secure. An assurance case is simply a way to organize information so it can be effectively communicated among stakeholders. Stakeholders may disagree on whether or not a given assurance case is adequate. That said, an assurance case enables all parties to focus on the issues (to determine what is being proposed) instead of being lost in unorganized details.

The sample assurance case pattern shown in this document is based on one for a web application; the text below discusses changes that could be made to deal with other kinds of applications, such as Internet of Things (IoT) or weapon systems. The sample assurance case shown here is for a system that only required moderate assurance; higher levels of assurance would call for more rigor. That said, we believe it will be easier to create another assurance case after seeing one that is worked out.

Chapter 2 presents the sample security assurance case pattern. Chapter 3 shows how this pattern is applied in a real commercial system. In actuality, the pattern shown in Chapter 2 was derived from the material shown in Chapter 3; they are shown in this order to make it easier to understand the pattern. We end the main body with conclusions. Appendix A discusses an important point that the sample assurance case depends on, but may not be understood by all: processes are neither phases nor stages. Security assurance cases are typically a tool that is not directly required; Appendix B discusses how an assurance case can support other documents and processes, including some specific examples for certain documents and processes.

2. Sample Assurance Case Pattern

The following figures present the high-level view of a simple assurance case pattern that focuses on security (aka a "security assurance case"). We expect that those who use this pattern will modify it as necessary to fit their system.

These figures are in Claims, Arguments and Evidence (CAE) notation, which is a simple notation often used for assurance cases. Ovals are claims or sub-claims, whereas rounded rectangles are the supporting arguments justifying the claims. Evidence is shown in rectangles. A common alternative to CAE is Goal Structuring Notation (GSN), but GSN is a more complex notation; for our simple example, we have intentionally chosen a simple notation.

There are tools that can help develop and maintain an assurance case (e.g., Adelard's Assurance and Safety Case Environment (ACSE)). Tools can be useful, but their effective use requires that users understand assurance cases and how they want to structure the assurance case for their system. In addition, users cannot properly evaluate a tool's effectiveness until the users understand what the tool is trying to help them accomplish. In this paper, we focus on the basics; once those are understood, tools can be useful.

Here are a few general principles that we believe are important when developing and updating an assurance case:

1. Where possible, it should be obvious that "all important cases are covered."
2. Where practical, apply the DRY principle ("don't repeat yourself"). In particular, an argument should be justified only once and then reused elsewhere.
3. An assurance case should be designed to be easy to maintain. For example, where possible, use URLs or searchable names to point to evidence (instead of embedding it within a document) to reduce the need for unnecessary updates.[2]

A. Top Level

We have decided that the system must meet a single overall claim, that the "system is adequately secure against moderate threats." This is shown as a claim in the top level diagram of Figure 1.

[2] See [Woody 2018] for a discussion on how to gather some evidence to support an assurance case in an automated way, including using machine learning and text analytics based tools.

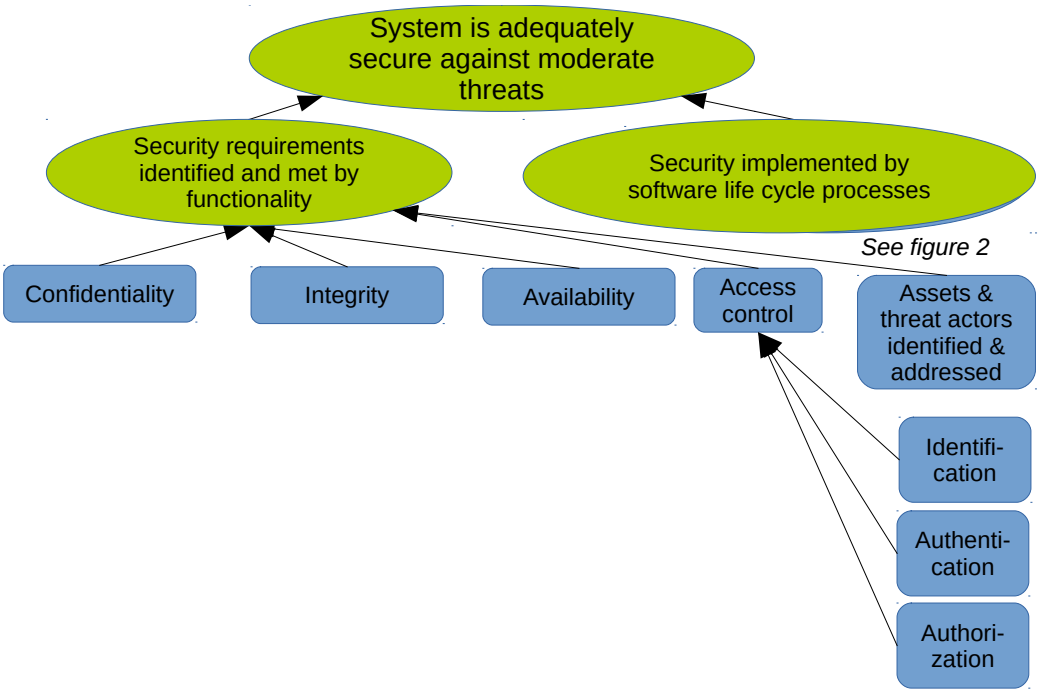

Figure 1. Top Level of an Assurance Case

We must find a way to argue that this top level claim is true. For our purposes, we have decided to divide this into two sub-claims:

1. Security requirements are identified and met by functionality. If we don't know what security requirements must be met, we cannot determine if the system meets them.

    a. This requires knowing the system's basic security requirements (confidentiality, integrity, and availability). Some systems might consider additional requirements as basic, such as non-repudiation and/or accountability.

    b. These basic security requirements must have adequate support by access control functions (identification, authentication, and authorization).

    c. Understanding the security requirements also requires identifying and addressing the assets the system must protect and the threat actors the system must defend against. If the system must directly withstand nation-state attacks, then much more will need to be done in comparison with a system that does not. If the system must withstand insider threats during development and/or operations, then that must also be identified and addressed.

We then decompose each requirement into arguments to justify the claim that the requirement is met, and those arguments should eventually be supported by evidence (typically from the system design, implementation, and verification).

2. Security is implemented by software life cycle processes (that is, the processes that occur within the software life cycle). Unfortunately, failures in security can occur in any software life cycle processes, so we must harden them all to provide adequate security.

The detailed security requirements supporting the top-level figure would be specific to a particular system. For example, most systems will need to keep some information confidential, but they will differ on the information that must be held confidential, which threat actors the information must be kept from, and how the access control will be managed to determine when receiving information is authorized. Chapter 3 shows an example of how this top-level pattern can be applied, including examples of specific requirements.

That said, some requirements apply to many systems. For example, systems that authenticate users using passwords should in almost all cases store passwords as iterated per-user salted hash algorithms (such as bcrypt); this is widely considered to be a minimum standard today. Multiple backups are wise to have when practical. In general, ensure that only authorized developers can make changes to the software, and use version control software to record every change, who made the change, and when the change was made. In many cases, both data in motion and data at rest should be encrypted.

This is not the only way to organize requirements, of course. For example, [Blanchette 2009a] is organized by system Key Performance Parameters (KPPs). The point is to identify the important requirements that will be part of the assurance case.

Each requirement is then decomposed through arguments and evidence to show that it is met and adequately assured (often by reference to evidence produced by the design, implementation, and verification processes). This could be done without any separate reference to processes in the life cycle. However, in practice, many of the detailed requirements will depend on common arguments or assumptions. We avoid repeating the same arguments across multiple requirements (i.e., “keep the assurance case DRY”) by supplementing a structure based on requirements with a structure based on life cycle processes. For example, to have a secure system, it is vital that (1) security principles be applied to the design and (2) common kinds of vulnerabilities are countered in the implementation. It is much clearer and easier to maintain the assurance case if these two points are covered once instead of repeating them for every requirement.

B. Life Cycle Processes

Figure 2 shows a set of arguments to justify the claim that security is implemented in the software life cycle processes as defined by the ISO/IEC/IEEE 12207 standard [ISO 12207:2017]. These are the processes that occur within a life cycle, not the phases or stages of a life cycle as discussed in Appendix A. Figure 2 focuses on the software life cycle technical processes, as we cannot easily show all processes in one figure. Figure 3 and Figure 4 will present the implementation process details, and Figure 5 will present the life cycle processes other than the technical processes.

For simplicity, in a few cases, we have merged multiple ISO/IEC/IEEE 12207 processes into a single process. What we term "requirements process" merges three related processes in ISO/IEC/IEEE 12207 (business or mission analysis, stakeholder needs and requirements definition, and systems/software requirements definition). What we term "design process" also merges three related processes in ISO/IEC/IEEE 12207 (architecture definition process, design definition process, and system analysis process). For each process in the assurance case, we name an argument "Security in <process name>"; this means that we argue that we address security issues as part of that process, and the totality of these arguments justify the claim they support.

We have already covered requirements, so we will now show how security is addressed in design, implementation (by reference), integration and verification, transition and operations, and maintenance. The "transition" process is often called "deployment."

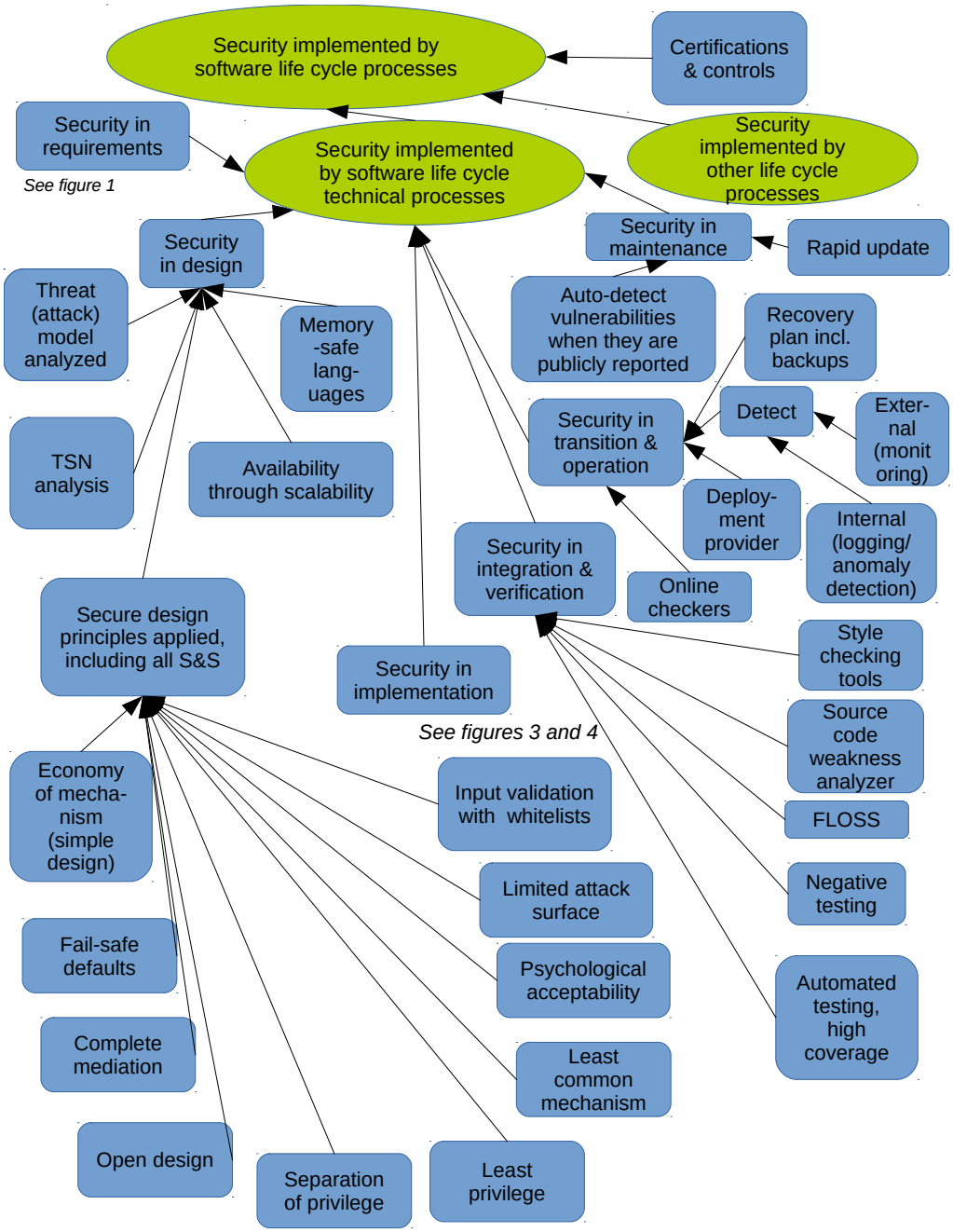

Figure 2. Life Cycle Processes

1. Security in Design

A secure system requires a secure design. The following subsections discuss some approaches for doing so.

a. Threat (Attack) Model Analyzed

It is wise to analyze the system design from the point-of-view of an attacker. This kind of analysis is called “threat modeling” or “attack modeling.” Such analysis can

primarily focus on one of three different aspects, though many real analyses have a little of the other aspects as well:

1. Attacker-centric. This approach starts with the attackers: evaluating their goals and how they might achieve them. The attack tree approach described in [Moore 2001] is an example. The Common Attack Pattern Enumeration and Classification (CAPEC)[3] effort provides a catalog of common attack patterns that can be very useful. This approach is very useful for security experts, but those new to security may struggle to apply an attacker-centric analysis, because they have trouble viewing the world from an attacker's point of view.
2. Asset-centric. This approach starts with the key assets entrusted to a system and works to show that they are adequately protected. This is a rational approach, but those new to security may struggle to correctly determine exactly what assets most need protecting and how to do it.
3. Design-centric. This approach starts with the system design: examining each major element and their interconnections to identify types of attacks against each element and countermeasures that can be employed. Microsoft's Security Development Lifecycle uses this approach. Those new to security may prefer to use this approach, because they will already be familiar with system design.

It is also possible to apply a cyber table top (CTT) exercise; see the verification section for further discussion about CTTs.

b. Trusted Systems and Networks (TSN) Analysis

Trusted Systems and Networks (TSN) analysis is a rigorous approach to identifying addressing and countering risks, including those from malicious components, as presented in the Defense Acquisition Guidebook (DAG) [DoD DAG] in support of [DoDI 5200.44]. This approach can be especially valuable for critical systems. Since this approach especially impacts design, we have placed this analysis within the design process of this assurance case pattern. TSN analysis includes criticality analysis (CA), which identifies mission critical functions and critical components. It also includes vulnerability assessment, risk assessment, and protection measure selection. A brief summary of the approach is in Appendix C.

[3] The CAPEC catalog is at https://capec.mitre.org

c. Secure Design Principles Applied

Secure design principles should also be applied, even if just the well-known principles from Saltzer and Schroeder (S&S). For more about the S&S principles, see [Saltzer 1975]. Here is a brief list and description:[4]

- *Economy of mechanism, aka "Simple design"*: Keep the design as simple and small as practical (e.g., by adopting sweeping simplifications where practical). Complex designs can easily hide defects, including defects that are vulnerabilities.
- *Fail-safe defaults*: Implement access decisions so they deny by default.
- *Complete mediation*: Check every access that might be limited to ensure the request is authorized and non-bypassable.
- *Open design*: Design security mechanisms so that they do not depend on the attacker's ignorance of their design but instead depend on more easily protected and changed information such as keys and passwords.
- *Separation of privilege*: Prefer to use multi-factor authentication, such as requiring both a password and a hardware token, because it is stronger than single-factor authentication.
- *Least privilege*: Operate processes with the least privilege necessary. This includes minimizing the privileges granted, minimizing the time such privileges are granted, and minimizing the size of the component that has elevated privileges.
- *Least common mechanism*: Minimize the mechanisms shared (held in common) by more than one user or process. Shared mechanisms such as memory, directories, databases, operating system kernels, and even CPUs should be reviewed carefully to reduce the risk that the shared mechanism will become a security weakness. Sometimes sharing is necessary or prudent, of course; the goal is to weigh the risks and benefits.
- *Psychological acceptability*: Design the human interface for ease of use; designing for "least astonishment" can help.

We have supplemented the S&S principles with two more: having a limited attack surface and using input validation with whitelists:

[4] The terminology used in the 1975 text is dated; we have reworded the principles here for clarity.

- *Limited attack surface*: Minimize the interfaces where an attacker has an attack opportunity. Minimize the Internet ports that are accessible, the URL paths that are accepted, etc.
- *Input validation with whitelists*: Validate inputs from untrusted sources[5] using a whitelist and not a blacklist. A whitelist is a rule that defines what is legal, as strictly as practical; anything that is not legal should be immediately rejected. For example, input validation for a particular field may require that it have exactly four decimal digits—anything else would then be unacceptable. This is fundamentally different from a blacklist (a rule that defines what is illegal). Clever attackers can often develop another pattern that should be illegal, so security should normally not depend on the use of blacklists. Blacklists can still be useful as a way to create test cases.

A different (though related) set of security principles can be found in an appendix of [NIST SP 800-160]; the point is to identify and consider a set of security principles.

d. Availability through Scalability

It is best when a system is designed for scalability—that is, when it can easily and quickly be scaled up to handle much larger transaction volumes. This provides some quick defenses against denial of service (DoS) attacks. This is much easier to achieve if the system is deployed on a cloud platform. Of course, not all systems can be deployed this way.

e. Memory-Safe Programming Languages

When practical, it is best to prefer memory-safe programming languages. Most programming languages are memory-safe because they prevent many mistakes (such as out-of-bounds array accesses and pointer references) from becoming security vulnerabilities. It is possible to use memory-unsafe languages instead (e.g., C, C++, and assembly). However, when developers make mistakes, use of these programming languages significantly increases the risk of serious security vulnerabilities caused by ordinary programming mistakes. If a memory-unsafe language is used, then a number of additional countermeasures will typically be necessary to have a chance of producing an adequately secure system. This could include using multiple source code weakness analyzers, aggressive use of warning flags, fuzzing, and address space layout randomization (ASLR).

[5] Inputs from at least untrusted sources must be validated. It might be wise to also validate inputs from trusted sources, as they may have inadvertent errors; for our purposes, we will not strictly require it.

2. Security in Integration and Verification

Software must be integrated into a larger system for it to be used. Many software projects use continuous integration (CI) in which all developer working versions are merged into a shared mainline several times a day. CI reduces many integration risks caused by long delays between integration. The CI merge typically also runs a set of automated verification tasks—and that brings us to verification.

Of course, we must verify the software. Verification includes not only execution testing, but any verification process that can detect problems or increase our confidence. There are a large number of different types of tools and techniques that can be used; the *State-of-the-Art Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation 2016* [Wheeler 2016] provides guidance on how to select types of tools and techniques.

The pattern shown here provides a few examples of how to provide some verification. Style-checking tools and source code weakness analyzers can analyze source code to find defects early (so they can be repaired quickly). Using free/libre/open source software (FLOSS), as appropriate, makes it possible to review the source code of reused software. An automated test suite with high coverage is vitally important, as such test suites can detect failures (including regressions). "High coverage" will vary depending on many factors; low-assurance systems may only require 80% statement coverage, whereas higher-assurance systems may require 90% statement coverage and 80% branch coverage or even higher. In addition, we would expect that high coverage systems would have automated tests that cover all the key parts of any high-level specification (e.g., there is a test for every major type of input and transaction). The test suite should include "negative tests" (tests that should fail) to ensure that important security-related actions that should fail will actually fail. For example, there should be tests that ensure that unauthenticated users cannot perform actions that require authentication (where this applies).

Another kind of verification (not shown in Figure 2) is a CTT, an approach used within the DoD. A CTT is a "lightweight, intellectually intensive exercise that explores the effects of cyber offensive operations on the capability of US systems to carry out their missions. It is a wargame-like exercise that focuses on two teams with opposing missions: the military forces charged with executing an operational mission and the cyber mission forces attempting to oppose those military forces." [DoD CTT]. This kind of verification exercise can be done early in development, even before software is developed.

3. Security in Transition and Operation

Of course, security must be considered during transition (aka deployment) and operation. Various "online" checkers should be used to detect common misconfigurations that might occur in the production system. The underlying deployment platform should be evaluated (where there is one). Perhaps most importantly, although *preventing* security

vulnerabilities is very important, we must also *detect* and *recover* when prevention fails. Recovery plans may be simple in some systems (e.g., perhaps you just restore data from backups), but they typically require pre-planning (e.g., you must make the backups and store them in a protected way so that subversion of the system will not damage the backups).

4. Security in Maintenance

Software does not "wear out" in the same sense that physical items do, but security vulnerabilities are often found in reused components after they have been selected. Therefore, projects should automatically detect when vulnerabilities are publicly identified in the software they used and have a process to rapidly update and field those components as necessary.

There are "origin analysis" or "software composition analysis" tools that can examine the subcomponents of a system and then perform analysis based on that. In our case, the key is to determine when a subcomponent has a publicly known vulnerability. In many cases, publicly known vulnerabilities are assigned an identifier called a Common Vulnerabilities and Exposures (CVE) identifier; this is then tracked by databases such as the National Vulnerability Database (NVD). CVEs and the NVD can help alert projects to publicly known vulnerabilities in their components.

5. Certifications and Controls

Many systems must acquire certifications, accreditations, and the use of a variety of controls. These can be valuable, because they provide additional reviews from a different perspectives to ensure that nothing important has been missed. For example, military systems must often comply with a set of controls from the Risk Management Framework (RMF) [NIST SP 800-37] [NIST SP 800-53] [NIST SP 800-53A] [DoDI 8510.01], and these controls (when correctly applied) can help ensure that important actions are performed. However, they can only provide additional review of a system; if a system is not designed for security, certification and controls will typically be unable to fully compensate for that lack.

In some assurance cases, it might make sense to disperse various controls to the parts of the assurance case in which they logically "make sense," and then show under certification the evidence that the system meets the full set of controls it is required to meet.

We now turn to the part of the assurance case that focuses on implementation and the many possible actions there.

C. Implementation

Of course, if the software as implemented is not secure, the system it controls is unlikely to be secure. The following figures show the portion of the assurance case focusing on the software implementation. Figure 3 shows a pattern for a web application, and Figure 4 shows a pattern for embedded software.

Ideally the software implementation would be rigorously proven using formal methods (the application of mathematics to software), supplemented with extensive testing to ensure that the assumptions of the mathematical models were correct. Formal methods have made many strides and are slowly becoming increasingly practical to apply for smaller, very high assurance systems. If your system requires the assurance provided by formal methods, use them and note that in the assurance case. However, the costs of such approaches exceed the benefits in many cases; what can be done for systems with more modest assurance requirements?

From a systems engineering approach, the overall goal is to reduce risks to acceptable levels, which means finding ways to prioritize what is most important. In particular, the goal should be to reduce the probability and/or impact of various risks.

A good way to handle this problem is to observe that most implementation vulnerabilities are due to common types of implementations or common misconfigurations. Thus, if we can identify an appropriate list of implementation errors and common misconfigurations that are likely to apply to this system, it makes sense to focus on those issues. In short, we focus on reducing the highest-risk areas: the kinds of implementation defects most likely to lead to vulnerabilities. No list of common implementation errors and misconfigurations can cover everything, but by also adding *hardening* (measures that reduce/eliminate the security impact of defects), we reduce the probability or impact of a vulnerability even if those other measures fail. In addition, we need to securely reuse components.

Different kinds of applications are different at the implementation level. For example, web applications are significantly different from embedded software such as those in IoT devices and weapon systems. In particular, embedded systems are typically resource-constrained and timing-dependent, leading to the use of memory-unsafe languages (such as C and C++) or languages where memory safety mechanisms are intentionally disabled in some cases (e.g., unsafe Rust or Ada pragmas that suppress run-time checks). The use of unsafe mechanisms should be limited where practical, but this is not always practical (especially for pre-existing systems). Memory-unsafe languages create additional risks, so additional measures should be taken when using them to manage their risks.

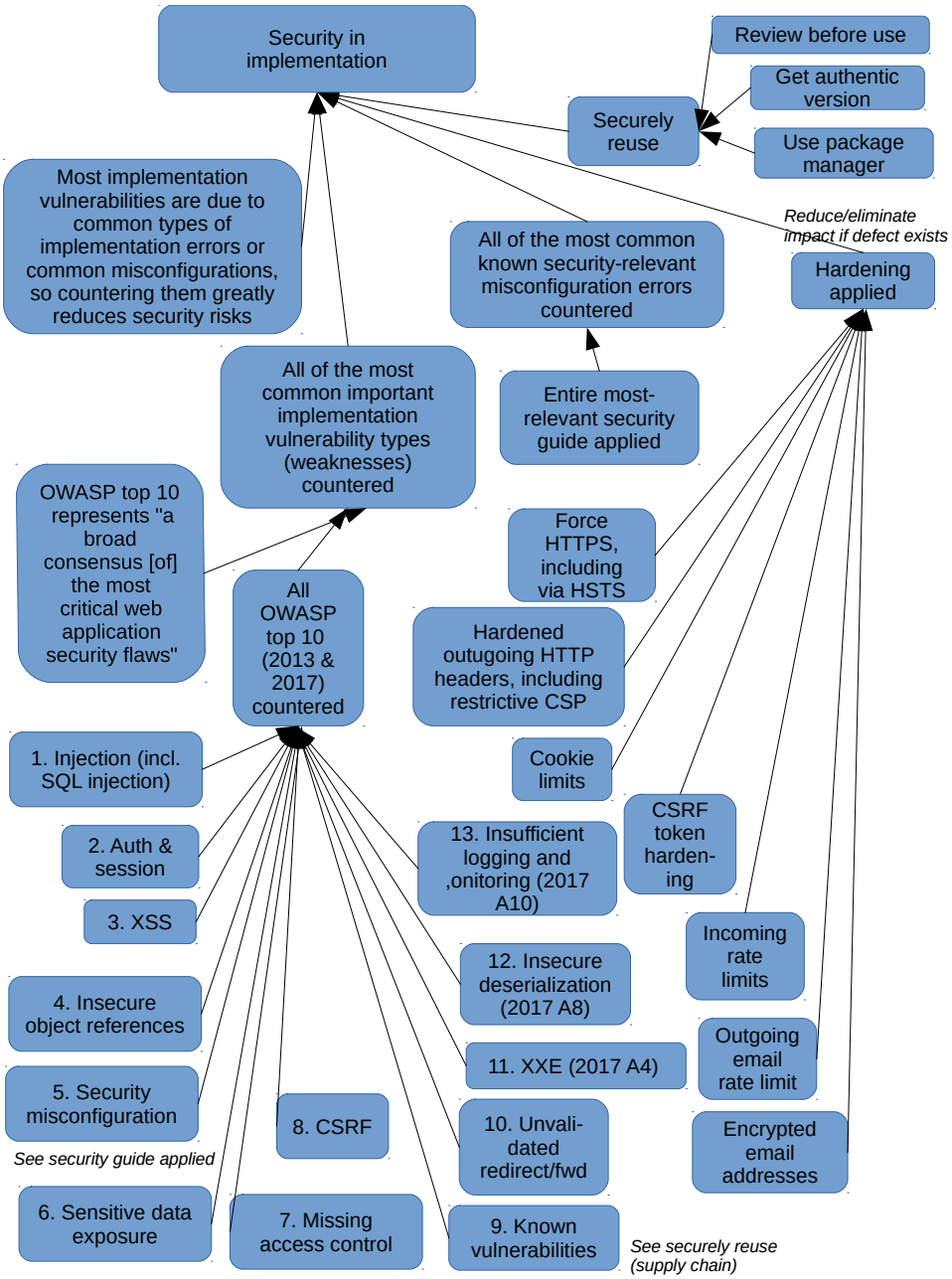

Figure 3. Implementation—Web Application

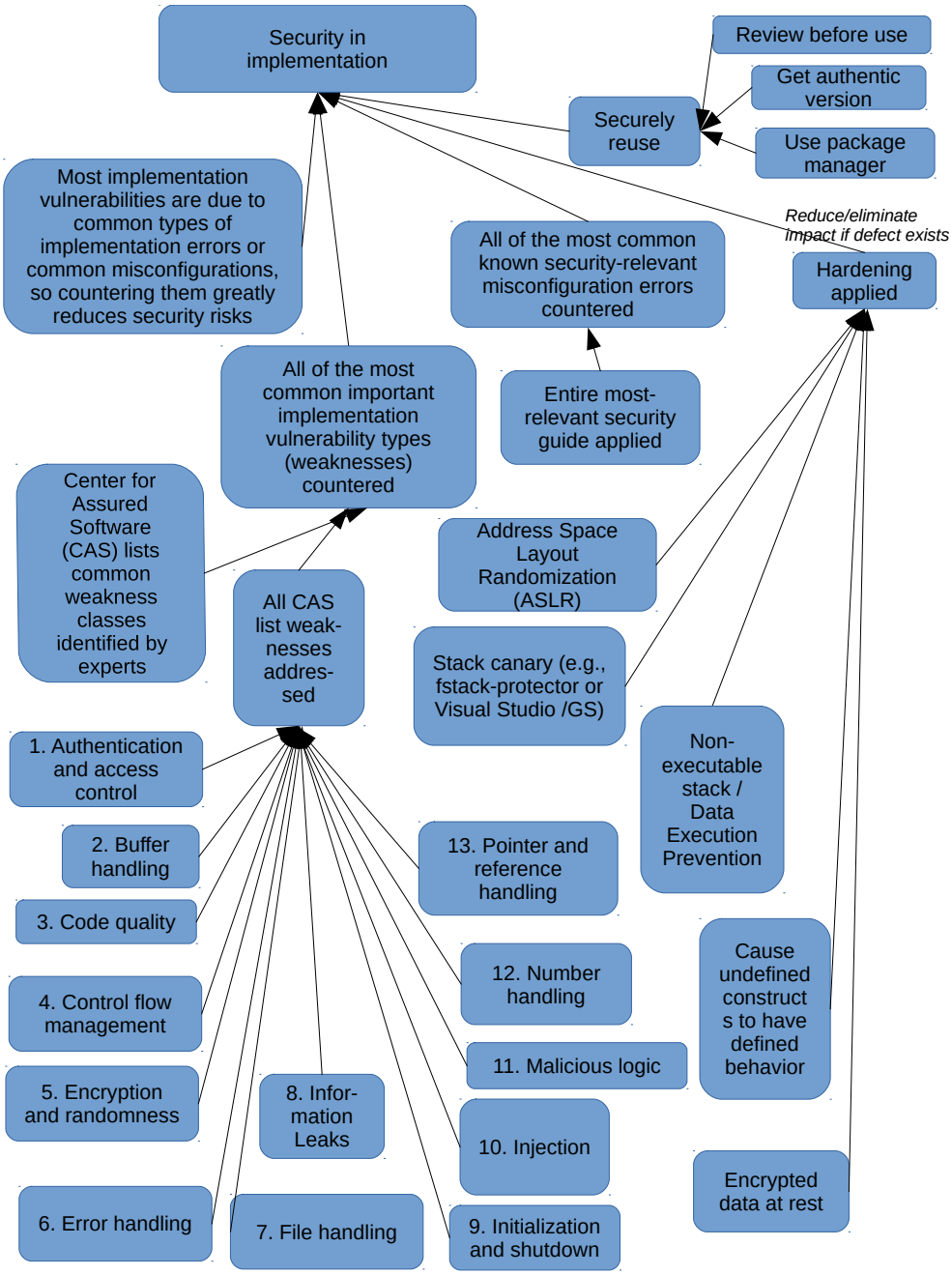

Figure 4. Implementation—Embedded System

1. Common Implementation Errors Countered

The best list of common implementation errors that lead to vulnerabilities would be a large set from that specific system built up over decades of time, but this is rarely available. The Common Weakness Enumeration (CWE) provides a large community-developed list of common security weaknesses, but though important for many activities, it is too large to use directly as a starting point for a single system. Here are some sources of information

on common kinds of vulnerabilities that can be used directly as a starting point for a single system:

- The OWASP Top 10 is a common starting point for web applications. This is a widely vetted list of the "top 10" implementation defects that lead to security vulnerabilities. The example shown in Figure 3 is a combination of the 2013 and 2017 versions of the OWASP Top 10, and thus has 13 items. Note, however, that the OWASP top 10 focuses on web applications and is not a good list for embedded software (where other issues tend to dominate).
- The 13 weakness classes identified by the Center for Assured Software [CAS 2012] are useful; they are listed in Table 1. Note that the State-of-the-Art Resources (SOAR) document [Wheeler 2016] builds on this structure. Figure 4 uses this structure.
- The CWE/SANS top 25 list[6] is often used to identify common kinds of vulnerabilities, especially for applications that are not web applications. This is a widely used general list reviewed by many. There are minor complications when using these vulnerability classes in an assurance case: they are ordered by a risk score instead of having similar items grouped together (e.g., missing authorization and incorrect authorization are far apart). That said, this list has been widely vetted and is a good starting point for an assurance case.
- The NIST Bugs Framework is working to develop rigorous definitions and (static) attributes of bug classes, along with their related dynamic properties. [Bojanova 2016]

For a more specific focus on the weaknesses that will have the most impact to the mission/business function, the project may consider predetermining its own top list of weaknesses. One approach is to create a Top-n CWE list for the project. This can be done by using past history, detailing what the software does, and considering the eight technical impacts of software weaknesses. These eight technical impacts are read data, modify data, DoS due to unreliable execution, DoS due to resource consumption, execute unauthorized code or commands, gain privileges / assume identity, bypass protection mechanism, and hide activities. For more about this approach, see the CWE documentation on prioritizing weaknesses [MITRE 2018]. Additional information supporting this approach is available in [MITRE 2017], section 2.2 of "Incorporating SwA into DoD Acquisition Contracts" [OSD 2017], and "Scoring CWEs" [CWE 2017].

[6] The CWE/SANS top 25 is available at http://cwe.mitre.org/top25/

Table 1. Weakness Classes from the Center for Assured Software

Weakness Class	Example Weakness (CWE Entry)
Authentication and Access Control	CWE-259: Use of Hard-coded Password
Buffer Handling (C/C++ only)	CWE-121: Stack-based Buffer Overflow
Code Quality	CWE-561: Dead Code
Control Flow Management	CWE-483: Incorrect Block Delimitation
Encryption and Randomness	CWE-328: Reversible One-Way Hash
Error Handling	CWE-252: Unchecked Return Value
File Handling	CWE-23: Relative Path Traversal
Information Leaks	CWE-534: Information Exposure Through Debug Log Files
Initialization and Shutdown	CWE-404: Improper Resource Shutdown or Release
Injection	CWE-134: Uncontrolled Format String
Malicious Logic	CWE-506: Embedded Malicious Code
Number Handling	CWE-369: Divide by Zero
Pointer and Reference Handling	CWE-476: NULL Pointer Dereference

It could be argued that some of these common mistakes are really design flaws rather than implementation errors. Structuring an assurance case that way would also be reasonable. In the end, the real goal is to provide assurance to all stakeholders that common mistakes are countered, and we find the structure shown here to be clear and useful.

2. Common Misconfigurations Countered

Misconfiguration errors are typically specific to a programming language, framework, and/or platform. There are already guidance documents for many common ones; web searches can often quickly find some possibly relevant guides.

3. Hardening Applied

Hardening measures are measures that would not be strictly required if the other parts worked perfectly, but because humans and machines are imperfect, hardening measures are an extremely important set of measures for reducing the probability or impact of risks. In a web application, measures such as using a restrictive Content Security Policy (CSP) and incoming rate limits can be very useful.

As noted earlier, embedded software often has additional risks due to the use of memory-unsafe languages (usually C or C++) or disabled memory safety checks. A variety of countermeasures can be used to reduce the risk that the inevitable implementation errors will lead to vulnerabilities caused by the lack of memory safety. These countermeasures

include Address Space Layout Randomization (ASLR), stack canaries (enabled by options such as fstack-protector or Visual Studio's /GS), and the use of non-executable stack / data execution prevention. Some systems may also support control flow integrity; this should normally be enabled where available.

Another problem with C and C++ is that they have a large number of constructs with undefined behavior; undefined behavior can immediately lead to security vulnerabilities, and it is difficult to develop large programs without accidentally causing them. One way to counter undefined behavior is to intentionally enable compiler options to cause undefined behaviors to be defined (e.g., gcc or clang support -fwrapv (wrap signed integer overflow), -fno-strict-overflow, -fno-strict-aliasing, and -fno-delete-null-pointer-checks). Another approach (not shown in the figure) is to use compiler options to enable run-time detection of some undefined behavior so that it can be detected during tests (e.g., gcc/clang -ftrapv (generates code to trap signed integer overflows), -fsanitize=address, -fsanitize=unsigned-integer-overflow, -fsanitize=undefined, and -fcatch-undefined-behavior).

4. Securely Reuse Software

We must also securely reuse software. Such software is a key part of our incoming supply chain. Typically, developers can do at least some basic review before use (e.g., look at the supplier's website for signs of problems), get an authentic version (by using HTTPS and double-checking the name), and use a package manager to track reused software. Such software may be proprietary software or open source software (OSS).

A key challenge today is to review proprietary software in depth where it matters. OSS provides the source code, so potential users can review the software themselves, hire others to do so, or read the reviews by others. In contrast, the source code for proprietary software is often not easily available for review. In some cases, a supplier may be willing to provide source code under some non-disclosure agreement. Potential users may be able to examine the binary (executable) code more deeply using various binary analysis techniques. This can include testing it using simulated attacks (the CAPEC attack patterns may be useful for this purpose). Another approach is to run it in a sandbox to attempt to detect unacceptable behavior. Learning more about the supplier may also help (e.g., the supplier's reputation and development processes). As the probability or impact of problems increases, the measures necessary may increase. None of these measures is foolproof; the goal in all cases should be to reduce the risk to acceptable levels, not to eliminate all risk. If review cannot adequately reduce the risk, design changes (such as restricting that component's privileges) and/or replacing the component may be warranted.

D. Other Life Cycle Processes

We should also consider all the other software life cycle processes beyond the technical processes. The 12207 standard defines three other process groups, with a number of processes within each one as shown:

- *Agreement*: acquisition and supply processes
- *Organizational project-enabling*: life cycle model management, infrastructure management, portfolio management, human resource management, quality management, and knowledge management processes
- *Technical management*: project planning, project assessment and control, decision management, risk management, configuration management, information management, measurement, and quality assurance processes

Figure 5 shows an assurance case pattern using this as a starting point.

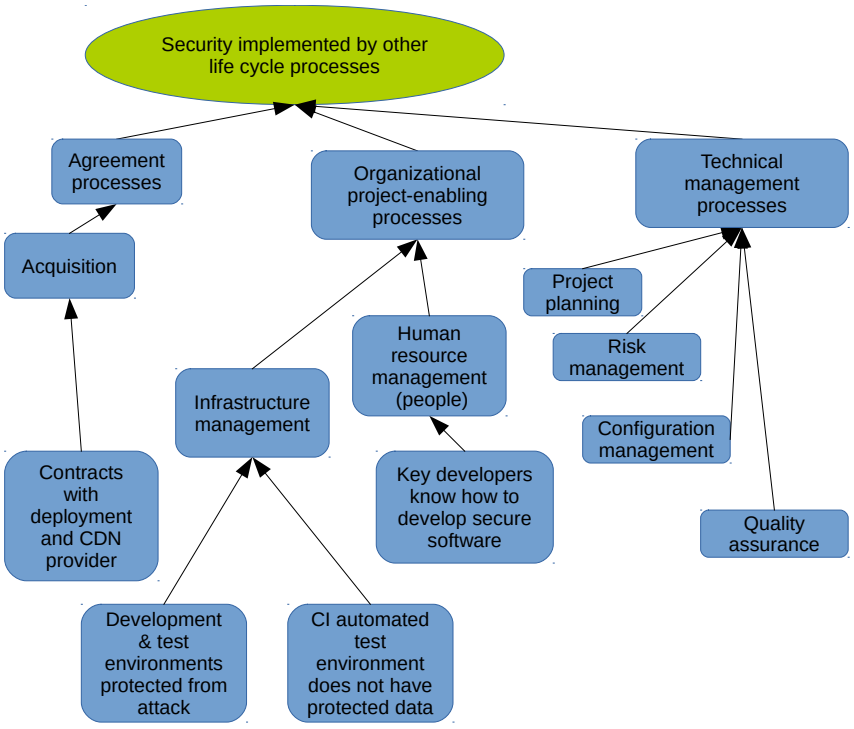

Figure 5. Other Life Cycle Processes

The example shown here notes that if your system depends on a content distribution network (CDN) provider, it would be wise to have a support contract with them. If there is no CDN, then there is no need for a contract with one, but the general pattern still stands: If it's important for an organization's mission, it's wise to have a support contract with someone to support it.

More generally, an acquirer can contract for a security assurance case, including its supporting evidence. That would enable an acquirer to judge the security assurance of what is being developed.

One weakness of a "process-oriented" view is that it can be easy to focus on only the process and not on the actual project state or results. This can especially be a problem with the "infrastructure management" and "human resource management" processes:

- *Infrastructure management*: It is important that the enabling environments (e.g., the development and test environments) actually be secure. Just having a process for establishing or evaluating infrastructure is not enough.
- *Human resource management*: It is important to have good people—in particular, key people must understand how to develop secure software if the results are to be secure. For example, the key people must understand the system's security requirements, must know how to design for security (including knowing secure design principles), must know common implementation failures and how to counter them, and must know how to verify that a system has adequate security (e.g., must understand the different kinds of tools that can be used and what they are effective for). Just having a process for hiring and evaluating people is not enough.

Risk management is vital, indeed, all of assurance could be considered a subset of risk management. In this assurance case, we could have placed "Criticality Analysis, Vulnerability Assessment, Risk Assessment, Protection Measure Selection" (aka the "TSN Analysis Methodology") under risk management instead. When developing an assurance case, the primary point is to provide a justification to ensure that the important activities are getting done and that they are likely to lead to adequate results; exact placement is less important for real systems.

E. Real Assurance Cases Include Supporting Text

In practice, an assurance case is a combination of figures and supporting text. Figures are good at showing larger structure, but can become voluminous and are time-consuming to edit—especially without tools specifically designed to edit assurance cases. We generally recommend using figures to show the larger structure and separate supporting text to provide important details. There should normally be supporting text in at least all the leaves of the diagram to provide additional justification (unless no detail is necessary), as illustrated in Figure 6.

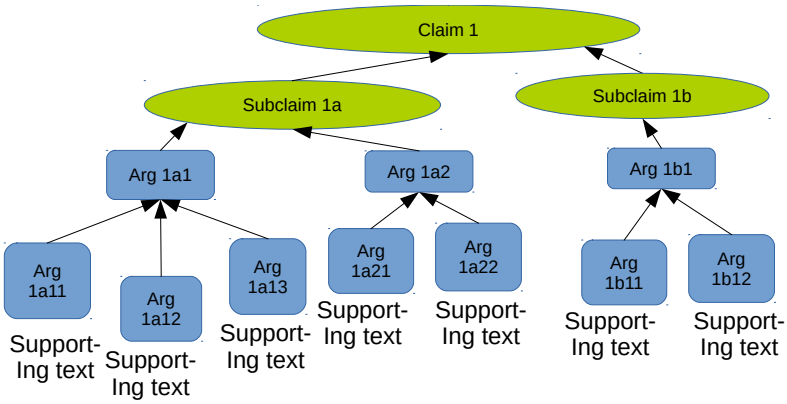

Figure 6. Supporting Text Provides Important Details

For example, in most cases we choose to not show evidence in our figures, but instead provide evidence in the supporting text. We do this because (1) large figures are time-consuming to edit and (2) for many purposes, it is adequate to provide evidence only in the supporting text. Other organizations may choose to show evidence in the figures in more cases than we have done here.

F. Determining Adequacy

The pattern shown here is not enough for a real system. As noted earlier, we expect that those who use this pattern will add key factors specific to their system, such as the system's specific requirements, and they may modify the assurance case as necessary to best address their system's needs.

How deep and rigorous the assurance case must be depends on a variety of factors, including the impact of failure (should a vulnerability be exploited), the likelihood of attack, and the probable strength of those attacks. As noted in [NIST SP 800-160], "The specific form of an assurance case and the level of rigor and formality in acquiring the evidence required by the assurance case is a trade space consideration." The point is to develop an assurance case for the system you are responsible for, so that all relevant stakeholders can understand what will be done and eventually agree that it is (or is not) sufficient. Simply having an assurance case is no guarantee that everything is adequately secure. Instead, an assurance case provides a structured approach so that important decisions can be made within their full context.

[Blanchette 2009a] makes the following observation that we agree with: "There is an enormous tendency to make very small steps in logic as one progresses down the analysis path in order to ensure the absolute soundness of the argument… However, when dealing with a vast analysis space and limited time, one must balance between precision in argument and comprehensibility. At some point, adding more detail is of diminishing value and judicious choices must be made about what to represent and what to omit."

In practice, an assurance case is never "done" until the system has been retired. It may need to be updated as the system is modified. In addition, stakeholders will identify over time various ways that the assurance case should be improved. This is not a failing but a success; an assurance case makes it much easier to determine what is done and not done, so that pointed questions can be directly answered.

[Blanchette 2009a] demonstrates one simple approach to indicate risk: color-coding the claims[7] and evidence. In their case, they colored symbols red for high risk, yellow for medium risk, and green for low risk. More specifically:

- For evidence:
  - Green: Evidence is complete and adequate
  - Yellow: Evidence is incomplete or planned for the future
  - Red: Evidence is complete but inadequate, planned but now late, or non-existent
- For claims:
  - Green: All lower-level claims and supporting evidence are green
  - Yellow: Some lower-level claims and supporting evidence are a combination of yellow and red
  - Red: All, or an overwhelming majority of, lower-level claims and supporting evidence are red

Such color-coding may be useful in some projects, though we do not use it in our sample application. We now turn to that sample application, to show how this security assurance case pattern can be applied in a real system.

[7] They do not separately discuss claims and arguments; we suggest reading "claims" as "claims and arguments" in this color-coding example.

3. Sample Assurance Case Application

This chapter shows how the pattern previously explained in chapter 2 can be applied. This is a real assurance case from a real commercial system, specifically the CII Best Practices Badge application (aka the "BadgeApp"). For more about that system, including more details about its assurance case, see [Wheeler 2018a]. As more details are available elsewhere, we will only briefly highlight a few points, focusing on how the pattern was adjusted and expanded for a particular system.

This particular assurance case was developed using only simple graphics editing tools (in this case LibreOffice Draw) and a text editor. The actual figures do not refer to other figures by number, since that could have made updating the figures more complicated. A specialized tool might be appropriate for a larger assurance case. However, this example clearly illustrates that specialized tools are not necessary to get started.

In practice, the pattern shown in chapter 2 was derived from the material in this chapter, but we believe this order is easier to understand. As we have already discussed the overall pattern, in this section, we only discuss the issues specific to applying the pattern to this particular system.

A. Top Level

We have a single overall claim that the "system is adequately secure against moderate threats." This is decomposed further.

Notice that we have a number of very specific security requirements that are specific to this system and do not necessarily apply to other systems. For example, confidentiality is easier when the data the system manages is almost all public, but this is not realistic for many systems.

That said, some of the requirements shown here do apply to many other systems. Systems that authenticate users using passwords should in almost all cases store passwords as iterated per-user salted hash algorithms (such as bcrypt); this is widely considered to be a minimum standard today. Multiple backups are wise to have where it is practical. In general, ensure that only authorized developers can make changes to the software, and use version control software to record every change, who made the change, and when the change was made. In many cases, both data in motion and data at rest should be encrypted.

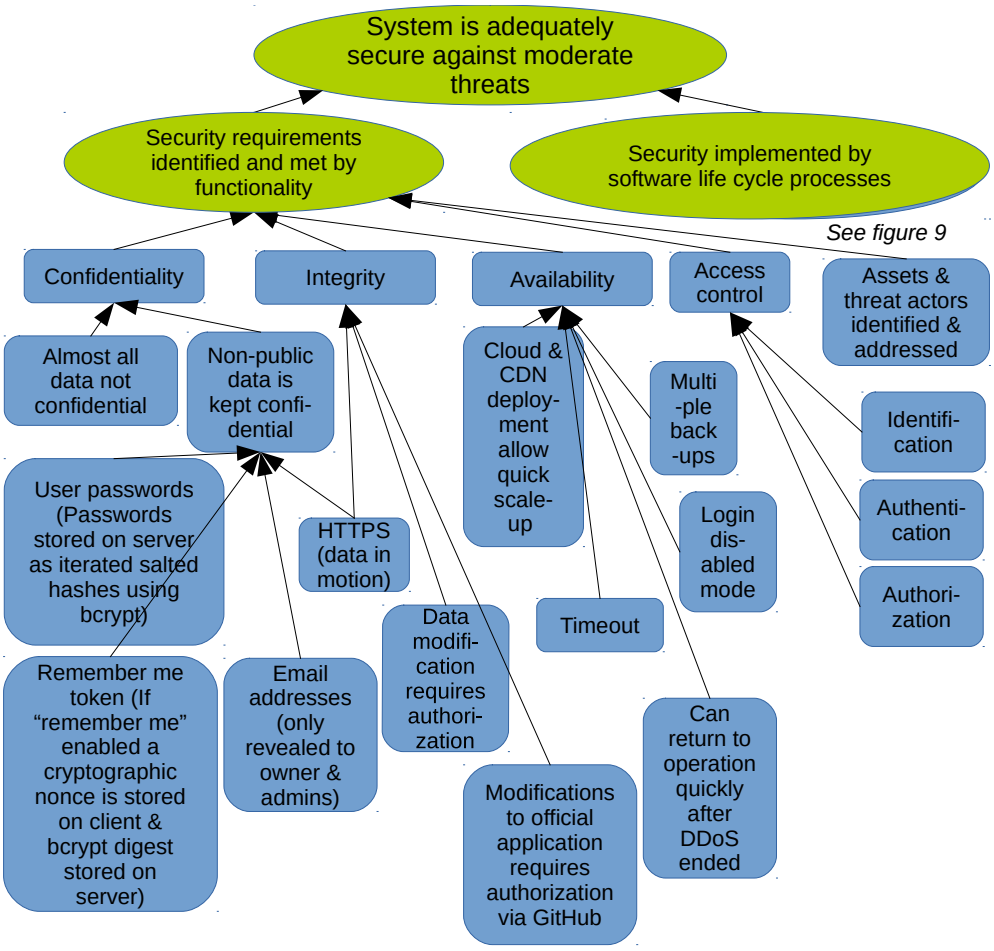

Figure 7. Application: Top Level of Assurance Case

As noted earlier, an assurance case is *not* just a few figures. Different people might choose to represent in figures what others would choose to represent in text; what matters is whether or not the assurance case is adequate for its purpose. In practice, an assurance case often uses figures to show a higher-level view, combined with other textual material to provide more detailed information.

Below is the supporting text for two of the nodes shown in the figure: *Email addresses (only revealed to owner & admins)* and *Data modification requires authorization*? The shaded text is quoted directly from [Wheeler 2018a]. Note the following points about this supporting text:

- The supporting text provides additional detail that justifies that the requirements really are met. Recall that the goal is to provide enough justification to satisfy the stakeholders. This supporting text subdivides the problem further to show (1) all relevant circumstances and (2) that each circumstance is handled properly. For example, the reference to “grep” below provides information on how someone could verify that all relevant circumstances have been covered. In many examples, this assurance case provides (as evidence) the identity of

specific code that implements the requirement and specific automated test(s) that verify them.

- Formatting here follows the original. For example, these subsections use straight quotes and courier font where they are used in the quoted text.
- This sample supporting text refers to other document sections that we have not included (e.g., sections about encrypted email addresses, authorization, and the additional_rights table). These other sections are not necessary for our purposes. For the complete example, see [Wheeler 2018a].

We follow this supporting text with an example of how some of that supporting text could be represented graphically instead.

1. Sample Supporting Text: Email Addresses

Email addresses are only revealed to the owner of the email address and to administrators.

We must store email addresses, because we need those for various purposes. In particular, we must be able to contact badge entry owners to discuss badge issues (e.g., to ask for clarification). We also user email addresses as the user id for "local" accounts. Since we must store them, we strive to not reveal user email addresses to others (with the exception of administrators, who are trusted and thus can see them).

Here are the only ways that user email addresses can be revealed (use `grep -Ri 'user.*\.email' ./` to verify):

- Mailers (in `app/mailers/`). The application sometimes sends email, and in all cases email is sent via mailers. Unsurprisingly, we need destination email addresses to send email. However, in all cases we only send emails to a single user, with possible "cc" or "bcc" to a (trusted) administrator. That way, user email addresses cannot leak to other users via email. This can be verified by examining the mailers in directory `app/mailers/` and their corresponding views in `app/views/*_mailer/`. Even the rake task `mass_email` (defined in file lib/tasks/default.rake), which can send a message such as "we have been breached" to all users, sends a separate email to each user using a mailer. A special case is when a user changes their email address: in that case, information is sent to both email addresses, but technically that is still an email to a single user, and this is only done when someone is logged in with authorization to change the user email address.
- The only *normal* way to display user email addresses is to invoke a view of a user or a list of users. However, these invoke user views defined in `app/views/users/`, and all of these views only display a user email address if

the current user is the user being displayed or the current user is an administrator. This is true for views in both HTML and JSON formats. The following automated tests verify that email addresses are not provided without authorization:

  - `should NOT show email address when not logged in`
  - `JSON should NOT show email address when not logged in`
  - `should NOT show email address when logged in as another user`
  - `JSON should NOT show email address when logged in as another user`

- The `reminders_summary` view in `app/views/projects/reminders_summary.html.erb` does display user email addresses, but this is only displayed when a request is routed to the `reminders_summary` method of the projects controller (`app/controllers/projects_controller.rb`), and this method only displays that view to administrators. This is verified by the automated test `Reminders path redirects for non-admin`.
- As a special case, a user email address is included as a hidden field in a local user password reset in `app/views/password_resets/edit.html.erb`. However, this is only displayed if the user is routed to the "edit" method of `app/controllers/password_resets_controller.rb` and successfully meets two criterion (configured using `before_action`): `require_valid_user` and `require_unexpired_reset`. The first criterion requires that the user be activated and provide the correct reset authentication token that was emailed to the user; anyone who can do this can already receive or intercept that user's email. The need for the correct authentication token is verified by the automated test `password resets`.

As documented in CONTRIBUTING.md, we forbid including email addresses in server-side caches, so that accidentally sharing the wrong cache won't reveal email addresses. Most of the rest of this document describes the other measures we take to prevent turning unintentional mistakes into exposures of this data.

Note: As discussed further in the later section on "Encrypted email addresses", we also encrypt the email addresses using AES with 256-bit keys in GCM mode ('aes-256-gcm'). We also hash the email addresses, so they can be indexed, using the hashed key algorithm PBKDF2-HMAC-SHA256. These are strong, well-tested algorithms. We encrypt email addresses, to provide protection for data at rest, and never provide the keys to the database system (so someone who can only see what the database handles, or can get a copy of it, will not see sensitive data including raw passwords and unencrypted email

addresses). These are considered additional hardening measures, and so are discussed further in the section on hardening.

2. Sample Supporting Text: Data Modification Requires Authorization.

Here we describe how these authorization rules are enforced. We first discuss how to modify data through the BadgeApp application and then note that data can also be modified by modifying it via the underlying database and platform. For more about the authorization rules themselves, see the section on authorization. Note that gaining authorization first requires logging in (which in turn requires both identification and authentication).

The only kinds of data that can be modified involve a project or a user, and this data can only be modified through the application as follows:

- Project: Any project edit or deletion request is routed to the appropriate method in the projects controller in `app/controllers/projects_controller.rb`. Users cannot invoke any other method to modify a project other than the four methods corresponding to the requests identified below, and these cannot be executed unless the appropriate authentication check has succeeded:
  - In the case of an `edit` or `update` request, there is a `before_action` that verifies that the request is authorized using the check method `can_edit_else_redirect`. (Note: technically only `update` needs authentication, since `edit` simply displays a form to fill out. However, to reduce user confusion, we prevent *displaying* a form for editing data unless the user is authorized to later perform an update.) This inability to edit a project without authorization is verified by automated tests `should fail to update project if not logged in` and `should fail to update other users project`.
  - In the case of a `delete_form` or `destroy` request, there is a `before_action` that verifies that the request is authorized using the check method `can_control_else_redirect`. (Note: Again, technically only `destroy` needs authentication, but to reduce user confusion we will not even display the form for destroying a project unless the user is authorized to destroy it.) This inability to destroy a project without authorization is verified by automated tests `should not destroy project if no one is logged in` and `should not destroy project if logged in as different user`.
- User: Any user edit or deletion request is routed to the appropriate method in the user controller in `app/controllers/users_controller.rb`. These cannot be executed unless the appropriate authentication check has succeeded. In the case of an `edit` or `update` or `destroy` request, there is a `before_action` that verifies that the request is authorized using the check method

`redir_unless_current_user_can_edit`. Users cannot invoke any other method to modify a user. This inability to edit or destroy a user without authorization is verified by these automated tests:

- `should redirect edit when not logged in`
- `should redirect edit when logged in as wrong user`
- `should redirect update when not logged in`
- `should redirect update when logged in as wrong user`
- `should redirect destroy when not logged in`
- `should redirect destroy when logged in as wrong non-admin user`

The `additional_rights` table, described below, is edited as part of editing its corresponding project or deleting its corresponding user, and so does not need to be discussed separately. No other data can be modified by normal users.

It is also possible to directly modify the underlying database that records the data. However, only an administrator with deployment platform access is authorized to do that, and few people have that privilege. The deployment platform infrastructure verifies authentication and authorization.

3. Sample Graphical Representation That Data Modification Requires Authorization

The actual assurance case only had the supporting text for the claim that "Data modification requires authorization." Text is easy to maintain, and the stakeholders were satisfied having that detail in text.

However, some stakeholders may prefer to have more of the assurance case represented using a graphical notation. Figure 8 is an example showing how more information could be provided using a graphical notation (as was not done in the actual system). Note that this graphical representation would still need to be supplemented with text; see the previous discussion of text versus graphical representations in section 2.E. In addition, this sample graphical representation includes some specific information such as the names of controllers, methods, and authorization checks.

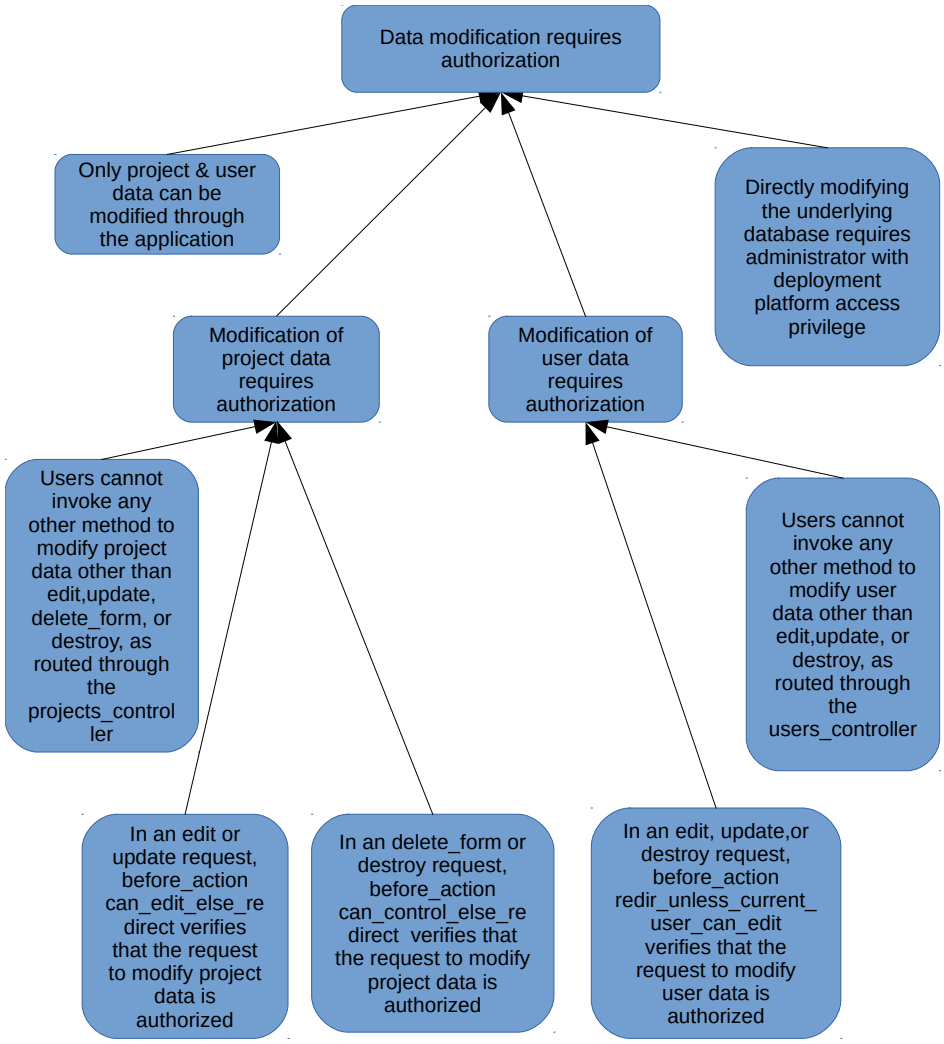

Figure 8. Graphical Representation That Data Modification Requires Authorization

B. Life Cycle Processes

Figure 2 shows a set of arguments to justify the claim that security is implemented in all software life cycle processes.

A secure system requires a secure design. It is wise to apply threat modeling (aka attack modeling) so that the system is evaluated from an attacker's viewpoint before it is even built. In this example, we used STRIDE,[8] which examines the security issues in every major design component. The actual assurance case in [Wheeler 2018a] walks through each design component and shows that each design component is adequate for its purpose (from a security point-of-view).

[8] STRIDE is a mnemonic, not an acronym. It is used to remind users of the following list of attacks to counter: spoofing of user identity, tampering, repudiation, information disclosure (privacy breach or data leak), DoS, and elevation of privilege.

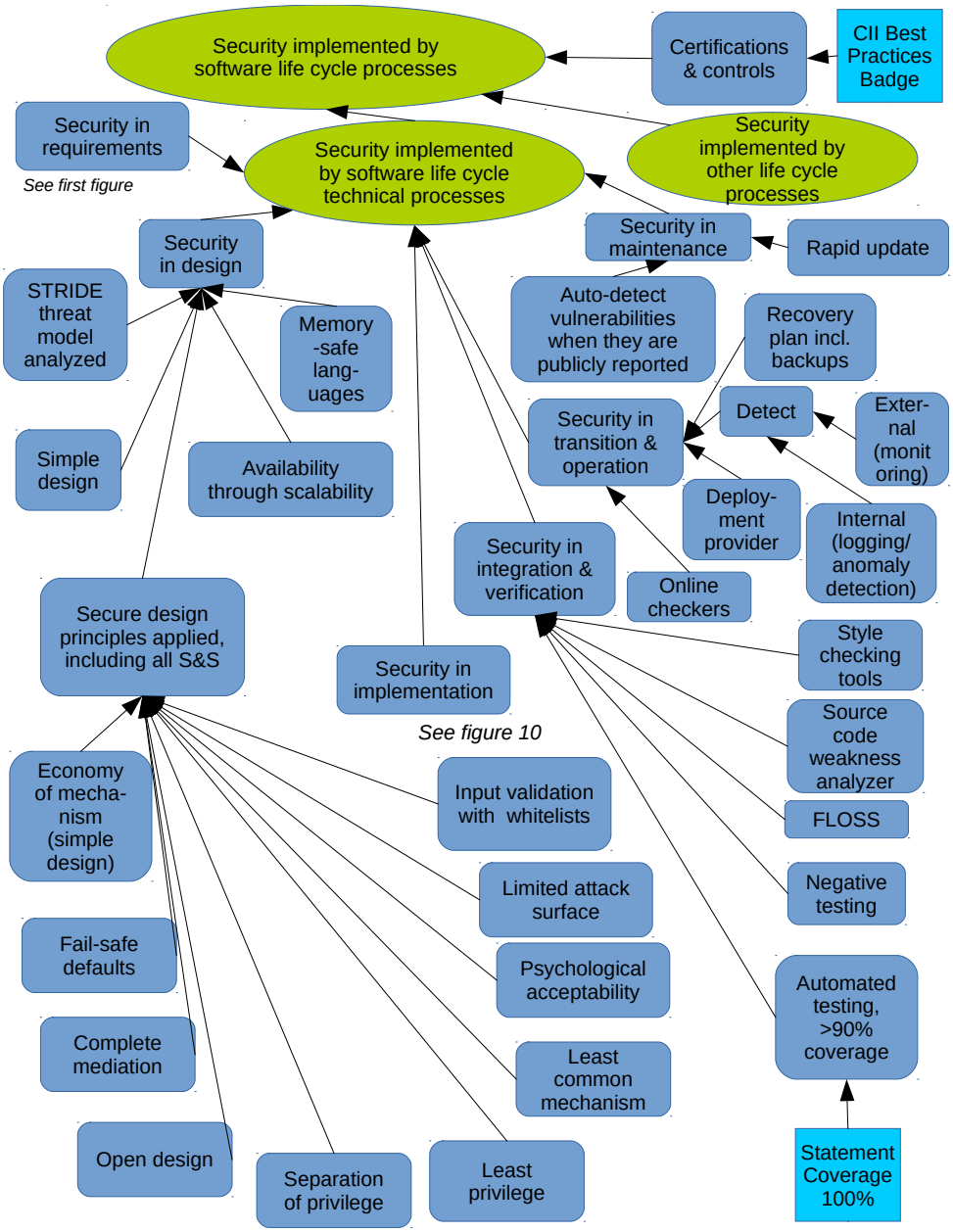

Figure 9. Application: Life Cycle Processes

Secure design principles were also applied, in this case, the well-known principles from Saltzer and Schroeder (S&S). For more about the S&S principles, see [Saltzer 1975]. The actual assurance case discussed "simple design" separately, and noted that it was an S&S principle. As noted earlier, we have supplemented the S&S principles with two more: having a limited attack surface and using input validation with whitelists.

The example system uses memory-safe programming languages, in this case Ruby. This eliminates the risk of using memory-unsafe languages in custom code as discussed in section 2.B.1.

Of course, we must verify the software. This system has an automated test suite. The project's policy is to require at least 90% statement coverage, but in practice it has 100% statement coverage, and its test suite includes "negative tests" to ensure that important security-related actions that should fail will actually fail.

Many systems must acquire certifications and accreditations. The example system has acquired the Core Infrastructure Initiative (CII) Best Practices badge (that is, it has earned the very badge it also manages).

We now turn to the part of the assurance case that focuses on implementation.

C. Implementation

Of course, if the software as implemented is not secure, the system it controls is unlikely to be secure. Figure 3 shows the portion of the assurance case focusing on the software implementation.

The best list of common implementation errors that lead to vulnerabilities would be a large set from that specific system built up over decades of time, but this is rarely available. This system uses a common alternative for web applications, the OWASP Top 10, a widely vetted list of the "top 10" implementation defects that lead to security vulnerabilities. In fact, it uses a combination of the 2013 and 2017 versions, for a total of 13 issues. When it was developed, the 2017 version did not exist, and there are advantages to covering both sets.

Misconfiguration errors are typically specific to a programming language, framework, and/or platform. There are already guidance documents for many common ones; web searches can often quickly find some possibly relevant guides.

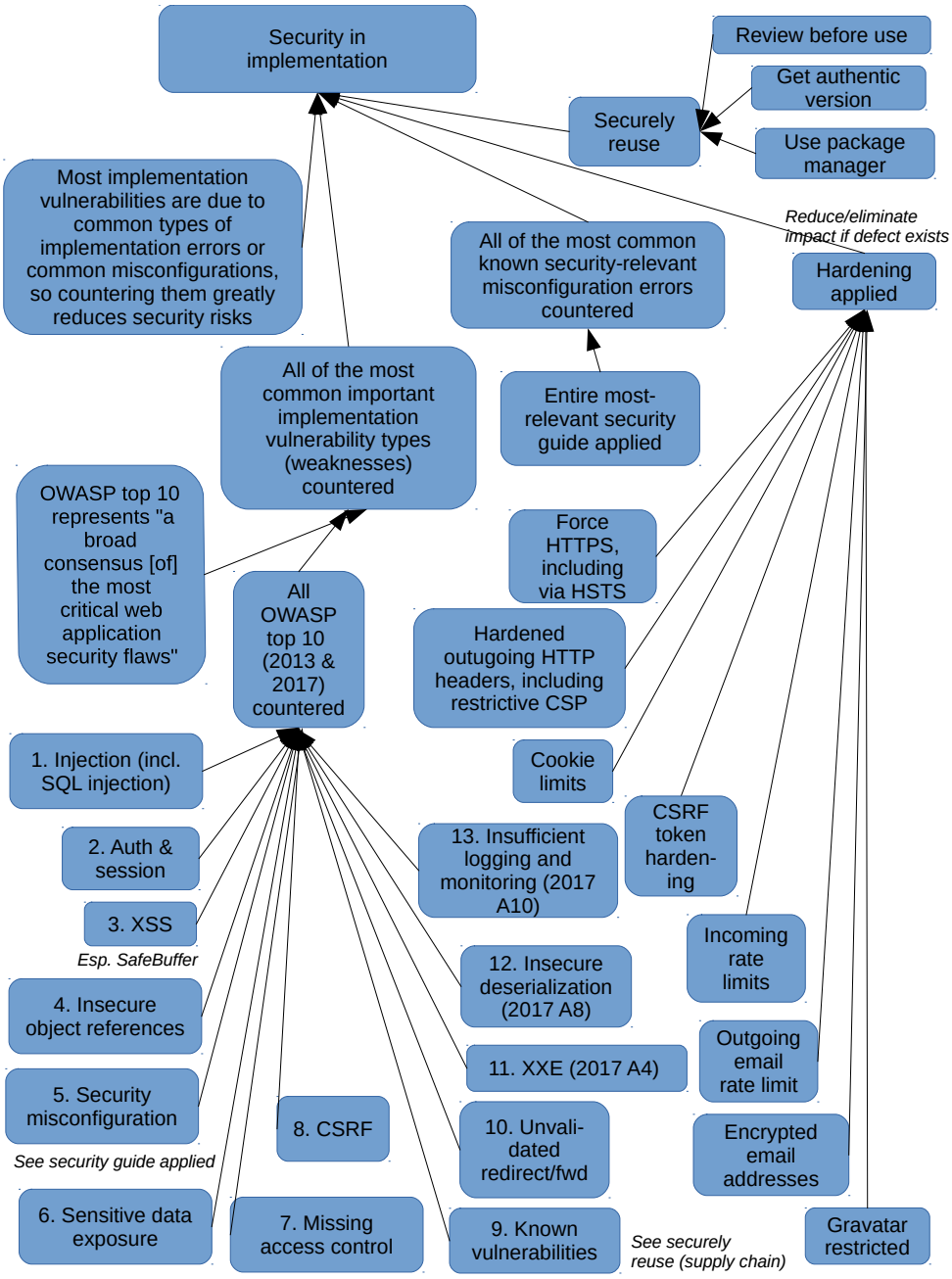

Figure 10. Application: Implementation

As stated earlier (and restated here for emphasis), hardening measures are extremely important for reducing the probability or impact of risks. In a web application, measures such as using a restrictive CSP and incoming rate limits can be very useful.

In this particular example, we have some specialized hardening titled “Gravatar restricted.” This particular system uses the “Gravatar” service (this service provides an avatar image given an email address’ MD5 hash). To provide additional protection of email

addresses, the system only provides links to the Gravatar service when the user has an image there. See the system's assurance case for details; the point is that specialized system-specific hardening measures can be used.

D. Other Life Cycle Processes

We should also consider other software life cycle processes beyond the technical processes. These are the agreement processes, organizational project-enabling processes, and technical management processes. This figure is the same as the pattern shown earlier; we include it here for completeness. Again, for more detail see [Wheeler 2018a].

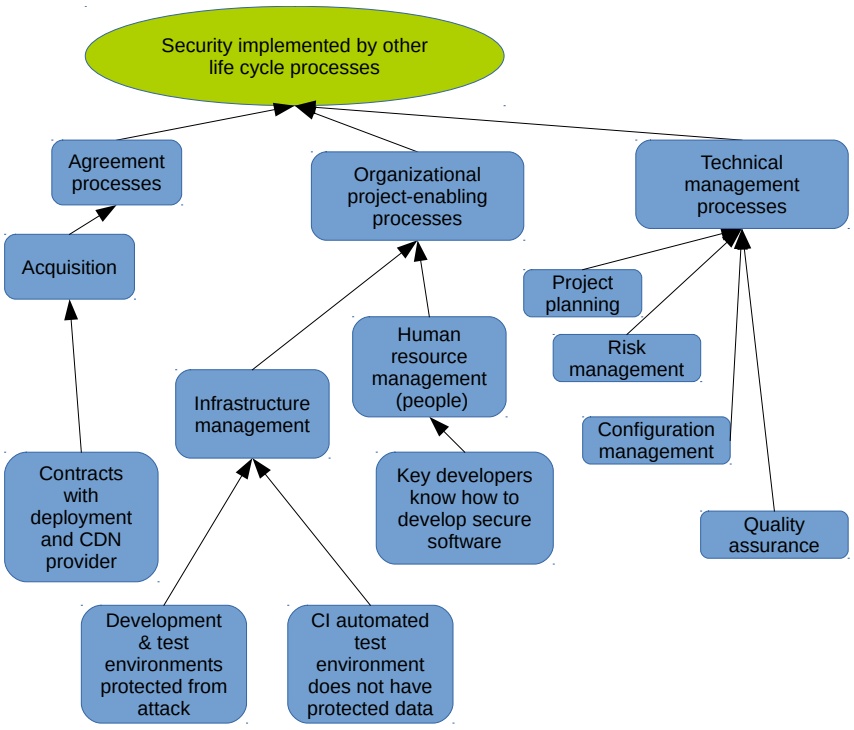

Figure 11. Application: Other Life Cycle Processes

4. Conclusions

We have presented a sample security assurance case pattern along with a specific example. There is no reason to believe that another system's assurance case should look exactly like the pattern shown here. We expect that those who use this pattern will modify it as necessary to fit their system.

You may or may not agree that the sample assurance case is appropriate for the sample system—but that is not the point. Instead, the point is to develop an assurance case for the system you are responsible for, so that all relevant stakeholders can understand what will be done, and eventually agree that it is (or is not) sufficient.

An assurance case provides a structured approach so that important decisions can be made within their full context. In the past, some have had trouble developing a security assurance case because of the lack of a pattern or limited public examples. We hope that this pattern, coupled with a public example of its application, will help those trying to develop a security assurance case for their system.

Appendix A.
Processes Are Neither Phases nor Stages

It's important to clarify that the word "process" has a different meaning from the word "phase." A "phase" is simply a period of time, and each phase is typically associated with a major decision point. A common synonym for phase is "stage." In contrast, a "process" is a "set of interrelated or interacting activities that transforms inputs into outputs" [ISO 12207:2017].

The processes that occur and reoccur throughout the life cycle are typically performed simultaneously and in parallel, iterating and feeding back as appropriate. Processes typically recur within multiple or all phases; for example, the systems/software requirements definition process typically recurs in every phase or stage of a system's development and maintenance, because there is always a need to capture requirements (including changed requirements). These attributes make processes fundamentally different from phases or stages. Processes *can* be done in a strict order. For example, in a strict waterfall model, each process is done to completion in a strict sequence. However, as noted in [Royce 1970], this strict waterfall approach "is risky and invites failure." In practice, processes reoccur throughout the life cycle.

In contrast, phases (aka stages) are often done in a specific order. For example, DoD Instruction 5000.02 defines a standard structure of phases: the material solution analysis phase, technology maturation and risk reduction (TMRR, aka "pre-milestone B") phase, the engineering and manufacturing development (EMD) phase, the production and deployment phase, and the operations and support phase. These phases can be eliminated and combined, and other tailoring is possible [DoDI 5000.02]. Again, note the contrast: The same process (e.g., systems/software requirements definition or verification) may occur in many (or all) phases.

Figure 12 visually illustrates the contrast between phases and processes. The top part shows the DoD Instruction 5000.02 standard structure of phases [DoD 5000.02]. The bottom part shows the life cycle processes as defined by ISO/IEC/IEEE 15288:2015 [ISO 15288:2015] and ISO/IEC/IEEE 12207:2017 [ISO 12207:2017]; the technical processes are directly identified, whereas the three other process groups are summarized due to space limitations. The various smaller arrows illustrate that processes accept inputs from other processes, and generate results to other processes, in a deeply and fundamentally intertwined way. The larger arrows illustrate that most processes typically reoccur in all or

nearly all phases. Note that these processes do not *define* a life cycle; instead, life cycle processes occur and re-occur *within* the life cycle of a system or of software.

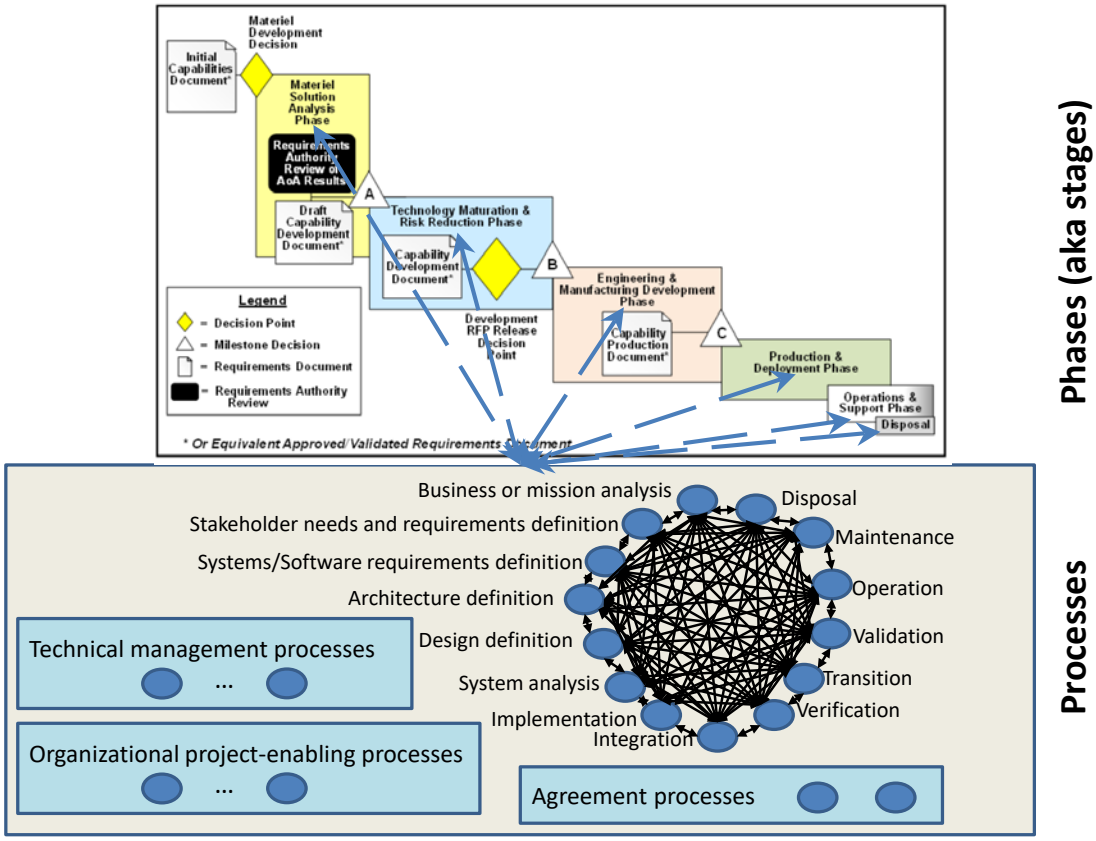

Figure 12. Phases vs. Processes

We emphasize the difference between phases and processes because many parts of this document are structured around processes, not phases. When discussing the major decision points in system acquisition it is useful and appropriate to focuses on phases. However, we find that processes are a better way to organize and discuss the specific assurance activities that occur within software development and sustainment. For example, if we organized this document by phase, we would need to repeat the same material for each phase, which would be repetitive and confusing. In addition, a phase-based structure would be hard to apply to all software, because not all systems use the same set of phases. Even within the DoD, Figure 12 merely shows the DoD Instruction 5000.02 standard structure of phases; in the DoD, these phases can be eliminated and combined, and other tailoring is also possible [DoDI 5000.02]. By focusing on the processes, we can avoid repetition, reduce confusion, and address assurance issues for all software development and maintenance projects.

This document is not the only document organized by processes and not by phases. The National Defense Industrial Association (NDIA)'s *Engineering for System Assurance* [NDIA 2008] is organized in exactly the same way (by process) for essentially the same

reasons. The ISO/IEC/IEEE 15288:2015 and ISO/IEC/IEEE 12207:2017 standards also focus on defining processes within a life cycle, not phases.

Appendix B.
How an Assurance Case can Support Other Documents and Processes

Security assurance cases are often *not* directly required for a project.[9] For example, DoD Instruction 5000.02 [DoD 5000.02] does not directly require the creation of an assurance case. Instead, an assurance case is a conceptual tool that can help a project that develops and sustains a system, including helping that project develop the system and documentation that they *are* required to do. Some documents do expressly describe their relationship with assurance cases, such as [NDIA 2008], [ISO 12207], [NIST SP 800-53A], and [NIST SP 800-160], but others do not.

This chapter shows how a security assurance case can support some other documents and processes. When an assurance case is directly discussed, we note that. In addition, we show examples of how a security assurance case can support that document. We will not try to show all ways that an assurance case can support those other documents and processes, as that would be voluminous. After all, an assurance case provides an integrated view of many other materials, so in practice there would often be many connections. Instead, we focus on some important or illustrative examples. Remember: An assurance case is simply a tool. The end goal is to develop and maintain an adequately secure system and to show that it is adequate, not to create an assurance case.

Below, we identify a few documents (many of which imply certain processes) and a few examples of how a security assurance case can support each of those documents. We particularly emphasize DoD-related documents.[10]

1. DoD Instruction 5000.02

DoD Instruction 5000.02 [DoDI 5000.02] provides instructions for the "management of all [covered] acquisition programs." Enclosure 11 requires initiation of the Cybersecurity RMF as early as possible (per DoD Instruction 8510.01) and a Cybersecurity

[9] A project's contract could expressly require an assurance case for security, including its supporting evidence, and even cite the ISO/IEC 15026 standard as additional information. However, such contracts are rare today.

[10] A useful source for official DoD instructions is http://www.esd.whs.mil/Directives/issuances/dodi/

strategy as an appendix to the Program Protection Plan (PPP). Enclosure 14 also notes the RMF and PPP. We will discuss those other materials separately.

The purpose of DoD Instruction 5000.02 enclosure 14 is to address "Cybersecurity in the Defense Acquisition System." Here are some of its requirements, and how an assurance case can support them:

- 1a(1) says, "Cybersecurity is a requirement for all DoD programs and must be fully considered and implemented in all aspects of acquisition programs across the life cycle." An assurance case (especially when organized with this pattern) can make it easy to consider and implement assurance across the life cycle and also demonstrates it by showing in a structured way how assurance is fully considered.
- 3b(1)(c) says, "Use requirements derivation methods, such as system modeling and analysis, security use and abuse or misuse cases, criticality analysis, and vulnerability analysis to determine cybersecurity requirements that are sufficient to minimize vulnerabilities introduced by design, implementation, system interfaces, and access points." An assurance case is a requirements derivation method to determine that cybersecurity requirements are sufficient to minimize vulnerabilities introduced by other factors.
- 3b(2) says, "Allocate cybersecurity and related system security requirements to the system architecture and design, and assess for vulnerabilities." An assurance case aids analysis of the allocation of security requirements to the architecture and design to enable assessment for vulnerabilities.
- 3b(4) says, "(4) Include cybersecurity and related system security in the conduct of technical risk management activities and change management processes to address risk identification, analysis, mitigation planning, mitigation implementation, and tracking."
- 3b(5) says, "Use evolving program and system threat assessments to continuously assess cybersecurity risks to the program and system." As information about attackers is gathered, the assurance case can provide a unifying source of information to enable continuous assessments of risks.

2. DoD Program Protection Plan (PPP)

The outline of the DoD program protection plan (PPP) [PPP 2011] is intentionally simple and streamlined:

- Section 2.2: The "CPI and Critical Components Countermeasure Summary" table maps CPI and components to countermeasures; the assurance case can

identify the countermeasures, as well as which ones are being applied to the CPI and components, and show *why* they needed.

- Section 5: This "Threats, Vulnerabilities, and Countermeasures" section can be summarized from the assurance case.
- Section 5.2 asks "How will identified vulnerabilities be mitigated?" The PPP can provide a brief summary; the assurance case can justify this, and drill down to show why the mitigations will be adequate.
- Section 5.3 (Countermeasures) has various questions. Its question, "How will countermeasures be selected to protect CPI and critical functions/components?" can be answered by "analysis and approval of the assurance case justifying their adequacy." It also asks to "Succinctly describe the implementation of each countermeasure used to protect CPI and critical functions and components." Again, this can be summarized from the assurance case.
- Section 5.3.3 (Software Assurance) also asks various questions.
  - The question, "How will software be designed and tested to assure protection of critical functionality and CPI?" can be answered by walking through those portions of the assurance case.
  - The question, "How will software architectures, environments, designs, and code be evaluated with respect to CVE (Common Vulnerabilities and Exposures), CAPEC (Common Attack Pattern Enumeration and Classification), and CWE (Common Weakness Enumeration)?" can be directly answered through the assurance case. If you follow the suggested pattern, the CVEs can be addressed through "securely reuse" (especially review before use) before selection and the maintenance item "auto-detect vulnerabilities when they are publicly reported" after that. CAPEC is at least addressed by considering the threat, and can be more thoroughly addressed by performing the item "threat (attack) model analyzed" with a view towards attack processes. The security in implementation section directly addresses CWEs.

3. DoD Cybersecurity Strategy

"The Clinger-Cohen Act (40 U.S.C. Subtitle III) in the 2001 NDAA §811(P.L. 106-398), DoDI 5000.02, Operation of the Defense Acquisition System, and DoDI 8500.01, Cybersecurity, set policy to ensure programs have a strategy to implement cybersecurity and manage associated risks…. All Acquisition programs acquiring systems containing information technology are required to develop and maintain a Cybersecurity Strategy… the Cybersecurity Strategy is a required acquisition program document created and

maintained by the Program Office and appended to the Program Protection Plan (PPP)." [Cybersecurity Strategy 2015].

As with the PPP, this is a summary document; it is presumed that more detailed analysis occurs elsewhere. Here are some examples of connections:

- Section III (Cybersecurity Approach), A (Management Approach), 1 (Stakeholder Communication and Documentation) says, "Describe methods and periodicity of communication between program and AO/AODR, including the communication of risks and changes affecting risk posture. Describe how the program will plan for stakeholder input (e.g. Integrated Product Teams (IPT), working groups) and plan for assembly, dissemination, and coordination of required documentation including documentation of cybersecurity risks." An assurance case can be a key integrator of that documentation, and the way it is maintained can answer the other questions.
- Section III (Cybersecurity Approach), B (Technical Approach), 3 (Risk Assessments) says, "Describe [the] plan for periodic RMF risk assessments (including periodicity, stakeholders, and methodology); Describe how they will be integrated with other risk assessment activities, including TSN Analysis (including criticality analysis), programmatic risk assessments, and operational testing." An assurance case can be mechanism for integrating the discussion and results of these various activities.

4. DoD Instruction 5200.44

DoD Instruction 5200.44 is the policy on the "Protection of Mission Critical Functions to Achieve Trusted Systems and Networks (TSN)" [DoDI 5200.44]. This is a high-level policy, and there are many ways to achieve its ends. Here are a few of its points, and their possible connections to an assurance case:

- 4a says, "Mission critical functions and critical components within applicable systems shall be provided with assurance consistent with criticality of the system, and with their role within the system." A security assurance case can help provide confidence that the functions and critical components are provided with assurance consistent with their criticality and role, as it shows *why* the assurance provided is believed to be adequate.
- 4c says, "Risk to the trust in applicable systems shall be managed throughout the entire system lifecycle [sic]." An assurance case can show that assurance is being managed throughout the system life cycle and how that is being done.

- 4c(1) says, “Reduce vulnerabilities in the system design through system security engineering.” An assurance case can show how the system design reduces vulnerabilities.

5. NIST Cybersecurity Risk Management Framework (RMF) / DoDI 8510.01

The NIST Cybersecurity Risk Management Framework (RMF) expressly discusses assurance cases in [NIST SP 800-53A]. We first provide a brief overview and then briefly discuss that material.

NIST Special Publication 800-37 [NIST SP 800-37] was developed to transform “the traditional Certification and Accreditation (C&A) process into the six-step Risk Management Framework (RMF).” Three of its key steps are to select, implement, and assess security controls. It also states that “Information security requirements are satisfied by the selection of appropriate management, operational, and technical security controls from NIST Special Publication 800-53.”

NIST Special Publication 800-53 [NIST 800-53] provides “guidelines for selecting and specifying security controls for organizations and information systems.” In particular, it provides a “security controls catalog,” a list of many potential security controls organized into families. No single set of security controls would be appropriate to all systems. NIST SP 800-53 addresses this by identifying baseline controls, which are “the starting point for the security control selection process.” NIST SP 800-53 identifies three security control baselines “corresponding to the low-impact, moderate-impact, and high-impact information systems.”

DoD Instruction 8510.01, Risk Management Framework (RMF) for DoD Information Technology (IT), establishes “the RMF for DoD IT” [DoDI 8510.01]. However, the DoD and intelligence community use a finer-grained approach to selecting controls. As described in Committee on National Security Systems (CNSS) Instruction 1253, system requirements are divided into confidentiality, integrity, and availability, and for each division the impact is selected as being low, medium, or high. These values then determine the recommended set of baseline controls. [CNSSI 1253]

NIST SP 800-53 expressly defines the term “assurance case” (as “A structured set of arguments and a body of evidence showing that an information system satisfies specific claims with respect to a given quality attribute.”). However, much more discussion about using assurance cases with the RMF is included in its companion document NIST SP 800-53A [NIST SP 800-53A].

NIST SP 800-53A discusses an assurance case as a fundamental construction. Its Section 2.3 discusses what an assurance case is and its relationship to the RMF, saying, “Building an effective assurance case for security and privacy control effectiveness is a

process that involves (i) compiling evidence from a variety of activities conducted during the system development life cycle… [and] (ii) presenting this evidence in a manner that decision makers are able to use effectively in making risk-based decisions about the operation or use of the system. The evidence described above comes from the implementation of the security and privacy controls in the information system and inherited by the system (i.e., common controls) and from the assessments of that implementation. Ideally, the assessor is building on previously developed materials that started with the specification of the organization's information security and privacy needs and was further developed during the design, development, and implementation of the information system. These materials, developed while implementing security and privacy throughout the life cycle of the information system, provide the initial evidence for an assurance case…"

6. NIST SP 800-160 volume 1

NIST SP 800-160 volume 1 (Systems Security Engineering: Considerations for a Multidisciplinary Approach in the Engineering of Trustworthy Secure Systems) [NIST SP 800-160] is a guide that specifically focuses on the use and importance of an assurance case. It has several purposes, including the following:

- "[to] provide a basis to formalize a discipline for systems security engineering in terms of its principles, concepts, and activities."
- "[to] provide considerations and to demonstrate how systems security engineering principles, concepts, and activities can be effectively applied to systems engineering activities"

[NIST SP 800-160] is built in part on its "system security engineering framework" which is divided into three parts (see the illustration from that document in Figure 13):

- "The **problem context** defines the basis for an acceptably and adequately secure system…"
- "The **solution context** transforms the stakeholder security requirements into design requirements for the system; addresses all security architecture, design, and related aspects necessary to realize a system that satisfies those requirements; and produces sufficient evidence to demonstrate that those requirements have been satisfied."
- "The **trustworthiness context** is a decision-making context that provides an evidence-based demonstration, through reasoning, that the system-of-interest is deemed trustworthy based upon a set of claims derived from security objectives. The trustworthiness context consists of:
  - Developing and maintaining the assurance case; and
  - Demonstrating that the assurance case is satisfied."

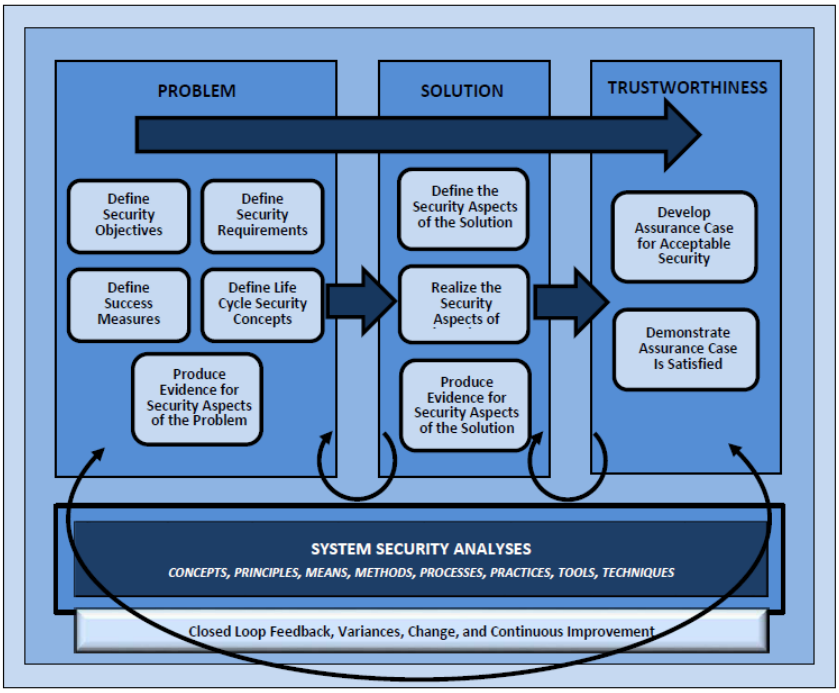

Figure 13. Systems Security Engineering Framework of NIST SP 800-160

It also notes, "The trustworthiness context is grounded on the concept of an assurance case. An assurance case is a well-defined and structured set of arguments and a body of evidence showing that a system satisfies specific claims with respect to a given quality attribute… An assurance case is used to demonstrate that a system exhibits some complex emergent property such as safety, security, resiliency, reliability, or survivability. An effective security assurance case contains foundational security claims that are derived from stakeholder security objectives, credible and relevant evidence that substantiates the claims, and valid arguments that relate the various evidence to the supported security claims. The result provides a compelling statement that adequate security has been achieved and driven by stakeholder needs and expectations."

7. ISO/IEC/IEEE 12207

ISO/IEC/IEEE 12207 [ISO 12207:2017] specifically discusses its relationship with a security assurance case and assurance cases for any purpose:

- Section E.6 (process view for software assurance) "provides an example of applying the process viewpoint to yield a process view for software assurance… The software assurance characteristics, their extent of achievement, and related information may support a software assurance claim, as described in ISO/IEC/IEEE 15026." Later in the same section, it notes that "the Measurement process (6.3.7), in its entirety, provides a common platform for collecting information about the software assurance claims, strategies, and evidence, sometimes referred to as an assurance case."

- Section F.3 notes that “Other necessary models can include some of these characteristics [of importance to stakeholders]... a software assurance case, regarded as a model, can help in deducing potential architectural mitigations to minimize operational risks (mission loss due to exploited security vulnerabilities) related to critical concerns and functions.”

This standard defines a set of processes (not phases or stages). As it notes, “implementation of this document typically involves selecting and declaring a set of processes suitable to the organization or project. There are two ways that an implementation can be claimed to conform to the provisions of this document — full conformance and tailored conformance… Claiming ‘full conformance to tasks’ asserts that all of the requirements of the activities and tasks of the declared set of processes are achieved. Alternatively, claiming ‘full conformance to outcomes’ asserts that all of the required outcomes of the declared set of processes are achieved.” Here are some examples where an assurance case can support 12207, organized by its processes:

- System analysis process:
  - Key outcomes are that “system analysis assumptions and results are validation” and that “system analysis results are provided for decisions.” A good assurance case clearly identifies assumptions, and results for validation can be provided for decisions.
  - The key activity “perform system analysis” is supported by “review the analysis results for quality and validity” and “record the results of system analysis.” An assurance case is itself an analysis, and it provides a way to organize and record the results of other analyses.
- Risk management process:
  - A key outcome is “appropriate treatment is implemented.” An assurance case can show what depends on the treatments (and therefore that they are appropriate).
  - Similarly, a key activity is to “treat risk” with the task “implement risk treatment alternatives for which the stakeholders determine that actions should be taken to make a risk acceptable.” An assurance case enables stakeholders to discuss and make that determination so that they can gain agreement on what actions should be taken.

ISO/IEC/IEEE 15288:2015, aka 15288, “establishes a common framework of process descriptions for describing the life cycle of systems created by humans” [ISO 15288:2015]. Because 12207 uses the same processes as 15288, the examples shown here generally apply to 15288 as well.

Appendix C.
Trusted Systems and Networks (TSN) Analysis

Trusted Systems and Networks (TSN) analysis is a rigorous approach to identifying addressing and countering risks, including those from malicious components, as presented in the Defense Acquisition Guidebook (DAG) [DoD DAG] in support of [DoDI 5200.44]. This approach can be especially valuable for critical systems. TSN analysis includes criticality analysis (CA), which identifies mission critical functions and critical components. It also includes vulnerability assessment, risk assessment, and protection measure selection. This appendix provides a brief summary of the approach.

A rigorous approach to identifying addressing and countering risks, including those from malicious components, is presented in [DoDI 5200.44] and the Defense Acquisition Guidebook (DAG) [DoD DAG]. This approach can be especially important for critical systems, and is also called "Trusted Systems and Networks (TSN) Analysis." As this approach especially impacts design, we have placed this as part of the design process within this assurance case pattern. It involves CA (which identifies mission critical functions and critical components), vulnerability assessment, risk assessment, and protection measure selection.

This approach begins with CA: an "end-to-end functional decomposition performed by systems engineers to identify mission critical functions and components. Includes identification of system missions, decomposition into the functions to perform those missions, and traceability to the hardware, software, and firmware components that implement those functions. Criticality is assessed in terms of the impact of function or component failure on the ability of the component to complete the system mission(s)" [DoDI 5200.44]. CA identifies mission critical functions and critical components; a critical component "is or contains information and communications technology (ICT), including hardware, software, and firmware, whether custom, commercial, or otherwise developed, and which delivers or protects mission critical functionality of a system or which, because of the system's design, may introduce vulnerability to the mission critical functions of an applicable system" [DoDI 5200.44].

Of course, just identifying mission critical functions and critical components (and thus the consequence of loss) is not enough. The system should typically be changed depending on the likelihood of loss and trade-offs that consider cost. Chapter 9 of the he DAG presents an engineering-driven approach for doing this, which is titled "Trusted

Software and Networks (TSN) Analysis Methodology," as illustrated in Figure 14 [DoD DAG, chapter 9].

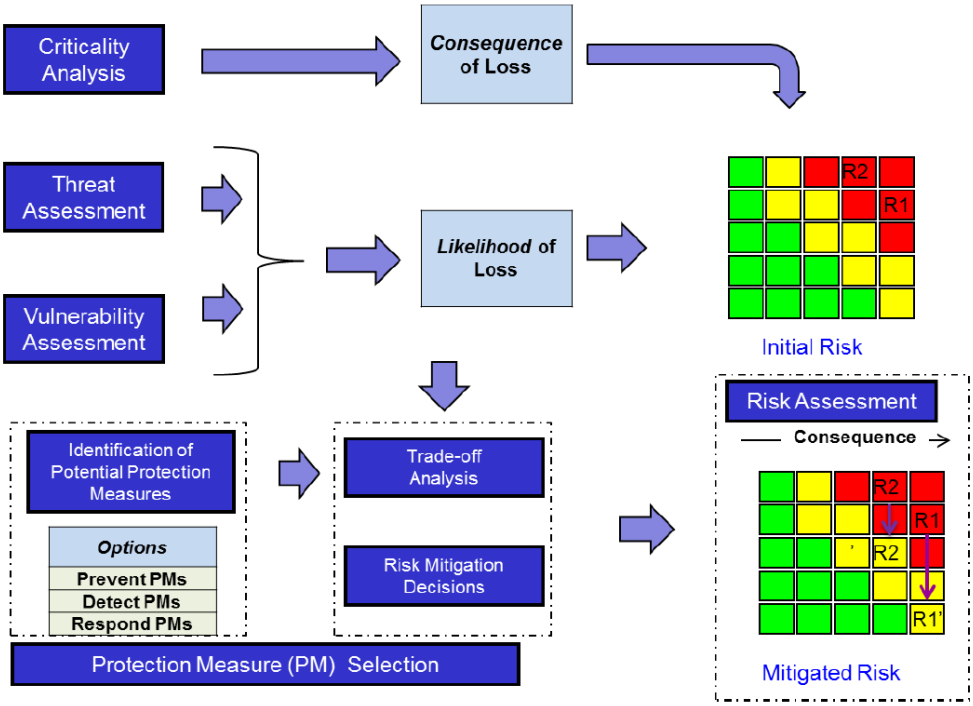

Figure 14. TSN Analysis Methodology [DoD DAG, chapter 9]

Vulnerability assessment searches for vulnerabilities, beginning with the mission-critical functions and associated critical components. Its results, along with a threat assessment, help determine the likelihood of loss. This likelihood of loss, combined with the consequence of loss, can then be used to identify potential protection measures and perform trade-off analysis of those measures. Risk mitigation decisions can then be made to produce a mitigated risk.

For more information, see [DoD DAG] and [DoDI 5200.44].

References

[Blanchette 2009a] Blanchette, Jr., Stephen. 2009. "Assurance Cases for Design Analysis of Complex System of Systems Software." *Proceedings of the AIAA Infotech@Aerospace Conference*. American Institute of Aeronautics and Astronautics (AIAA). https://arc.aiaa.org/doi/abs/10.2514/6.2009-1921

[Blanchette 2009b] Blanchette, Jr., Stephen. 2009. "Assurance Cases for Design Analysis of Complex System of Systems Software." *Proceedings of the AIAA Infotech@Aerospace Conference*. American Institute of Aeronautics and Astronautics (AIAA). https://resources.sei.cmu.edu/asset_files/Presentation/2009_017_001_22469.pdf

[Bojanova 2016] Bojanova, Irena, Paul E. Black, Yaacov Yesha, and Yan Wu. 2016. "The Bugs Framework (BF): A Structured Approach to Express Bugs." *Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. https://ieeexplore.ieee.org/document/7589797

[CAS 2012] National Security Agency (NSA) Center for Assured Software (CAS). 2012. *CAS Static Analysis Tool Study – Methodology*. http://samate.nist.gov/docs/CAS%202012%20Static%20Analysis%20Tool%20Study%20Methodology.pdf.

[CNSS 1253] Committee on National Security Systems (CNSS). n.d. *Security Categorization and Control Selection for National Security Systems*. http://www.dss.mil/documents/CNSSI_No1253.pdf

[CNSS 4009] Committee on National Security Systems (CNSS). n.d. *National Information Assurance Glossary. CNSS Instruction No. 4009*.

[CWE 2017] Common Weakness Enumeration (CWE). 2017. *Scoring CWEs*. https://cwe.mitre.org/scoring/index.html

[Cybersecurity Strategy 2015] Department of Defense. 2015. *Outline and Guidance for Acquisition Programs' Cybersecurity Strategies*. https://www.dau.mil/cop/cybersecurity/dau%20sponsored%20documents/dcio%20cybersecurity%20strategy%20memo%2010nov15.pdf

[DoD CTT] Department of Defense, Deputy Assistant Secretary of Defense Developmental Test and Evaluation (DT&E). *The Department of Defense Cyber Table Top Guidebook.* Version 1.0. July 2, 2018. https://www.acq.osd.mil/dte-trmc/docs/The%20DoD%20Cyber%20Table%20Top%20Guidebook%20v1.pdf

[DoD DAG] Department of Defense. *Defense Acquisition Guidebook* (DAG). https://www.dau.mil/tools/dag

[DoDI 5000.02] Department of Defense. 2015. "DoD Instruction (DoDI) 5000.02: Protection of Mission Critical Functions to Achieve Trusted Systems and Networks

(TSN)," (Incorporating Change 3, August 10, 2017). http://www.esd.whs.mil/Portals/54/Documents/DD/issuances/dodi/500002_dodi_2015.pdf?ver=2017-08-11-170656-430

[DoDI 5200.44] Department of Defense. 2012. "DoD Instruction 5200.44: Protection of Mission Critical Functions to Achieve Trusted Systems and Networks (TSN)," (Incorporating Change 3, October 15, 2018). https://www.acq.osd.mil/se/initiatives/init_pp-sse.html

[DoDI 8510.01] Department of Defense. 2014. "DoD Instruction 8510.01: Risk Management Framework (RMF) for DoD Information Technology (IT)," (Incorporating Change 2, July 28, 2017). http://www.esd.whs.mil/Portals/54/Documents/DD/issuances/dodi/851001_2014.pdf

[Goodenough 2014] Goodenough, John, Howard F. Lipson, and Charles B. Weinstock. 2014. *Arguing Security - Creating Security Assurance Cases*. https://www.us-cert.gov/bsi/articles/knowledge/assurance-cases/arguing-security-creating-security-assurance-cases

[ISO 12207:2017] ISO/IEC/IEEE. 2017. "ISO/IEC/IEEE 12207:2017, Systems and software engineering—Software life cycle processes." https://www.iso.org/standard/63712.html

[ISO 15026-2:2011] ISO. 2011. "ISO 15026-2:2011, Systems and software engineering—Systems and software assurance—Part 2: Assurance case." https://www.iso.org/standard/52926.html

[ISO 15288:2015] ISO/IEC/IEEE. 2015. "ISO/IEC/IEEE 15288:2015, Systems and software engineering—System life cycle processes." https://www.iso.org/standard/63711.html

[Lipson 2014] Lipson, Howard F. and Charles B. Weinstock. 2014. *Evidence of Assurance: Laying the Foundation for a Credible Security Case*. https://www.us-cert.gov/bsi/articles/knowledge/assurance-cases/evidence-assurance-laying-foundation-credible-security-case

[MITRE 2017] MITRE. January 18, 2017. *Enumeration of Technical Impacts*. https://cwe.mitre.org/cwraf/enum_of_ti.html

[MITRE 2018] MITRE. April 2, 2018. *Prioritizing Weaknesses Based Upon Your Organization's Mission*. https://cwe.mitre.org/community/swa/priority.html

[Moore 2001] Moore, Andrew P, Robert J. Ellison, and Richard C. Linger. 2001. "Attack Modeling for Information Security and Survivability" (Technical Note CMU/SEI-2001-TN-001). https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=5417

[NDIA 2008] National Defense Industrial Association (NDIA) System Assurance Committee. 2008. *Engineering for System Assurance*. https://www.acq.osd.mil/se/docs/sa-guidebook-v1-oct2008.pdf

[NIST SP 800-37] NIST. Updated 2014-06-05 (original February 2010). NIST Special Publication 800-37 Rev. 1: Guide for Applying the Risk Management Framework to Federal Information Systems: a Security Life Cycle Approach. https://csrc.nist.gov/publications/detail/sp/800-37/rev-1/final

[NIST SP 800-39] NIST. 2011. *NIST Special Publication (SP) 800-39: Managing Information Security Risk. 2011-03*. http://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-39.pdf

[NIST SP 800-53] NIST. 2015. *NIST Special Publication (SP) 800-53 rev 4: Security and Privacy Controls for Federal Information Systems and Organizations* (Includes updates to 2015-01-22). https://csrc.nist.gov/publications/detail/sp/800-53/rev-4/final

[NIST SP 800-53A] NIST. 2014. NIST Special Publication (SP) 800-53A rev 4. Assessing Security and Privacy Controls in Federal Information Systems and Organizations: Building Effective Assessment Plans. https://csrc.nist.gov/publications/detail/sp/800-53a/rev-4/final

[NIST SP 800-160] Ross, Ron, Michael McEvilley, and Jane Carrier Oren. 2016. NIST Special Publication (SP) 800-160. Systems Security Engineering: Considerations for a Multidisciplinary Approach in the Engineering of Trustworthy Secure Systems, volume 1 (Updated 2018-03-23). https://csrc.nist.gov/publications/detail/sp/800-160/vol-1/final

[OMG 2018] Object Management Group (OMG). March 2018. *Structured Assurance Case Metamodel (SACM)*. https://www.omg.org/spec/SACM/2.0

[Open Group 2013] The Open Group. July 2013. *The Dependability through Assuredness™ (O-DA) Framework*. https://publications.opengroup.org/c13f

[OSD 2017] Department of Defense (DoD) Software Assurance (SwA) Community of Practice (CoP) Contract Language Working Group. November 2017. *Incorporating Software Assurance into Department of Defense Acquisition Contracts*. https://www.acq.osd.mil/se/docs/2017-11-15-SwA-Contracts.pdf

[PPP 2011] Deputy Assistant Secretary of Defense, Systems Engineering. July 2011. *Program Protection Plan Outline & Guidance, Version 1.0*. https://www.acq.osd.mil/se/initiatives/init_pp-sse.html

[Rhodes 2010] Rhodes, Thomas, Frederick Boland, Elizabeth Fong, and Michael Kass. 2010. “Software Assurance Using Structured Assurance Case Models.” *Journal of Research of the National Institute of Standards and Technology* 115(3): 209–216. doi: 10.6028/jres.115.013. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4548534/

[Royce 1970] Royce, Winston. 1970. Managing the Development of Large Software Systems.

[Saltzer 1975] Saltzer, Jerome “Jerry” H., and Michael D. Schroeder. September 1975. “The Protection of Information in Computer Systems.” *Proceedings of the IEEE,* 63(9):1278–1308,

https://www.acsac.org/secshelf/papers/protection_information.pdf or https://ieeexplore.ieee.org/document/1451869/

[Wheeler 2016] Wheeler, David A. and Amy H. Henninger. November 2016. *State-of-the-Art Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation* (IDA Paper P-8005). https://www.acq.osd.mil/se/docs/P-8005-SOAR-2016.pdf

[Wheeler 2017] Wheeler, David A. September 18, 2017. "How to Develop Secure Applications: The BadgeApp Example." (Video). https://www.youtube.com/watch?v=5a5D4d6hcEY

[Wheeler 2018a] Wheeler, David A. July 18, 2018. *BadgeApp Security: Its Assurance Case*. Released under the Creative Commons Attribution 3.0 International (CC BY 3.0) license or later. https://github.com/coreinfrastructure/best-practices-badge/blob/master/doc/security.md

[Woody 2018] Woody, Carol, "Automated Decision Support with an Assurance Case," October 17, 2018, *Proceedings of the Development of Trustworthy and Secure Systems*, Worcester, MA.

Acronyms and Abbreviations

ACSE	Assurance and Safety Case Environment
ASLR	Address Space Layout Randomization
CA	Criticality Analysis
CAE	Claims, Arguments and Evidence
CAPEC	Common Attack Pattern Enumeration and Classification
CDN	Content Distribution Network
CI	Continuous Integration
CII	Core Infrastructure Initiative
CNSS	Committee on National Security Systems
CSP	Content Security Policy
CTT	Cyber Table Top
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DAG	Defense Acquisition Guidebook
DoD	Department of Defense
DoDI	Department of Defense Instruction
DoS	Denial of Service
DRY	Don't Repeat Yourself
EMD	Engineering & Manufacturing Development
FLOSS	Free/Libre/Open Source Software
GCM	Galois/Counter Mode
GSN	Goal Structuring Notation
HTTPS	Hypertext Transfer Protocol Secure
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronic Engineers
ICT	Information and Communications Technology
IoT	Internet of Things
ISO	International Organization for Standardization (sic)
KPP	Key Performance Parameter
MITM	Man-in-the-middle
NDAA	National Defense Authorization Act
NDIA	National Defense Industrial Association
NIST	National Institute of Standards and Technology
NVD	National Vulnerability Database
OMG	Object Management Group
OSS	Open Source Software
OWASP	Open Web Application Security Project
P.L.	Public Law
RMF	Risk Management Framework
S&S	Saltzer and Schroeder

SACM	Structured Assurance Case Metamodel
SANS	SysAdmin, Audit, Network and Security
SOAR	State-of-the-Art Resources
SP	Special Publication
SwA	Software Assurance
TMRR	Technology Maturation & Risk Reduction
TSN	Trusted Systems and Networks
URL	Universal Resource Locator

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE (DD-MM-YY)	2. REPORT TYPE	3. DATES COVERED (From – To)
00-12-2018	Final	

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
A Sample Security Assurance Case Pattern	HQ0034-14-D-0001
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBERS

6. AUTHOR(S)	5d. PROJECT NUMBER
David A. Wheeler	AU-5-3856
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES	8. PERFORMING ORGANIZATION REPORT NUMBER
Institute for Defense Analyses 4850 Mark Center Drive Alexandria, VA 22311-1882	P-9278

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR'S / MONITOR'S ACRONYM
Thomas D. Hurt OUSD(R&E) Enterprise Engineering 4800 Mark Center Dr., Suite 16D-08 Alexandria, VA 22350-3600	OUSD(R&E)
	11. SPONSOR'S / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

Project Leader: E. Kenneth Hong Fong

14. ABSTRACT

Essentially all systems with software should address security. However, there is no single "magic bullet" that makes software secure, because security is an emergent property of a system. Tracking and managing the application of the various techniques across the software corpus and throughout the software life cycle can be overwhelming. An assurance case is a widely-recommended practical alternative to other approaches for managing the assurance activities. An assurance case "includes a top-level claim for a property of a system or product (or set of claims), systematic argumentation regarding this claim, and the evidence and explicit assumptions that underlie this argumentation." [ISO 15026-2:2011]. Since an assurance case is systematic, it is much easier for people to determine if important areas have been adequately covered, and to understand the ramifications of different decisions. Maintaining an assurance case for security properties (a "security assurance case") is a simple idea, but many have found it difficult to create a security assurance case because of the limited number of sample patterns and worked examples. This document provides a sample security assurance case pattern, based on a publicly-available assurance case of a real commercial system. This document also shows how this pattern can be applied to a real system. We hope that many system/software developers and approving authorities will find this sample pattern and application to be a useful place to start when developing their own assurance cases. This document also discusses changes that could be made to deal with different kinds of applications, such as Internet of Things (IoT) or weapon systems. The sample security assurance case pattern provided here is for a system that only requires moderate assurance; higher levels of assurance would call for more rigor. This pattern can make it much easier to create a security assurance case.

15. SUBJECT TERMS

Security, assurance, software assurance, secure software, trust, trustworthy, trustworthiness, cybersecurity, assurance case, security assurance case, sample assurance case, example, pattern, developing secure software, building security in, stakeholders, ISO 15026, life cycle, assurance case pattern, assurance case guidelines, claims, arguments, evidence, CAE, goal structuring notation, GSN, ISO 12207, risk management framework, weaknesses, vulnerabilities, rationale, Structured Assurance Case Metamodel, design principles, Saltzer and Schroeder, threat model, attack model, OWASP, OWASP top 10, weakness classes, CWE/SANS top 25, continuous integration, hardening, Address Space Layout Randomization, Content Security Policy (CSP), coding guidelines, reuse, supply chain, processes, phases, stages

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Thomas D. Hurt
Unclassified	Unclassified	Unclassified	Unlimited	60	19b. TELEPHONE NUMBER (Include Area Code) 571-372-6129