BROPHEY CONSULTING
Bridging Technology and Business Strategy

TRANSVASIVE
Transparent and Pervasive Security

BEHAVIORAL SECURITY MODELING: FUNCTIONAL SECURITY REQUIREMENTS

By John Benninghoff and Karl Brophey

Abstract

Defining functional security requirements is a key component of Behavioral Security Modeling, a method to improve security through accurately modeling human/information interactions in social terms. The paper proposes a practical, SDLC agnostic method for gathering functional security requirements by establishing limits on interactions through a series of questions to identify, clarify, and uncover hidden constraints. Five categories of constraints are presented, along with advice and "requirement patterns" to facilitate discussions with stakeholders and translate business needs into unambiguous security requirements. General advice on improving constraints, implementation considerations, security actions, quality assurance, and documenting post conditions are also discussed.

Version 1.0 disclaimer: this white paper attempts to formally capture our collective knowledge on how to effectively define functional security requirements. The next step is to test the theory by implementing the approach in a number of application development environments.

1 Behavioral Information Security and Behavioral Security Modeling

Behavioral Information Security (BIS) is a formal methodology to manage information risk, derived from knowledge of how humans behave and interact with information. It is a new philosophy of security created to address gaps in our current understanding and treatment of people issues in the information security profession. The goals of BIS include developing new methods to address the "people problem", reduce cost, and improve the overall effectiveness of information security.

Behavioral Security Modeling (BSM) was conceived as a way of modeling interactions between information and people in terms of socially defined roles and the expected behaviors of the system being designed. By reducing the difference between the expected system behaviors and the actual system behaviors, we can manage the vulnerabilities that are inevitably introduced when the expected and actual system behaviors are out of alignment. BSM asserts that robust, secure information systems are best achieved through carefully modeling human/information interactions in social terms.

In a very real sense, the goal of BSM is to improve software quality as it relates to security by reducing ambiguity and ensuring security design accurately represents the stakeholders' needs. "More predictable systems" means systems that correctly implement stakeholder requirements. While the complete Behavioral Security Modeling approach includes all aspects of planning, designing, building, and deploying information systems, the first step, establishing the stakeholder requirements, is critical to the BSM approach, and is the focus of this white paper.

2 Functional Security Requirements Gathering

The Behavioral Security Modeling approach to functional security requirements gathering is a practical method based on real-world experiences working with business, development, and infrastructure teams on specifying security requirements, designed to address a current gap in both information security and application development practices. In meeting the gap it is simultaneously focused on meeting the needs of various participating stakeholders and leveraging the strengths they each bring to the table.

This gap is made apparent in outsiders' perception of information security. One of the authors personally experienced this perception in a telling conversation with a department head. He was greeted with "I don't care about this security stuff, it only causes problems when my managers request new users." Later in the conversation, the department head stated "Some of these system functions are so sensitive that not only do they need to be restricted to my team, but this specific group of people on my team." In retrospect, it is clear that the department head viewed security as "keeping the bad guys out," (thus "not her job") while at the same time having real needs for business controls that can only be addressed through security controls.

Functional security requirements gathering is driven primarily by organizational needs and the need for safety controls (protecting against errors and other non-malicious threats). BSM posits that to have robust and secure information systems that the functional requirements must define all human/information interactions permitted by the system; if it is discovered they do not, the team has discovered a defect and thus a quality issue[1].

The BSM approach specifically excludes any consideration of malicious actors (threats) from *functional* requirements; they are to be addressed through architectural (non-functional) requirements, adapted to current threats. Malicious actors either exploit weaknesses in the system architecture to bypass the normal application security controls, or abuse privileges they already have. Defending against malicious attacks is a separate problem that is better addressed through architectural requirements and design review. Good functional security requirements indirectly protect against insider abuse, by applying appropriate constraints to authorized users' actions.

Traditionally, security requirements have been exclusively non-functional / quality / architectural requirements, and most of the literature on security requirements reflects this, and can be summarized as "Keep the Bad Guys from messing with our stuff." Misuse cases, policy requirements, and the PCI Data Security Standard all fall into this category, which is characterized by requirements that are definable and repeatable across projects within an organization, or even between organizations, driven by the threats and exposures the organization faces.

In contrast, the goal of functional security requirements is to specify "This is what the Good Guys are allowed (authorized) and not allowed to do," which is a more difficult task since it is specific to the system being built. Where non-functional requirements provide a blacklist of bad things to be controlled, functional requirements provide a whitelist of authorized activity[2].

[1] Different teams may consider this a deficiency in the original requirements, while others may see it as having identified the need for an enhancement; regardless of the semantics, when the additional

[2] Although whitelists don't directly address malicious attacks, they make implementing security controls easier by defining the boundaries of authorized activity.

The majority of academic papers and articles from professional journals on security requirements discovered in the (admittedly cursory) literature search focused mainly or exclusively on malicious threats through misuse cases. The authors of this paper are not aware of any methods for systematically gathering functional security requirements beyond simple approaches such as CRUD (defining Create, Read, Update, and Delete for each function/field in the application for each user type), a gap that this paper attempts to address. Use of Role-Based Access Controls (RBAC) *is* considered a best practice to *implementing* authorization, the major component of functional security requirements, but it doesn't facilitate capture of implementation-agnostic authorization requirements, and has limitations that will be discussed later.

We have written this paper as a first attempt to address this methodology gap and to provide development teams and security professionals with tools to gather and refine good functional security requirements.

The approach described in this paper is meant to be flexible and adaptable to the software methodology used by any organization, whether it is Waterfall, Agile, or The Next Big Thing in Software Development. To that end, we present a framework for describing functional security requirements, checklists of questions to assist with eliciting requirements, as well as general advice, including some considerations that impact design. We also begin the process of creating a catalog of requirement patterns that address functional security concerns—a process we expect to take some time and community involvement to mature. This paper covers what the authors consider the most-neglected area of development: functional security requirements.

3 Approach

BSM views functional security requirements as including definitions of well thought out constraints, security actions, and post conditions. Within BSM we will define a security-related constraint as placing limits on interactions between Actors and Objects through defined Actions in information systems. Actors include people as well as external systems. When Actors are people they can be described by social groups within the organization (or broader community), by organizationally relevant roles filled by individuals, or by specific persons. System Actors can similarly be described by their social role—the role of the person or group they represent—or as a specific system. Objects represent logical sets of information or individual records or data elements that exist within the information system being described. Actions are externally driven operations within the information system, and include simple actions (e.g. read or write), complex actions that represent a business process (e.g. create meeting), or security actions (e.g. identify, authenticate, authorize, delegate, etc.).

Functional requirements have always contained some amount of expressed constraints. BSM seeks to make constraints a much more prominent piece of the requirements gathering process. To this end, identifying the desired constraints is accomplished through a structured series of questions to be considered. These questions are designed to establish appropriate limits of user authority within the system, while eliminating ambiguity and uncovering any implied "hidden" constraints.

For each externally driven operation within the system, the requirements need to define under what condition(s) the interaction is allowed and not allowed. A complete requirement[3] captures all of the constraints that apply to a particular operation. In general, stakeholders should consider "who needs

[3] See appendix on Styles of Requirements Capture

access?" first, and "who shouldn't have access?" second, in order to encourage thinking in terms of least-privilege.

In addition to explicitly defining the criteria when an interaction is allowed to proceed normally (the "Go Path"), functional requirements must also specify what happens if the criteria are not met (the "No-Go Path"[4]). Defining the No-Go path is important to both good user experience, and to ensure conditions that should be stopped or redirected are tested and verified later in the development cycle.

Returning to the first premise, BSM asserts that for a robust system that functional requirements must define all human/information interactions permitted by the system. Our focus is primarily on constraints, as we have observed that to be the largest area of deficiency in application development efforts we have participated in, but there are other areas to keep in mind. Some projects come up short in capturing and recording security actions, (e.g. business functions with specific security implications, such as identify, authenticate, authorize, create user, and delegate). We will touch on recording these types of requirements.

Since the goal is to ensure that requirements are comprehensive, in practice there is great synergy with modern quality assurance practices. From a functional point of view, security issues are quality issues and quality issues are security issues. General quality assurance techniques for software requirements gathering are well beyond the scope of this paper, but fortunately there is a wealth of information already available; we will call out a few of the resources we think look particularly promising.

Finally, we will touch briefly on the inclusion of "post conditions" within the set of requirements for an information system. A post condition is a statement of what must be true when a user's Action is complete, and can be stated both for successful completion and unsuccessful completion. For example, when an account transfer is completed successfully, the funds must be credited to the destination account and debited from the source account; if the transfer fails, the source account must still contain the funds, and the destination account must not have been increased. They are a means of explicitly calling out conditions that otherwise might be left as assumptions.

4 Constraints

Constraints are limits on the functionality of the system. They can limit who, where, when, or on what a function can be applied, and may constitute an outright prohibition, or be nuanced such as bounds on the size of a financial transaction. To help teams find gaps in requirements, we will discuss constraints in terms of five major categories: Social, Information, Location, Temporal, and Input. We will discuss each type in turn, and then provide additional advice on effectively gathering requirements around constraints.

Some constraints may fit the description for more than one of the categories. That is okay. What is important is that if the constraint is needed for the system, that it be reflected in the requirements for the system. Don't get hung up in philosophical discussions of whether a particular constraint is social, data, or input. We call out some distinctions of how we think about the constraints just for the purpose of helping create a mental model for the types of topics to consider for each category.

[4] The term "No-Go Path" appears to have originated in the Systems Safety discipline in the US Military; the earliest reference the authors could find is in NSWC TR 89-33 "Software Systems Safety Design Guidelines and Recommendations."

For each category of constraint we will first describe what the constraint is. We then discuss what the questions are that can be asked or considered to identify constraints of the given type. These questions are organized as "opening questions" followed by "clarifying questions" that should help ensure requirements are complete and unambiguous. To that end we also provide guidance around "uncovering hidden constraints," designed to reveal unstated security constraints, as well as general "advice" for the type of constraint being discussed.

Finally we provide known patterns for constraints of the given type. We use the term "pattern" in the manner used of "software design patterns," which are a well-established means in software development to simplify communication and avoid "re-inventing the wheel." A design pattern is a generalized solution for a problem frequently encountered, which is then given a name that can be used to succinctly communicate not only the problem but also the solution.

Design patterns are generally focused on problems faced in implementing a system, which is to say in system architecture, or in the structure of code. Our patterns are focused on the requirements for software rather than how those requirements are implemented, and thus the existing collections of software patterns are not of great help; however we firmly believe that the pattern mindset will both ease the specification of constraints and improve the quality of those specifications. To that end we have identified requirements patterns we have seen in our practice as a starting point. We will work to have a location available online[5] where these and additional patterns can be found and used by all. We encourage others to nominate patterns from your experience.

In some cases we also identify "anti-patterns." Like the name implies, it is the antithesis of a pattern, which in this case it means a known approach *not* to use. There may be situations where using an anti-pattern is appropriate, but it should be done with caution. We acknowledge that some anti-patterns can be controversial (even if a majority agree an approach is a bad idea, others may still embrace the approach), so as always use your judgment!

After discussing the types of constraints we will address an equally key concept, which is what happens when a user violates a constraint in a section titled "Go Path and No-Go Path: the Effect of Constraints." A "Go Path" is what happens when a constraint is not encountered. Thus the section really discusses the options and patterns for "No-Go Paths"—which could equally well have been called "Beyond 'Access Denied'" as we explore more modern ways of guiding users through constraints.

We will wrap up our discussion of constraints with general advice on the requirements gathering process around constraints, and even though our focus is on requirements gathering rather than implementation, we will end by touching on some key implementation considerations.

4.1 Social (Organizational) Constraints

Social constraints define permission to perform an Action based on the user's identity. That is to say, "who you are." Such constraints can be expressed for each Action as the social group permitted access. In our experience these are best expressed in the context of the organization, for example "Salesperson," or, more accurately, "all salespeople of Company X."

[5] A link to the security requirements pattern repository will be found on http://transvasive.com when it is available.

Whenever possible, base your social constraints on existing social groups. Organizations will frequently have language to describe the groups used to provision access. Using such existing terminology results in requirements that are easier for the business[6] to understand. Requirements that are more easily understood are more easily verified to be correct, and so in the context of security are more likely to result in a system that appropriately implements security rules. Using the natural language of the organization is an inherently human-centric practice and thus very important from a BSM point of view. A more formal way of saying this is that BSM asserts that access to functions is granted based on specialization within a group or due to a trust relationship between groups/individuals.

Generally, subjects are not expressed in terms of individual people, such as "I want to grant access to Alice," rather they are expressed in terms of the subjects' role (specialty) or social group (organizational unit), such as "I want to grant access to all cryptographers." However, in some cases individual access is appropriate, with a specialty or social group of one. This is particularly true with a specialty case where access is being granted not to a person, but to an external system; such access is frequently on a case-by-case basis.

Keep in mind that social constraints restrict access to Actions (functions or operations of the system). We will explore constraints that restrict access to Objects (information) in the next section (Information Constraints). You will see that information constraints are often expressed in terms of social groups as well. In our formulation, social constraints apply only to the relationship between Actors and Actions.

Opening Questions

For each Action, what groups of people need access to this Action to do their jobs / conduct business? Who should never have access?[7]

Clarifying Questions

If you know a constraint is tied to a specific person, Alice, but are thinking it is Alice herself: What is it about Alice's job (specialty) that makes it appropriate for her to have access? What is it about Alice's position within the organization (social group) that makes it appropriate for her to have access? (Note: *sometimes* it will turn out there are singletons of access, but this is by far the exception rather than the rule.)

Socially ambiguous roles, where B is a subset of A: Do you mean all A? or just B? Example: When you say "employees," do you mean all employees and contractors, or just all employees?

Ambiguous role definitions: Are all roles listed defined in such a way that would be clear to any reader (stakeholder, analyst, developer, tester, etc.)?

[6] We will make reference to "the business" to mean "representatives of the organization sponsoring the development of the information system" or with whatever term you use for representatives of the user base.

[7] It is good security practice to focus on who does have access (whitelisting) before who should be denied access (blacklisting). Controlling access primarily by denial tends to lead to mistakes that allow unintended access. More importantly, whitelisting mistakes are nearly always detected, while blacklisting mistakes are frequently not detected until a security incident occurs.

Distinguishing from information restrictions: Are people in this group (dis)allowed from using this function on some information in the system, or all of it? If it turns out to be information-based, see the guidance in the section on information constraints.

Uncovering Hidden Constraints

If someone struggles with identifying groups: What specific people need access to this function? What do they have in common? (Generalize from the specific examples)

Exceptions: Does everyone on this team need access to this function? What about (pick a specific example)?

Missing groups: So, everyone outside of this team will not need access to this function? What about (pick a specific example)?

Consider usage by external groups: Are partners / teams outside of the organization who need access to the function? Are there any that should never have access?

Advice

Understanding what groups exist within the organization is quite helpful to capturing good social constraints. Having a checklist of already-established groups (a Social Group Catalog) is even better. Org charts are a form of this from business side, and actor inventories are a technical-side example. Security practitioners should develop and refine a standard set of social groups for their organizations; this is commonly done when the organization has implemented a Role-Based Access Control (RBAC) model, but need not be.

It is not usually a part of analyzing the constraints for any particular Action, but at some point you should confront the question of what a user with no account sees/can do. In many systems the answer is that such a user would see the login screen and nothing else, but web applications frequently have other answers and the “no account” case can be inadvertently left out until very, very late in development.

In the end, our strongest advice for the practitioner is to try to think less about *who* the Actor is and more about *what* characteristic of the Actor leads to authorization to perform an Action. As you look at the patterns, below, you will see that this holds both with RBAR and with ABAR.

Patterns

Role-Based Access Requirements (RBAR) Pattern. Role-Based Access Requirements is a means to define how to authorize Actors to Actions. People (Actors) are assigned to one or more roles, based on their job functions or other social criteria, and permissions to Actions are granted to roles. This is a method of defining requirements that would directly feed into a Role-Based Access Control (RBAC) implementation, which is the security infrastructure used by most large organizations, and is widely accepted as the standard approach to managing authorization. RBAR works very well for Social constraints, but is not effective for other types of constraints. An advantage of defining Social constraints via RBAR is that there are readily available tools to assist in implementing such a scheme.

Attribute-Based Access Requirements (ABAR) Pattern. Our discussion of social constraints as being based on social groups almost pre-supposes the RBAR Pattern, but the ABAR Pattern is an alternative. Just as RBAR is a method of defining requirements that feed easily into RBAC implementations, ABAR is a method of defining requirements that would feed easily into an Attribute-Based Access Control (ABAC)

implementation. ABAR defines authorization based on attributes of the user, for example restricting an Action unless the Actor is over 21 years old. How you verify the accuracy of the attributes is an issue with implementation, but if the security controls are truly in terms of attributes, then the requirements should dictate the attributes rather than the "roles" (social groups, etc.).

Action Groups Pattern. Action groups are higher-level Actions ("meta-Actions") that the finer grained Actions the system offers roll up into. "CRUD" (create, read, update, delete) is sometimes used in this manner, where there may be many, many Actions that perform a "read" operation, but permissions are set for all such Actions based on the parent action of "read". To use this pattern, define your set of meta-Actions and map each Action to a meta-Action. Write your constraints in terms of the meta-Actions. If you find you must map one Action to multiple meta-Actions, a least-privilege principle should be applied such that a user must be authorized to *all* associated meta-Actions in order to perform the Action. Note that this pattern is fully compatible for use with either the RBAR Pattern or the ABAR Pattern.

"Everyone" (Anti-Pattern). "Everyone" is a logical construct in many operating systems representing any security context, the logical opposite of "Nobody." While useful in system administration, it is dangerous to use in requirements definition, since people will frequently make unstated assumptions by associating "Everyone" with a social group that fits with the context. This anti-pattern can be repeated with other broadly defined social groups, such as "All Employees" which could mean either people on the organization's payroll, or all people working for the company, which would include off-payroll contractors. Avoid this anti-pattern by asking clarifying questions to specifically define what "Everyone" means, but be ready to embrace it in those cases where you establish that it is the correct answer (such as with a public website).

Deny Access To… (Anti-Pattern). "Deny access to (social group)" explicitly forbids a specific group from executing a function. In this anti-pattern, access is often but not always broadly granted, with a specific team excluded. Deny logic is often how people think, but invariably leads to broken security later on, and should be avoided. Deny is the security professions' GOTO statement. Refactor by exploring why an exception exists and more narrowly defining the group(s) granted access.

4.2 Information Constraints

Information constraints define permission to access Objects (information) based on "what the information is." They can refer to any single Object or any definable set of Objects managed by the system. Information constraints can apply to either a single object, such as "the customer's (the one accessing the function) credit card number," or all objects, such as "all blog posts," and can also be limited by other properties of the information itself, such as "clients served by the St Paul office."

Information constraints at their simplest may apply in the context of a particular system Action, just as we saw with social constraints; an example could be "delete account can only be applied to accounts that have never been billed". In this case we do not contemplate anything about the Actor, only the Action and the Object.

Information constraints more frequently get combined with a social role resulting in rules such as "a patient may access their own records" or "sales people may only access clients served for their sales district". In these examples we see both an Object and a characterization of an *associated* Actor. While one could say that "access" is a system Action, the intent is in defining a universal requirement for the system that crosses all Actions. The key to note for information constraints that include Actor information is that you are now stating that there has to be a correlation in the managed data of the system between

the system users and the data being managed. In the first example, "patients" are an entity managed by the system but are also users of the system, and thus there is a need to correlate user identities to the managed data. In the second example, it is implied that Actors who are "sales people" must have an attribute called "sales district", and that in the managed data that a "client" also has an attribute called "sales district" (or perhaps "served by sales district"). Because of these complications, make certain the system specifications include user functions to manage the associations between managed data and system users.[8]

The most complicated information constraints will factor in Actor, Action, *and* Object. This would be the case when a broad set of users can view a type of data, but only a subset can edit that same type of data. This level of detailed control can make a system tedious to specify and can greatly increase the magnitude of complexity for a system (with related impacts on cost and quality). When you need it, you need it, but think if you have alternatives before easily jumping to this style of "compound constraint". If you do need it, we have some suggestions in the "Advice" section on taming requirements of this sort.

Information constraints differ from social constraints in that they restrict access to information (Objects), where social constraints restrict access to functions (Actions). Even when information constraints are expressed in terms of social groups, the constraint is tied to a property of the information itself, either the data or metadata. Where social constraints express a relationship between identity (Actor) and functions (Actions), information constraints can relate identity (Actor) to information (Objects), functions (Actions) to information (Objects), or functions (Actions) in the context of a particular Actor, to information (Objects).

Opening Questions

Can this Action be applied to all data? If not, what data can it be applied to? Do all users have access to the same data with this Action? If not, what information does each social group need to execute this function against? Is there data that should be excluded?

Does the person/system calling this function only need access to their own data?

Clarifying Questions

Ambiguous grouping of data: What attributes differentiate the allowed data from the disallowed data? Are the differentiations implicit? (Make them explicit) Are they defined in such a way that would be clear to any reader (stakeholder, analyst, developer, tester, etc.)? (If not, rewrite to make them unambiguous.)

The remaining clarifying questions only apply if the constraint has a social component.

Ambiguous associations between Actor and Object: Is it clear for each constraint that links Actors and Objects, how to interpret that association? For example, if the constraint limits access to "the user's own bank accounts", is it clearly understood how that is determined? (Must you be the first owner listed? Also listed on the account? Listed as a signer?) Rewrite ambiguous requirements so that assumptions don't need to be made (for example, by explicitly stating assumptions).

If you know a constraint is tied to a specific person, Alice, but are thinking it is Alice herself: What is it about Alice's job (specialty) that makes it appropriate for her to have access? What is it about Alice's position within the organization (social group) that makes it appropriate for her to have access?

[8] It is also useful for those who model information for the system recognize that this form of requirement immediately implies that there must be a correlation to users within the managed data.

Socially ambiguous roles, where B is a subset of A: Do you mean all A? or just B? Example: Do you mean all employees and contractors, or just all employees?

Identifying social-only restrictions: Are people on this team allowed (or disallowed) from accessing this data with just one function, or all functions? If it is for all functions, write the constraint as a general constraint for the system, not just for this Action (what would happen if you forgot it for one of the Actions?)

Uncovering Hidden Constraints

If someone struggles with identifying data groupings: Is there sensitive data in the system that requires additional restrictions?

Identifying "private" data: What impact would there be if this data was shown on the welcome page, immediately after login?

Exceptions: Does everyone on this team need access to this information? What about (specific example)?

Missing groups: So, everyone outside of this team will not need access to this information? What about (specific example)?

Consider usage by external groups: Are partners / teams outside of the organization who need access to the data? Which specific data? Are there any that should never have access?

Advice

Understanding what data exists within the organization, ownership of data, and how data is shared between groups informs capture of data constraints. In principle, formal data ownership is desirable, but rarely implemented in practice (but data governance programs are on the rise—leverage it if your organization has one). Security practitioners should develop a catalog of common data groupings (a Data Group Catalog), such as "sales office" or "sales region." If your organization has prepared a formal data classification scheme and already mapped it onto data, it may also be an asset for requirements definition.

Social constraints are much more common than information constraints, which can lead to problems when implementing information constraints. When possible, Action access should be managed separately from Object access. In that model each Action would have a list of authorized users, (regardless of Object) and each Object would have a list of authorized users (regardless of Action). The more complex relationships (granting access to objects for a subset of functions the actor is authorized to use) are inherently costly to implement and manage[9].

In the cases where you can't avoid combining Action, Object, and Actor into a single constraint, our best advice is to seek to generalize. Rather than making constraint determinations at the lowest level—i.e. each user group for each type of data for each system function—instead define as many of the constraints as broadly as possible. At a minimum, use the Action Group pattern from the section on social constraints. If you can define a relationship between Actor and Object independent of Action, do so. If you can define a relationship between Action and Object (or better yet, Action Group and Object), do so.

[9] As a note for implementation: trying to manage object access with infrastructure exclusively designed to support function (Action) access is very difficult, if even possible. Most solutions the authors have seen have been homegrown. ABAC and XACML look promising for taming this problem.

Define the fully detailed Actor for Action on Object only on that subset of cases where the rules do not lend to generalization.

Patterns

Dual Controls Pattern. The Dual Controls pattern enforces a common financial control: requiring two different people to approve a transaction. Having two "signatures" makes collusion necessary in order to defraud the organization, which is much less likely than a malicious insider acting alone. Frequently this pattern may also prohibit one of the Actors from having organizational authority over the other. It thus manifests as a limit for approval Actions on which transactions a given Actor can act upon (i.e. not transactions that they originated, nor transactions that subordinates originated).

Role Based Data Access Requirements (RBDAR) Pattern. Role Based Data Access Requirements[10] define how to authorize permissions to data contained and managed by the information system. Like RBAR, people are assigned to one or more roles, based on their social group memberships, and permissions are granted to roles. Unlike RBAR, RBDAR permit or deny use of any function (Action) on the designated data (Object).

Access Control List (ACL) Style Requirements Pattern. This is again a style of writing requirements based on a common security management implementation, in this case Access Control Lists. An Access Control List allows arbitrary parings of identity and objects to functions. In the most complex form, an ACL is a list of Access Control Entries (ACEs), attached to a single Object where each entry consists of a role (Actor) and function (Action) pair that is allowed to manipulate the Object. Access Control Lists traditionally provide a fixed list of primitive functions, such as read and write, so either also use the Action Group Pattern, or extend past the traditional ACL and include all Actions defined within the system. While the traditional ACL would be defined for each Object individually, for the ACL-Style Requirements Pattern, we recommend also using the RBDAR Pattern whenever possible (to set access for groups of Objects, rather than individual Objects).

An ACE in the complex form might look like:

```
Allow Accountants (role) to Add a General Ledger Entry (function) to the
General Ledger of Firm X (object).
```

Use this pattern only in cases where all such permissions are being chosen during design and where they will remain fairly static once in production.

By contrast, in situations where the system manages access permissions as part of the user functionality (e.g. Microsoft SharePoint®) then you need not *provide* ACL-Style Requirements, but rather you need to ensure that such functionality is defined for the system. This would include user Actions to maintain the ACLs for the data as the data changes within the system.

Attribute-Based Data Access Requirements (ABDAR) Pattern. Perhaps unsurprisingly, this takes the notion of identifying users based on attributes of the user, rather than roles (social grouping) they associate to, and applies it to permissions to data contained and managed by the information system.

[10] Unlike RBAR and ABAR, there is no pre-existing implementation of "RBDAC" that the authors are aware of.

"My Data" Pattern. The "My Data" Pattern is used when managed information within the system logically "belongs" (is owned by) to or represents the current user of the system. Note that logical ownership does not necessarily equate to legal ownership. The key to using this pattern in requirements is to provide an unambiguous definition of what data "belongs" to an arbitrary user. Unambiguous, as usual, means to define in such a way that would be clear to any reader (stakeholder, analyst, developer, tester, etc.).

Delegation Functionality. If you find yourself with constraints stating that an Actor may view Objects based on a different Actor having delegated permissions to see and do what that different Actor can see and do, then what you need isn't constraint language, but a full delegation and delegation management feature. See the discussion of "Delegating Authorization" in Section 5.3, Security Actions Performed by the End User, for more information.

4.3 Location Constraints

Location constraints are defined by "where you are," the physical (or logical) location of the Actor executing an Action. Location can be constrained to a specific location (Minneapolis office), a group of locations (all offices, remote access locations), or unconstrained.

Although information constraints frequently incorporate location, information constraints are based on attributes of an Object, and restrict access to Objects, while location constraints are based on attributes of the Actor, and restrict access to Actions. It is possible for a location constraint to limit access to data rather than Actions (e.g. not being able to access Top Secret designated documents when not on a properly secured network), but the authors have not seen this come up in our experience.

Historically, location constraints have been rarely used, except by default ("in the office" vs. "offsite"), although with the growth of IP geolocation, location constraints are becoming more common.

Opening Questions

Are there policies governing where the system (or particular functions, or particular data) may be accessed from? If so, do you want those policies enforced by the system?

Where is the team that will be executing this function physically located?

Does this team need the ability to execute this function from anywhere in the world, or from specific locations?

Will there be localized versions of the application presented to users in different countries? Note: ensuring users only use appropriately localized versions can be a security concern if the localization includes mandated limits.

Clarifying Questions

Ambiguous location or location groupings: What differentiates the allowed location(s) from the disallowed location(s)?

Uncovering Hidden Constraints

Location Anchoring: Does the team always operate in a specific physical location? *If so, use of the function can be constrained to that location.*

On-Premise Only / Remote Access: Does the team need the ability to work from home, or from other remote locations away from the organization's physical offices?

Consider usage by external groups: Are partners / teams outside of the organization who need access to the data? Where do they physically reside?

Consider "Banned" locations: Are there Countries, States, or other locations that should be blocked from using the application, or specific features? *For example, some countries have laws restricting the use of encryption.*

Consider simultaneous login: If an account is accessing the system twice from different locations, should the system react in some way?

Advice

Don't assume that you need to find location constraints. Many systems do not have any. See Section 4.7, Techniques to Improve Constraint Requirements, for discussion on the question of whether you should apply restrictions just because you can.

If you will be considering location constraints, understanding the physical locations the organization occupies and how they relate to the organizational social structure is helpful. A catalog of locations and logical groups may be helpful.

Using location in authorization *decisions* is relatively new, but can be a powerful tool to limit unauthorized access. (In the past, location based security was largely limited to network design rather than logical authorization decisions based on location markers.) Credit card issuers already use location in authorization decisions, denying card-present charges in two distant locations in too narrow of a time span. Care should be taken in trusting location data, however. One example of this was related to the authors: a credit card transaction with a Texas-based vendor at a traveling fair in Minnesota made it appear to the issuing bank that the cardholder had made transactions in two states in a very short period of time, and triggered a fraud alert.

The example highlights one of the main challenges in implementing location-based constraints: how do you determine physical location? Even IP geolocation isn't fully reliable. As always, the authors recommend the requirements be written strictly as the business need, with implementation methods only dictated if the business actually can't abide a different means of implementing. With location constraints however, after the requirements are articulated there will almost always be design tradeoffs that need to be discussed with the project stakeholders.

Patterns

On-Premise Only Pattern. On-Premise Only limits all use of an application to people physically located on company premises, office buildings and the like. (Before the introduction of remote access technologies, all computer use followed this model.) Remote access and VPN must be taken into account when you use this pattern; for example particular applications can be deliberately excluded from VPN access if needed (that is the implementation detail, the point is that if you have a requirement to list it and then let the implementers come back with options on how to make it work).

Console Only Pattern. This pattern is seen in the Root Login on Console Only restriction to limit administrative logins, including the built-in Windows Administrator and UNIX root accounts, to the keyboard and display physically attached to the server hardware. So if your project is a server operating system, you may already use this pattern. This pattern can also be used for highly sensitive business operations, like high-value wire transfers, by limiting them to dedicated terminals. Note that connecting the

physical server to a KVM switch with a remote control option may ease administration, but breaks the security model. The Console Only Pattern is a specialized version of the On-Premise Only Pattern.

Non-Concurrent Login Pattern. Non-Concurrent Login provides the system behavior for when one account is used to login more than once at the same time. There are sometimes specialized No-Go Paths for this pattern. Notably there are patterns where: (1) upon a second login, the previous session is immediately logged off, or (2) a second login is denied until the first session is logged off, or (3) a variant that combines the first two where the second login is given the option of terminating the prior session, and only if they do so can they log in, or (4) schemes where one or both of the application sessions are notified of the concurrency. Many (but not all) instant messaging services employ (1). (2) was used on some old terminal-based computer systems, but this approach is considered by the authors to be an "anti-pattern" today. (4) is perhaps the most nuanced, and is actually required behavior for systems subject to US FDA oversight in complying with 21 CFR Part 11.

4.4 Temporal Constraints

Temporal constraints are defined by "when," the actual or logical time when the Actor executes the Action. Temporal limits can be a time of day (during business hours), or some other defined window of time. Temporal constraints also include limits like time to perform an Action – for example, limiting the time to complete a ticket purchase.

Temporal constraints are very much like location constraints, in that they both restrict access to Actions based on attributes or behaviors of the Actor (in this case, when the Actor is acting, how long they take to act, etc.). Unlike location constraints, temporal constraints generally stand alone and are not easily confused with other constraint types.

One counter example of this is in financial securities where limits are frequently imposed that start at a fixed point before the end of each quarter and continue until financial statements for the quarter are released; this defines a window of time, but because the window could be based on Actor (which company are they part of), or the Object (which security is being traded) even temporal constraints can have ambiguity with data constraints. The good news is that "type" of constraint is just a tool for remembering to consider all aspects when developing requirements. The resulting requirement is the same regardless of whether you conceive of it as a temporal constraint, a data constraint, or any other type.

Opening Questions

When (what time of day) will teams be executing this function? Will they be working outside of normal business hours? Even if they don't usually work outside of normal business hours, could they ever?

Are there specific months/weeks/days in the calendar year when teams will be using this function?

Will there be different versions of the application presented depending on the time of day / year? Note: it can be viewed as a security concern that users not access a given version outside of the prescribed window. In any case, it is an overall quality issue to have the system behave appropriately.

Clarifying Questions

Ambiguous time period: What differentiates the allowed time(s) from the disallowed time(s)?

Risk of Interfering with Needed Actions: Does the benefit of constraining the users' behavior outweigh the risk of disrupting needed activity?

Uncovering Hidden Constraints

Planned Obsolescence: Is there a planned retirement date for the function/application? Will the function be needed only for the duration of a project? Do these need to be enforced by the system?

Exceptions: Are there times when this function should be disabled? Are there days when this function should be disabled?

Timeout: Are there time limits for completing a transaction? Are there limits on the length of a [login] session? Are there limits on the maximum period of inactivity?

Advice

For temporal constraints, understanding normal operational hours, what times of day are meaningful to the organization, as well as the organizational calendar will help define temporal constraints. Whenever possible, leverage the existing financial calendar and daily schedule of operations many organizations have.

Not all organizations employ temporal constraints, and those that do will typically use either the During Business Hours Pattern or the Not During the Batch Cycle Pattern described below. When implemented, temporal constraints generally affect all functions within a system, either for all users or specific users. However, it is conceivable to restrict access by group, such as for different shifts, or on other more tailored bases such as is done with physical access to facilities (via key cards).

If you choose to implement temporal constraints it is critical that you carefully consider how they will affect users' ability to achieve their goals, which for an in-house system is to serve your customers. Temporal constraints are most commonly applied to deter bad actors, which is good. At other times they serve to protect system resources (time-outs), again worthwhile, or even to create hard limits to enforcement management policies. *All* temporal constraints can impede necessary work in ways that are sometimes unexpected. We will discuss reviewing the difference between wanting to and being able to apply a constraint versus it being something you should implement in the Common Themes section.

Patterns

During Business Hours Pattern. During Business Hours defines a time period during the day for "normal" operational hours during which all access to the system is allowed, for example, 6 AM to 6 PM, Monday-Friday, excluding holidays. Access outside of normal operating hours is either denied or permitted only to privileged users. Physical building access is often controlled using this pattern, and most operating systems can restrict logon hours to specific times of day. If you use this pattern, make sure there is a means in the requirements to specify and/or maintain the business hours for the system, and to facilitate overrides when needed.

Not During the Batch Cycle Pattern. Similar to the During Business Hours Pattern, Not During the Batch Cycle defines a time period when updates to the system are not allowed, usually to maintain data integrity during a batch processing cycle. Access to functions that can change information in the system is blocked, putting the system in "read-only mode," or access is blocked entirely. Large batch processing cycles have been a hallmark of legacy systems, dating back to the first mainframes. Faster computers, more sophisticated software engineering techniques, and an expectation for 24×7 access have led to a

decrease in the prevalence of batch jobs having exclusive use of the system, but if you have any batch processes you should consider if there are any processing actions that are not safe to perform during the batch and design accordingly.

For the Duration of the Project. For the Duration of the Project relates access to information or functions to a project, program, or process with a defined start date and completion date. A full implementation of this pattern grants access to a set of information and functions on a specified start date, and removes access after the completion date. In practice, organizations rarely implement this pattern fully; at best, project-related documents are retained for a specified period of time and then deleted, which effectively removes access.

4.5 Input Constraints

Input constraints are limits on the direct and indirect input values to an Action, including monetary values, quantity, and other factors; for example, limits on the maximum value of a check for which an employee can authorize payment, or ensuring the person who requests a check can't also approve the same check. The input values can either be provided by the user or can be constraints on the value options made available to the user by the system.

Input constraints blur the lines between security and quality, since some constraints will exist solely to enforce reasonable inputs; for example, ensuring a future date is actually in the future. Unsurprisingly, input validation falls under the category of input constraints, a concern that has both quality and security implications. As we discussed in the Approach, quality and security are often flip sides of the same coin.

Like information constraints, input constraints limit use based on values or properties of information, however, information constraints relate identity (Actor) to information (Objects), while input constraints apply limits on executing functions (Actions) on information (Objects) based on values of Objects provided as inputs to the Action, or combinations of Actors and Objects.

Opening Questions

What are the maximum (or minimum) input values for which the team is authorized to execute this function?

Is a type designated in the transaction where the types available to different users vary?

Are there limits to what is reasonable for a field? (E.g. a person who is 12 feet tall, or a 120 years old)

Are there combinations of inputs that are nonsensical?

Clarifying Questions

Ambiguous Object/Actor combination: Does the limit apply only to this team, or anyone executing this function?

Uncovering Hidden Constraints

Pick extreme values for inputs: Would the team still be authorized to execute the transaction for values greater than [silly extreme value; 100 Billion]?

Pick invalid inputs: What if [someone entered a letter in a numeric field]?

Scenario-based constraints: Is the team/individual always allowed to execute the function? Are there certain situations when the team/individual should be prevented from executing the function? (Like for restrictions based on the input values, transaction options, etc.)

Advice

Input validation is the most common type of input constraint. If there is a way to define these globally, it is often preferable. This can be general statements: "dates must be MM/DD/YYYY" (beware of localization needs!), "unless otherwise specified, integers must be non-negative", "unless otherwise specified, non-whole numbers will accept up to two digits after the decimal", "text fields may not be longer than the database field they map to can store" (assuming here that the stakeholders will see and review the lengths in a data model), etc..

It is frequently not deemed worthwhile to spend time documenting the input validations required for every input field. Some projects will leave it to the developers to do what is "reasonable." In such a case the authors would recommend that the developers record what is implemented, and that the stake holders review that to confirm it is acceptable, and that those de facto requirements be tested.

Even if you don't want to define the constraints on every field, there may be particular fields you want to call out. Date of Birth being in the future is nonsensical. Many systems will prevent back-dating and/or future dating transactions, or limit the distance. Policy end dates may be required to be on the last day of a month. End dates/times generally cannot be earlier than start dates/times (unless the system gives this special meaning).

Never forget the format of structured data like phone numbers: if you don't specify whether you want "+1.555.123.4567" versus "(555) 123-4567" versus "555-123-4567" you may get a system that accepts all three, or only one (and there are many other ways of formatting phone numbers). It goes beyond constraints, but you may also want to specify that the user types only the numbers and the formatting characters are to be inserted by the application. (And again, keep localization needs in mind.)

Regardless of whether you specify the detailed input validation, all projects should consider what the application behavior should be when an input validation catches something that is not accepted. This is a specialized form of "No-Go Path", and we will discuss it further in that section.

Beyond basic input validation, awareness of key threshold values, either in terms of volumes or dollar amounts, can be helpful to defining input constraints. Building a threshold catalog may or may not be practical, as these thresholds may vary depending on the business function in question.

If your system handles money, you should always ask whoever oversees the financial dealings whether there are financial/accounting controls that apply. Whether you think of these as constraints, business rules, workflow, or general requirements, make sure they are accurately reflected in the requirements. For systems that produce checks, originate funds transfers, etc. all but the smallest organizations typically implement both the Transaction Limits and Dual Controls patterns to protect payments.

Patterns

Transaction Limits Pattern. Transaction Limits define a maximum (or minimum) value for a key input to a transactional function, for example, a dollar-value limit on a purchase or account transfer. Limits can be absolute (Functional Transaction Limits) or vary depending on role (Role-Based Transaction Limits).

Many organizations implement purchasing limits using a Role-Based Transaction Limit pattern, where higher-ranking managers have larger limits on their purchasing approval authority.

Input Validation Requirements Pattern. Input Validation, the intersection of quality and security, as a pattern establishes the list of valid values that can be supplied as inputs to an Action. Most input validations are trivial and should be defined globally, but some can be tied to a specific requirement. Although input validation frequently has been left as an exercise for developers to implement in code, business rules often determine what values are valid. By including Input Validation in requirements, valid values can be constrained more precisely, and are more likely to be tested (with proper traceability). There are a large number of common validation checks, which we will attempt to catalog as we get the catalog overall patterns online for community use. We will also work to catalog "No-Go Path" options where it comes to input validation.

4.6 Go Path and No-Go Path: The Effect of Constraints

Requirements generally define the "normal," expected action, the "Go Path" (also sometimes called the "happy path"). Your requirements may also define alternate paths that may occur. With a focus on constraints, we need to make sure we document what happens whenever a constraint is encountered, which we call the "No-Go Path."

For example, if the Actor meets all of the restrictions, the function is executed as defined; for example a Salesperson in the Minneapolis office is allowed to create a new Minneapolis client. But what happens if someone tries to execute a function and is not authorized; for example a New York Salesperson tries to create a Minneapolis client? Throwing up an "ACCESS DENIED" error message[11] isn't always the right answer, and is hostile to a user who is only trying to get work done.

Consider SharePoint – when a user tries to open a page they're not authorized to view; instead of seeing "Access Denied. Sorry, you're done," SharePoint presents an access request form, complete with a box to explain why access is needed, that is automatically sent to the page's owner. Other possible responses could include automatic redirection (escalation) to an authorized user (such as a supervisor), or allowing the action anyway, but warning the user beforehand, and alerting a supervisor afterwards.

For each constraint we must always define what to do when the constraint is violated—the "No-Go Path." Considering Go/No-Go Paths is the final component of the requirement gathering process when it comes to constraints. In practice the No-Go Paths are usually defined as constraints are defined.

As the reader may observe, once you begin to define No-Go Paths, there is a natural synergy with implementation of exception handing. Including the No-Go Paths in requirements helps ensure conditions that should be stopped or redirected are designed, coded, and tested later in the development cycle. Similarly, when developers encounter unexpected conditions ("system errors"), there will be a natural framework to discuss how such conditions are to be communicated to the user, and how the application should behave as a result.

[11] Security professionals have been conditioned towards denying access being the primary tool for achieving security. This creates tension with groups who have business needs to meet. We're advocating finding a "middle path" so to speak.

Questions to Consider

If a constraint is violated, what should the system do? Does it help the user ultimately accomplish their goal if possible? If the Actor receives an "access denied" message, what should be their next step? *The next step can be used to fashion a more helpful error message, or be directly implemented in the system.*

Can user experience be improved by preventing invalid input actions rather than reacting to invalid actions?

Consider ambiguities where no path applies: Do any business rules have situations that are unaccounted for? (e.g., we cover <500 and >500 but not =500)

Have the requirements provided guidance for what to do in the case of a "system error"? Is there a fail-safe path the system can follow "when all else fails?"

Advice

No-Go Paths may represent new Actions within the system, and (recursively) also need fully defined constraints. Infinite recursion can be avoided by specifying (at some point) ALL violations follow a defined alternate flow. "If all else fails… do this." No-Go Paths need not be defined separately for each constraint, they frequently can be grouped or defined globally.

Remember the primary purpose of information systems is to allow users to accomplish their goals. Security (including integrity) needs to be part of getting them to their goal, not a road block. The system will be more useful the more it guides the user towards their goal. Instead of just an "invalid input" message, mark the fields that are invalid, and provide the validity criteria. Instead of "access denied", provide an alternate means for the user to accomplish their goal (e.g. routing a request to an authorized approver, or provide instructions on alternatives available to the user). In some cases the action simply violates policy, and if so, identify the policy. The goals in designing a system are not to be cryptic and to facilitate success whenever possible.

A primary way to enhance the security of a system is to design it in such a way that users are not incented to circumvent the security controls. "User-friendly" isn't coddling, it is enhancing to all aspects of the system, and that includes security.

Patterns

These first patterns are fundamentally preventive in nature. They constrain user behavior by not giving the user the option to perform an invalid action. Note that if you use these techniques that during implementation care must be taken to ensure that security is still applied on the server side of processing so that malicious actors cannot gain access by circumventing limitations imposed in the user interface.

Prevent by Not Offering Pattern. If the user cannot perform a given function, remove the means by which they would have initiated the action. This could mean changing the menus of the application to remove menu items that the user is not authorized to select, or that are situationally not allowed based on the current context in the application. The modification has two available forms: actual removal (not displayed) or "greyed out" (displayed, but shown in grey and not selectable). The same can be done for

buttons that trigger options—either only display the button when the action is allowed[12], or "grey it out" when not allowed.

Delay Offering Submit Option Pattern. This is a variant of the Prevent by Not Offering Pattern, and applies most commonly to input validations. The button/link to proceed from the current page is either not displayed, or is "greyed out" until all requirements on the page are met. It is common for deficiencies to not be displayed until an attempt is made to submit, so if you use this pattern, make sure the application still guides the user through all required details.

Filter Available Options Pattern. Filter choice lists to only those options available to the user. For example, if the user cannot view a particular patient, don't show that patient in search results (unless to omit them would be confusing, and then this pattern is a poor choice), filter drop-down lists to only applicable values in the current context.

The remaining patterns are fundamentally reactive in nature. They define system that needs to occur when a user attempts to perform a restricted Action.

Route for Approval Pattern. If the user attempts to perform an action for which they are not authorized, notify them and give them the option of still submitting the transaction and having it routed to an appropriate person for approval. Variants of the pattern could include automatically routing for approval without prompting for confirmation.

Allow Override Pattern. The user is notified of the issue and given a chance to proceed anyway (to override the constraint). Typically used for minor safety controls, for example validating that you really meant to say that your new hire is over 100 years old. It is common for such override actions to be logged (who performed the override and what the details are), so if that is desired make sure it is part of the articulated requirement. A variant would include notifying the user that their decision to override will be logged. It is common for only particular roles to be granted permission to override, in which case this pattern would pair with other patterns, based on the particular Actor.

Warn and Notify Pattern. A variant of the Allow Override Pattern. The user is notified and given the change to override. Notices of overrides are sent to a responsible party for follow-up. Variants would include notifying or not notifying the user that a decision to override will result in such and such party being notified.

Access Denied Pattern. When the Action is simply not allowed, the user is notified. The notification should be in terms the user will understand, and if possible the reason it is not allowed should be included. As previously discussed, too much use of this pattern can constitute an anti-pattern.

Log Only Pattern. In this situation, action is not prevented, it is merely noted (or alternately, notification may be actively sent). A variant of this pattern would be to log all attempts to violate a particular constraint, regardless of whether the Action ultimately is successful or not. This pattern really is just defining a form of audit logging, which is normally considered a non-functional requirement. We are including it here

[12] Features appearing and disappearing for the same user can lead to frustration. Only remove entirely if it is beyond obvious when it appears and disappears. For greyed out items, consider offering a tool-tip when hovering to say why it is unavailable.

because during the process of determining constraints, outlier events may be discovered that should be sent to the audit log.

One Error at a Time Pattern (Anti-Pattern). Each time submit is pressed, only the first error is reported to the user, even though there are multiple issues. For example, on a form where the user has made three errors, they attempt to submit and are told only of the first error; they fix that error and attempt to submit again, and are now told of the second error, and so forth. Failure to supply all errors at once leads to significant user frustration and wasted time. One instance where this pattern should absolutely be used is providing feedback on user credentials (i.e. username and password). For credentials, the only feedback that should be provided is that the credentials are invalid. Reporting "invalid username" allows malicious attackers to discover valid usernames. This pattern becomes advisable anytime detailed feedback would realistically assist in an attempt to compromise the system.

4.7 Techniques to Improve Constraint Requirements

At this point we have conveyed our conception of constraints as part of functional requirements, particularly as regards functional security requirements. Since many teams may be unaccustomed to having so many constraints as requirements, we want to provide some general guidance; this is both to make the requirements as useful as possible as well as to attempt to avoid having a newfound focus on constraints gum up the whole application development process.

The techniques described here should help ensure an added focus on constraints doesn't become a detriment. If focusing on constraints hinders the process, attention will be withdrawn. Then we'll continue to have too many systems with too many rough edges that compromise security and safety.

As you read these suggestions, keep in mind that they all can be used for all requirements, not just constraint-based requirements.

Prioritize

"Just because you *can* doesn't mean you *should*."

This is probably the single most important piece of advice in this paper. Designing a system to control absolutely everything it possibly can is likely to result in a user experience that feels like working for the worst micro-manager in history.

Putting in too many constraints has a number of very real negative effects, beyond just user annoyance. It can increase the complexity of the code, making it more error prone and more "brittle," which is to say, difficult to maintain and especially enhance. It may sap resources so that not-so-important constraints get implemented while so-very-important features are still waiting to be done. And well-meaning constraints can get in the way of valid work that occurs in ways that the group specifying the requirements didn't contemplate.

This is not meant to deter you from using constraints, only to encourage you to look critically at which available constraints add to safety, security, and usability, and which are not as valuable.

Depending on your work style, you can still bring up all of the potential constraints (some people prefer free-ranging discussions, and others want to focus on only the most important and squash "tangents" early—do what works for your team). Discuss them in requirements/design sessions, throw them on the

white board, or put them on a list of requirements. Just make sure you review what to keep and what to cull before the requirements start being implemented.

A useful technique is to create a list of all constraints (and a summary of their No-Go Path), including what they apply to. "Force rank" the list, which is to say, put the constraints in order from what you think is the most important and valuable constraint down the least valuable. Hopefully the items at the top of the list qualify as "must have" requirements. But as you proceed down the list you should go through the succession of "should have", "nice to have", and "probably should skip".

There is a school of thought that the majority of "nice to have" features are never used and thus are a waste of resources to implement. There is an acronym that reflects this: "YAGNI" (pronounced YAG-nee), "Ya Ain't Gonna Need It." Constraints are preventive measures for security, safety, or both. So the *need* is based on the likelihood of malicious or inadvertent misuse, with the reduction in risk exposure (tangible and intangible) weighed against the cost of prevention. This type of risk assessment is exactly the information security professional's job, and so the determination of which constraints to include or exclude is where functional security requirements will overlap with traditional security practices and where expert advice should be sought.[13]

However you arrive at your assessment, drop any requirements that don't add value. Note which are firm and fast requirements, and for the rest, work with the implementation team to understand what is easy and what is hard to implement. This may be enough to decide which requirements to keep and which to drop. Another approach would be to request that the development team initially implement all requirements that you place above a cut-off line (usually somewhere between the end of "must have" and the end of "should have"), along with any others that don't materially affect how long it takes to implement the overall functionality the constraint applies to. Then if there is still time remaining later in development, you can return and implement more.

If you are using an Agile methodology with user stories, a backlog, etc. then you already have a mechanism for doing exactly what was just described. Implement what is either absolutely critical, more easily done up front, or fits the schedule as part of the main user story, and create additional stories for aspects you defer. The product owner will determine how important those other stories are relative to other functionality that could be developed, and thus the constraints will get implemented when they are the most valuable feature of the system awaiting implementation.

Generalize

We have said this already, but it bears repeating: define as many of the constraints as broadly as possible.

Repetition has many costs while you are working on requirements. These costs include the costs of repeating the analysis and/or writing, applying changes in many places if a detail changes, ramifications of inconsistency when one location is inadvertently missed, both developer and tester effort in trying to determine if there are any subtle differences in the repeated text, and both developer and tester in determining if there are cases where the repeated text is omitted.

[13] The quote that began this section can be rephrased as: "Security amateurs know how to secure things; security professionals know when you don't have to."

When constraints are universal, state them universally rather than as part of particular Actions (features or functions). When a constraint repeats but is less than universal, be creative with how you represent it. Having text describing the constraint and a matrix showing where it does and doesn't apply is often a good structure. Likewise, stating requirements that are generally true and then noting the exceptions can be clearer than repeating requirements in two thirds or more of the function descriptions.

Conciseness is a virtue in any recorded requirements, as long as the brevity isn't so severe that it causes ambiguity or becomes cryptic.

Remove Ambiguity

Ambiguous requirements are almost more insidious than incorrect requirements. With incorrect requirements, any reader will get the wrong answer, so such defects can be caught in reviews and walk-throughs. But with ambiguous requirements, a reviewer who is validating the requirement may interpret with the correct frame of reference and believe it specifies things correctly, but the coder could misinterpret it and implement something consistent with the requirement, but not acceptable. Or perhaps the coder gets it right, but the tester misinterprets creating a flurry of activity as everyone has to help determine where the disconnect is, and then which interpretation was the correct one.

Here are several causes of ambiguity to keep an eye out for:

- Unstated assumptions
- Poor language/grammar
- Loosely used terminology

The issue with unstated assumptions is that frequently we aren't conscious of the assumptions we make[14]. There are frequently details of the problem domain for an information system that the group providing the requirements knows inherently but that implementers of the system have no reasonable expectation of knowing. Other times assumptions may be due to not knowing there are alternatives. An example comes in systems that perform detailed calculations because there are actually multiple rounding rules to pick from (it isn't always round one half or more up and less than a half down).

To combat the assumptions issue, be in the habit of questioning whether there are assumptions being made, and explicitly state assumptions that are not well understood by all team members[15]. Even more important is for anyone reading a specification to call out if they find the need to make an assumption; this could be a reviewer, a developer, a tester, etc..

On the topic of grammar, this is not an invitation to nit-pick every issue found in written requirements. However, some grammar issues lead to requirements that could have multiple meanings. It is in that context that you need to call it out as an issue. In essence what is happening is that you are being forced to make an assumption about how to parse the language of the requirement. So like the assumptions above, call it out if you find yourself having to make an assumption in meaning as you read.

---

[14] This is likely due to cognitive biases that affect our beliefs. Awareness of these biases may help mitigate them; Wikipedia has an excellent list at http://en.wikipedia.org/wiki/List_of_cognitive_biases.

[15] It is okay to have implicit assumptions if it a reasonable expectation that everyone will have the same assumption. As a trivial example, if a requirement is to place a DVD in the computer, it is assumed that it will be placed in face down (label up), but that does not need to be stated.

We cited an example of a common term that is used loosely earlier: When you say "employees," do you mean all employees and contractors, or just all employees? "Employee" has a precise meaning, but it is often used colloquially to mean "staff." Look at words and phrases used in requirements with an eye for whether there are ways to misinterpret them. Again, when there are multiple viable interpretations available, an assumption must be made, so the same strategies apply; the most important strategy is still to always question when you find yourself having to make an assumption.

Re-Use

Re-using materials is a boon to efficiency, but it can equally well be a boon to quality (and thus security as we have discussed elsewhere). Materials that have already been used have already been validated, so are less prone to error than drafting from scratch. The other way of looking at the same idea is that if you know a particular artifact will be used over and over, it becomes more cost effective to spend time making sure it is well thought out. The following are key ways to avoid "reinventing the wheel":

- Find existing artifacts
- Stand on the shoulders of others
- Tool for re-use
- Incorporate into policy

Throughout the paper we have suggested existing artifacts your organization might have that would be an aid to functional security requirements analysis. These included items such as org charts, data classifications, and financial control policy documentation

By standing on the shoulders of others, we are suggesting you use prior work that is [legally] available to you. This could include repositories of patterns prior projects in your organization have developed, requirements documents from similar prior projects, and public resources. The authors intend to make the Patterns described in this paper available online in a repository (but described in more detail) for anyone to use. We will be adding new patterns as they are suggested to us.

When we say "tool for re-use" we are suggesting that on each project you look at materials and techniques you had to develop and you determine how to make the available for other projects in the future. This would in turn feed the "Find existing artifacts" and "Stand on the shoulders of others" strategies. In particular keep in mind if you developed a social group catalog, a data group catalog, a location catalog, or a limits catalog (e.g., for Transaction Limits).

Our last re-use strategy is directed towards security professionals. Security policy documents frequently focus on infrastructure controls and address application development mainly in the context of non-functional requirements and vulnerability management. While we have pointed out that most constraints are application-specific, some will turn out to apply to nearly every development project for the organization. Identify constraints and other functional requirements that are a matter for policy enforcement for all applications, and incorporate them into your information security policy.

5 Security Actions

For most information systems[16], user accounts and other means of managing user identity come into play. Thus for most systems the functional requirements must include administrative activities to create and

[16] The primary exception would be web applications offered anonymously to the public.

maintain user accounts. Administrative functionality in general can tend to get short shrift in the requirements process. After all, the administrative operations are not where most systems deliver their value to users; they are merely a means to an end. Just as constraints do not represent the "happy path" of allowing users to attain the system's value proposition, administrative functions need to be included in the process in order to achieve BSM's goal of having the requirements define all human/information interactions permitted by the system. While all administrative activities must be included, we will in particular provide some guidance around the actions that support security administration.

Just as with all other functional requirements, you will need to look at what the constraints are to be applied to Actions within administrative functions. These tend to be straight forward—for example most organizations know who they want to have creating new user accounts, increasing permissions of existing users, and who can reset passwords.

Broadly speaking, security Actions break down into three categories: Managing User Accounts/Identities, Managing User Permissions, and Security Actions by the User. The first two categories contain administrative Actions, and the third contains Actions performed by the end user of the system[17]. We will discuss each category in turn, and finish with some notes on how a functional requirements view of security fits into the system activity that must be implemented to have a secure system.

Be aware that many of the details of the Actions we are describing in this section will be prescribed by non-functional security requirements (password policies, multi-factor authentication requirements, system architecture dictates, etc.). We feel there are good resources (and generally good practices) available on establishing non-functional security requirements, so we will assume your project has robust non-functional security requirements and not focus on that aspect here.

For all of these actions, the functional requirements may turn out to be boilerplate for all future development activities. This is particularly true when there is a common security infrastructure (such as third-party tools to manage RBAC, ABAC, etc.) that all new information systems are expected to run atop. Keep in mind when first establishing the requirements that they are not just a matter of security policy, they are creating user experience and thus stakeholders representing the users should be at least consulted and ideally would be actively involved in the definition of the user experience for security Actions.

5.1 Actions to Manage User Accounts/Identities

This could be described as the "user lifecycle." The key point is to make sure you provide for all steps that will be relevant to your system. A typical set of Actions might look like this:

- Create user
- Update user information
- Suspend user
- Reinstate user
- Delete user

[17] Administrators are also users, and are sometimes "end users." To try to ease confusion, we will use the terms "administrative user" and "end user" where an administrative user has permission to perform administrative Actions and an end user has permission to perform non-administrative Actions.

Create user can be complicated by the steps to set one or more roles, other permissions, etc.. Create user can also turn out to be a multi-step, or even multi-person workflow where the Dual Controls Pattern applies.

Update user information may be a single Action, or it can be broken out into several smaller pieces. A common piece to break out is “reset password” since aside from creating users and later removing their access, password resets are by far the most common security administration activity for most systems (and in actuality probably occurs in most places even more often than creating and removing access).

5.2 Actions to Manage User Permissions

Creating the account is probably only a means to an end. That end is in using permissions to control what actions they may perform, as has been discussed heavily in this paper, and to be able to attribute actions to a particular user[18].

The most common means of granting permissions is by assignment of roles, and thus the Actions to consider in functional security requirements might be:

- Add user role
- Remove user role

In such a case it is assumed that all permissions are granted to the roles, so users acquire permission by virtue of their role or roles.

However, if your system will be allowing permissions to be set for individual users, then the Actions could resemble:

- Set user authorization
- Change user authorization
- Revoke user authorization

Change would only apply if the authorization was other than a “yes/no” proposition. In examples we have seen, there can be a dollar limit on transactions that a user is authorized to either initiate or approve—thus we have an authorization setting that can be changed.

In systems where there is a notion of “ownership,” there can be additional administrative Actions that pertain to maintaining ownership data. Assigning ownership of an Object to a different user would be an example. ACL-based implementations where the control is per user account rather than per role would also entail management operations.

In your system, consider the operational implications of the constraints you have imposed, and wherever you have an operational component where an administrative user is able to or required to intervene, make sure you have appropriate functions defined.

[18] We generally consider audit logging, which is recording actions a user takes, to be a non-functional requirement.

5.3 Security Actions Performed by the End User

There are critical security activities performed by the end user in the course of using an information system. Common Actions would include:

- Logging in
- Logging out
- Updating user information
- Delegating authorization
- Requesting access (either to obtain an account, or to gain additional permissions)

We will provide brief guidance on each for aspects that may not always be considered. There is a great deal of further information available for readers wishing to dive deeper into these topics.

Logging In

When contemplating the functional requirements for logging in, the two key things to note are (1) what happens when unsuccessful, and (2) what are the audit logging requirements.

"Failed login" is frequently a complex, multi-layer workflow, with options to use pre-stored information to re-obtain credentials, etc.. Previous reminders that a well-built system should assist legitimate users in achieving their goals also applies here.

While the fact of logging in is frequently a security event requiring logging to meet audit requirements, many sub-steps in the process may also be worth considering. The obvious one is to record failed login attempts. Less obvious is that you may want to record the success individual steps within a successful login. If your system requires multi-factor authentication, such as with the use of a physical token, being able to demonstrate that a successful login included verification of the token can be critical in non-repudiation disputes.

Logging Out

We will provide three quick thoughts on this topic. If you allow the user to log in, you should give an explicit option to log out. When a user is logged out, it should result in positive indication that they have successfully done so. After logging out, it should be a relatively simple matter to log in again (e.g. don't place the user on a static page with no links).

Updating User Information

This pertains to user information that is part of security operations (credentials, and mechanisms for re-obtaining credentials if forgotten). It is assumed that if the system allows users to record information about themselves (pictures, contact details, etc.), that maintenance of such information has already been contemplated.

The key Actions here are "Change Password" and an update Action for information used to re-obtain a username or password if forgotten.

Delegating Authorization

Some systems include functions that allow users to manage permissions of other users. Almost all social media includes this, as do shared calendaring systems.

A particular security Action is to "delegate" authority, which is to say, to grant to another user a permission that the delegating user possesses. The key detail to consider here is that permission to *perform* an action is different than permission to *delegate*. Thus when delegating you need to clearly note whether you are only giving the user the ability to delegate a permission, or whether you are also giving the ability for the user to delegate the delegation permission itself. Typically these are managed as two separate grants. For example, if you own a resource in a system, you may be considered an "administrator" for that resource and also be able to set another user as also being an "administrator" for the resource; but an additional permission you could grant is for that other user to also be able to designate additional "administrators."

Requesting Access

Some systems allow a user to create their own account, in which case this would be the same as the "Create User" Action discussed in Section 5.1, Actions to Manage User Accounts/Identities. In other cases, an anonymous user to a web site may be given the opportunity to request access, which would trigger a workflow as was discussed in Section 5.1 as part of the "Create User" Action.

The other version of this comes as a result of constraints where access is denied, but an option to request access is provided. This would occur as part of a No-Go Path, as discussed in Section 4.6, Go path and No-Go Path. We noted at that section that Microsoft SharePoint provides an example of this technique in action.

5.4 System Actions at Runtime

For information security, identification, authentication, and authorization are key, however, the details of *how* each of these operations takes place are not functional requirements. Within the application when it is built, there will be technical mechanisms for managing these. The functional requirements should not dictate those technical mechanisms, and whenever possible should not even be concerned with them.

For example, when a user first uses a system, there will be a business requirement that the user identify who they are and provide credentials that allow the system to authenticate that they are who they say they are. That is the human-centric portion of the process. How that information is technically vetted is not a matter for functional requirements.[19] Nor are the equally important technical mechanisms by which the subsequent transactions performed by the user are validated by the system, wherein the system verifies that it is still working with the same user.

Likewise, authorization at a functional level occurs at a time far removed from when the user performs a particular action. However behind the scenes of the system, whenever a user performs an action the system works to consult the artifacts left behind by the functional authorization process to ensure that the user is performing a valid action. Both are crucial parts of the application of information security to an information system, but one of them is a matter of functional behavior and user experience, while the other is a technical matter best addressed in technical design.

[19] In cases where it is necessary, functional requirements should specify a level of assurance that the user is authentic.

6 Requirements Quality Assurance

As we mentioned earlier, quality assurance is a broad topic and more than we can take on here. On the other hand everyplace a system behaves differently than it ought to poses the potential for misuse, risk of loss to the organization (or user), or compromise of integrity and availability.

Because BIS is focused on performing information security tasks with human behavior in mind, we think the most promising trend is the use of checklists. To this end, here are our favorites:

- http://www.jot.fm/issues/issue_2005_11/column4.pdf
- http://elearning.tvm.tcs.co.in/rwi_html/SRSCheckList.pdf
- http://www.cdf.toronto.edu/~csc340h/winter/assignments/inspections/JPL_reqts_clist.pdf

7 Post Conditions

A post condition is a statement of what must be true when a user's Action is complete, and can be stated both for successful completion and unsuccessful completion. For example, when an account transfer is completed successfully, the funds must be credited to the destination account and debited from the source account; if the transfer fails, the source account must still contain the funds, and the destination account must not have been increased. They are a means of explicitly calling out conditions that otherwise might be left as assumptions.

They are a sort of formalized hygiene—think of it as an assertion: "At this point, the following must be true, or we have an exception from a functional point of view." Many organizations that employ Use Cases will include "Post Conditions" as a part of their use case template. Regardless of whether Use Cases are part of your development practice, defining post conditions for every operation is a matter of organizational style. If you do use them in general, we advise you use them for both success and failure conditions. If you don't use them for in general, we advise that you note situations that have greater than normal potential for integrity issues and use them in those cases.

8 Conclusion

We have explained how the majority of current information security activity in gathering requirements misses the mark when it comes to engaging non-security team members, especially non-technical stakeholders. Because of this, there is a gap where information security practices today insufficiently address safety concerns in software use, and threats in the form of misuse of privilege (as opposed to mis-acquisition of privilege).

To address this gap, we have presented a simple, pragmatic, check-list driven approach to engaging the stake holders of a system to provide the requirements that will appropriately control for safety risks and misuse of privilege. This approach is founded in and inspired by current behavioral and cognitive research, and Behavioral Information Security's focus on how humans behave and interact with information. The founding assertion is that to have robust and secure information systems that the functional requirements must define all human/information interactions permitted by the system.

The techniques presented make no assumptions of the SDLC or requirements gathering techniques employed, and in fact impose no constraints on how these activities are performed. Within any requirements gathering practice, it is possible to easily improve the robustness of the requirements by:

1) including consideration of constraints, the means of enforcing constraints, security actions, and post conditions, and
2) improving the efficacy of requirements by employing quality assurance techniques, the practice of guiding by checklist, addressing ambiguity, employing prioritization and generalization, and driving re-use.

Finally, we proposed the value of bringing the "design pattern" concept used in system design across to system requirements gathering. To this end we have proposed a number of requirement patterns pertaining to functional security constraints, and will be bringing a new repository to the web to house a pattern language of requirements design patterns.

All of these techniques are drawn from the authors' practical experience. The next step is to test the techniques as explained by implementing the approach in a number of application development environments. We will be doing so in our current consulting work, but are also looking for others to help. Please contact us if you are interested in contributing to the expansion of these ideas.

9 Bibliography

Here is a list of reading material that either influenced our thinking or helped us write this paper.

Anne Adams, Martina Sasse, *Users are not the Enemy.* 1999. http://dl.acm.org/citation.cfm?id=322806

Brad J Cox, *Policy Based Access Control (PBAC) for Diverse DoD Security Domains.* 2011. http://virtualschool.edu/cox/pub/PBAC.pdf

Charles B Haley, Robin Laney, Jonathan D Moffett, and Bashar Nuseibeh, *Security Requirements Engineering: A Framework for Representation and Analysis.* 2008. http://oro.open.ac.uk/10058/1/01435458.pdf

Donald Firesmith, *Specifying Good Requirements.* 2003. http://www.jot.fm/issues/issue_2003_07/column7/

Donald Firesmith, *Quality Requirements Checklist.* 2005. http://www.jot.fm/issues/issue_2005_11/column4/

Gunnar Peterson, *Collaboration in a Secure Development Process Part 1.* 2004. http://www.arctecgroup.net/ISB0905GP.pdf

Inger Anne Tøndel, Martin Gilje Jaatun, and Per Håkon Meland, *Security Requirements for the Rest of Us: A Survey.* 2008. http://dl.acm.org/citation.cfm?id=1340062

Jan Jürjens, UMLsec: *Extending UML for Secure Systems Development.* 2002. http://dl.acm.org/citation.cfm?id=719625

M A Sasse, S Brostoff, D Weirich, *Transforming the 'weakest link' — a human/computer interaction approach to usable and effective security.* 2001. http://dl.acm.org/citation.cfm?id=592514

Michael L Brown, *Software Systems Safety Design Guidelines and Recommendations,* 1989. http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA209832

Nancy R Mead, Eric D Hough, Theodore R Stehney II, *Security Quality Requirements Engineering (SQUARE) Methodology.* 2005. http://www.cert.org/archive/pdf/05tr009.pdf

Oracle, *Fine Grained Authorization: Technical Insights for Using Oracle Entitlements Server.* 2011. http://www.oracle.com/technetwork/middleware/oes/oes-product-white-paper-405854.pdf

Oracle, *Oracle Entitlements Server.* 2008. http://www.oracle.com/technetwork/testcontent/oes-entitlements-133195.pdf

Richard Bender, *The Ambiguity Review Process.* 2004. http://benderrbt.com/Ambiguityprocess.pdf

Rudolph Araujo, Shanit Gupta, *Design Authorization Systems Using SecureUML.* 2005. http://www.mcafee.com/us/resources/white-papers/foundstone/wp-design-authorization-systems-secureuml.pdf

Sascha Konrad, Betty H.C. Cheng, *Requirements Patterns for Embedded Systems.* 2002. http://dl.acm.org/citation.cfm?id=731467

Torsten Lodderstedt, David Basin, and Jürgen Doser, *SecureUML: A UML-Based Modeling Language for Model-Driven Security.* 2002. https://edit.ethz.ch/infsec/research/publications/pub2002/SecureUML.pdf

10 Appendix: Styles of Requirements Capture

Different teams will handle capturing requirements in very different ways. Our goal was to have an approach that would enable any team to be able to incorporate the additional focus on functional security requirements. This section will highlight how it fits in to several of the most prevalent styles of eliciting the requirements and of recording the requirements.

10.1 Elicitation Techniques

Written By the Business

The least formal method, and the method where there are no dedicated requirements analysts involved in the creation of the requirements is where the business documents their requirements in advance and provides them to the technology group to implement. In this model, it would be preferable if the business stakeholders creating the requirements would work with a checklist based on this paper to ensure completeness by considering the various questions posed herein as they work. In absence of this, the technology group can ask the questions posed here during a joint review of the requirements documentation. In the former the business would take on more responsibility for facilitation or business analysis, and in the latter the technology group would be doing so.

In Facilitated Session

This style of elicitation goes by many names including “JAD” (Joint Application Design), etc., and some Agile development practices may employ sessions the fit this model. The key feature from this paper’s point of view is that there is a professional facilitator or analyst leading a discussion session during which the requirements are dictated and recorded. We encourage the leader of such sessions to base all work on checklists, but in particular to use a checklist that covers the topics we have listed regarding functional security requirements.

In Collaboration Between Technical Team and Users

Here we are talking about less formal sessions where the developers and users sit down together to discuss what the system must do. This would differ from facilitated sessions only if there is not a designated leader for the session who is trained in facilitation for elicitation of requirements. This can occur both when the process is very unstructured (small environments where developers and users sit down informally to discuss needs) as well as for certain Agile processes.

We would suggest in such cases that the team essentially follow one of the two models above. They could have a checklist available to all participants and be in the habit of consulting the checklist as the group works to ensure all aspects of requirements are being addressed. Otherwise they could also designate one team member to act to monitor completeness; they would track discussion against any available checklists or review standards, and inject questions if they the discussion was failing to contemplate an area or concern.

10.2 Recording Techniques

Functional Decomposition/Hierarchical Requirements

In this model, features are grouped into hierarchies, and specific requirements are listed under the feature. They will typically each have a discrete number to reference them (such as 3.14.2, which would be a detail underneath 3.14).

Constraints and post conditions that are specific to a feature would be sub-requirements under that feature; “No-Go Path” information would be details on the constraint. Global constraints would generally have their own section of the hierarchy.

There should be little difficulty in adding the new topics to existing requirements, and the ability to establish topic templates for new features can help ensure that constraints and post conditions are considered.

Prose Requirements

In this model, system behavior is listed in narrative descriptions of the desired system. Because there tends to be little structure in such requirements, adding consideration of constraints and post conditions should be easy to do, but difficult to do reliably as there is generally not a significant template structure to such requirements.

Requirements in Use Cases

Use cases tend to be the most user-centric form of requirements. If you are using use cases, it is common practice to include pre-conditions and post conditions. A pre-condition effectively is a constraint, and thus any constraints that can be determined before the start of execution of the use case have a solid home, although typical use case formats don’t lend themselves to talking about No-Go Paths.

Constraints during the execution of the use case can be more of a problem. If you are showing flows with UML Activity Diagrams, there is a format for constraints that can be followed. A constraint on a UML diagram is any text written: {inside of curly braces}, and include an option to list what happens as a result (essentially a No-Go Path). Thus activity diagrams can be annotated to show constraints. Otherwise you will need to discuss the constraints in other sections of the use case, or add a section explicitly in your use case template. It is also common practice when using use cases, to also have a separate repository of business rules; many constraints would fit within such business rules.

Story Cards

Story cards are used in many SDLC practices based on Agile. Typically the goal is for all requirements for a given user story to be able to be recorded on a 4”×6” index card. While the authors think that story cards are wonderful for organizing and managing a software project, that they are somewhat lacking when it comes to supporting a process of having robust documentation of requirements. While it is acknowledge that in Agile we must put people over process and running systems over documentation, there is room for judicious use of both process and documentation.

Our preference would be that the story card be augmented by some other form of written support document that lists qualitative aspects of a story, which might include constraints and post conditions. For teams where that level of writing is considered out of line, it is encouraged that the discussions of requirements then include the topics of constraints and post conditions; it is better for such needs to at least be discussed even if it is deemed impractical to record them.

About the Authors

John Benninghoff is the owner of Transvasive Security. John started Transvasive to develop the idea of Behavioral Information Security, which combines Information Security with current research in the fields of economics, cognitive science, and organizational theory. John can be reached at john@transvasive.com.

Karl Brophey is the owner and principal consultant for Brophey Consulting. Brophey Consulting works to bridge business product strategy and operations to information technology planning and delivery. Karl has spent his career helping companies meet their business goals using custom software, encouraging people who think differently to communicate effectively, and focusing on user experience and business value. Karl can be reached at karl@brophey.net.