5-2013

Categorization of Security Design Patterns

Jeremiah Y. Dangler
East Tennessee State University

Categorization of Secure Design Patterns

A thesis

presented to

the faculty of the Department of Computer and Information Sciences

East Tennessee State University

In partial fulfillment

of the requirements for the degree

Master of Science in Computer Science

---

by

Jeremiah Y. Dangler

May 2013

Dr. Martin L. Barrett, Chair

Dr. Phillip E. Pfeiffer

Dr. Michael R. Lehrfeld

Keywords: Security, Design Patterns, Security Design Patterns

ABSTRACT

Categorization of Security Design Patterns

by

Jeremiah Dangler

Strategies for software development often slight security-related considerations, due to the difficulty of developing realizable requirements, identifying and applying appropriate techniques, and teaching secure design. This work describes a three-part strategy for addressing these concerns. Part 1 provides detailed questions, derived from a two-level characterization of system security based on work by Chung et. al., to elicit precise requirements. Part 2 uses a novel framework for relating this characterization to previously published strategies, or patterns, for secure software development. Included case studies suggest the framework's effectiveness, involving the application of three patterns for secure design (Limited View, Role-Based Access Control, Secure State Machine) to a production system for document management. Part 3 presents teaching modules to introduce patterns into lower-division computer science courses. Five modules, integer overflow, input validation, HTTPS, files access, and SQL injection, are proposed for conveying an aware of security patterns and their value in software development.

ACKNOWLEDGMENTS

I would never have been able to finish my thesis in a short amount of time without the guidance of my committee members and support from my wife. First, I would like to thank Dr. Martin Barrett, who is a good friend, for his continued support throughout my graduate studies and his consistent guidance through the thesis process. Many thanks to Dr. Phil Pfeiffer for his insightful edits and recommendations to my thesis and to Dr. Michael Lehrfeld for his encouragement in the thesis process.

Finally, my greatest thanks to my wife, Dr. Marsha Dangler, for her constant support and encouragement throughout my academic endeavors.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Security in software engineering has become an important research area, in part because of recent events that exposed security vulnerabilities in major systems. Modern enterprises store large amounts of sensitive and confidential information in information systems. This information must be protected from unauthorized access and modification to assure its value and to protect those who would be harmed by its disclosure or alteration.

Numerous incidents attest to how breaches of policy can result in lost data or financial loss. Siemens, a German-based international corporation, had its operations control system attacked by a highly sophisticated virus, called Stuxnet, which wreaked havoc on systems. [14] In early 2011, the Stuxnet virus "is thought to have caused severe damage to Iranian uranium" centrifuges, possibly setting back the nation's nuclear power policies. [13] Another highly publicized breach involved an April 2011 compromise of Sony's PlayStation Network (PSN). Sony reported that information such as a user's name, address, email, birthday and PSN online ID were stolen along with the possibility of credit cards associated with the user's account. [18]

Enterprises use security policies to define how a system should be accessed, who can access it, and when it can be accessed. These requirements for system security are often treated as non-functional requirements (NFRs) for system implementation. NFRs, which describe how a system should perform, address concerns such as operational costs, reliability, and system robustness. These types of requirements are often not formulated until after a system is under development. [3] This is due, in part, to the difficulty of soliciting and documenting NFRs. Unfortunately, delaying the formulation of security NFRs can lead to incomplete, inappropriate, or inadequately realized requirements, leaving the resulting system vulnerable to the theft of sensitive data and loss of service. [8]

To meet security NFRs, better methods for soliciting and documenting security quality attributes are needed. Soliciting NFRs early in the system development process should promote the development of well documented specifications. These specifications would then be more likely to be captured in a system's design. The likelihood of their being realized effectively could be further increased through good design practices, including the use of well-documented design patterns for secure design. Design patterns are reusable solutions to common problems that occur in software development. They include security design pattern, a type of pattern that addresses problems associated with security NFRs.

This thesis is concerned with strategies for promoting the integration of security NFRs into software development. To this end, it first presents a method of classifying security issues using a hierarchical structure (cf. Chapter 2). This hierarchy decomposes security, a broad-based concern, into multiple, smaller concerns that should be easier to translate into precise specifications. To aid in this translation, a series of simple, focused questions has been developed for identifying a system's quality attributes, relative to this hierarchy. Answers to the questions are documented as security NFRs.

This hierarchy of security-related issues is then used to categorize 30 security design patterns that can be used to satisfy security NFRs (cf. Chapter 3). This catalogue includes a short description of the pattern, a characterization of the specific NFR with which each pattern is concerned, the stage of the software development life cycle in which each should be applied, and a reference to a more complete description of that pattern. This material is intended to provide designers with a way to easily identify the pattern that applies to their security concern.

To assess its effectiveness as a tool for secure software design, this catalogue was then used as the basis for a case study related to improving the design of a production software system (cf. Chapter 4). This system, the Online Tenure and Promotion System (OLTP), manages

documents and other artifacts for East Tennessee State University's (ETSU) tenure and promotion process. This is a year-long process whose requirements include multiple checkpoints for generating and freezing documentation, reasonably complex criteria for determining when these checkpoints are met, and the need to allow selected participants to assume multiple roles: e.g., as committee member and unit head. This case study demonstrates the effectiveness of using patterns to simplify maintenance as well as the need to apply multiple patterns to a system to achieve the desired level of quality.

A final concern, the need to educate students about the value of these patterns, was addressed by developing modules on security that can be integrated into a standard computing curriculum (cf. Chapter 5). Introducing topics as the student progresses through the curriculum allows these modules to be tailored to a student's current skill set, while emphasizing the need to consider security in all phases of software development. Five modules for freshman and sophomore level courses are presented here. Topics include integer overflow, file access, HTTPS, SQL injection, and input validation. Future work will include additional modules for upper-division classes.

CHAPTER 2

SECURITY

Security in software systems is the protection of an enterprise's data against unwanted access or modification. [4] Security is a diverse challenge that includes practices that range from escaping user input to protecting against unauthorized access to sensitive information.

Security is traditionally characterized as a non-functional requirement (NFR), a requirement for how a system should operate. [8] Security, unfortunately, is often engineered into systems late in the design process or after implementation. This results in a system that is less maintainable and less trustworthy in terms of putting an enterprise's data at risk. [3]

NFRs are often slighted in systems development because their requirements are difficult to formulate. The strategy for managing security-related NFRs described here first decomposes security into a hierarchy of more specific NFRs, then relates the decomposed requirements to a catalogue of best practices for security development. This use of decomposition to reduce a vague, broad-based NFR to a set of more precise and verifiable requirements is one that has been discussed at length in the literature of NFRs. [15] The practices for security assurance that have been catalogued here are ones that are routinely neglected in traditional systems development, relative to the stages of the development process where they can most effectively be applied.

The required decomposition of the security NFR was accomplished in two steps. Chung's [8] work on NFR decomposition was first used to decompose security into four first-level properties. One, confidentiality, is the protection of an enterprise's sensitive data from unauthorized access. A second, integrity, is the protection of an enterprise's data from unauthorized modification. A third, availability, is the protection of an enterprise's data from becoming unavailable from authorized users. [8] The fourth, accountability, is the association of actions that affect the enterprise's data with the person or persons who perform those

actions. [16]. The hierarchy allows software engineers to focus on specific security concerns and provides a vocabulary to better communicate specific security concerns.

In order to support the development of precise requirements for system security, each of these properties was then decomposed into a second set of properties. These properties are confidentiality, integrity, availability, and accountability. This second-level decomposition proved precise enough to develop a set of detailed questions for eliciting requirements: one set per property.

A Detailed Characterization of Security NFRS

Chung et al. [8] created a framework for capturing and documenting NFRs. This framework associates every NFR with a particular topic: i.e., a consideration related to system operation. For example, a topic could be an Account that needs protection from unauthorized access or a Service that needs to respond within a specific time. This framework was used to decompose security, considered as an NFR, into four more specific NFRs: confidentiality, integrity, availability, and accountability.

Requirements for the confidentiality of an enterprise's data are typically derived from that enterprise's security policies as they relate to this data. These requirements, as noted in [8], can be further decomposed into requirements for user authentication and authorization. Requirements for confidentiality should strongly influence how a system is designed, how access is controlled, how processes are controlled, and how users are authenticated. Breaches of confidentiality can often be attributed to mistakes in system implementation that are exploited by malicious parties to gain unauthorized access to the compromised data. An example of a recent breach in confidentiality occurred in 2011 when hackers infiltrated Sony's PlayStation Network resulting in the theft of over 77 million account holders' personal information. This breach has cost Sony over $171 million, in terms of lost revenues and costs to secure its

networks. [17]

Requirements for the integrity of an enterprise's data are derived from that enterprise's operating policies. Chung et al. [8] cite data's completeness, precision, and validity as considerations in assessing its integrity. A fourth concern is timeliness, or the extent to which a system's information is current. Policies for data integrity must clearly define who can modify what data. Data corruption can be attributed in part to the deliberate modifications to datasets or processes by unauthorized users: for example, the alteration of experimental data to suggest a desired outcome. This corrupt data, when used by other processes or systems, can further contaminate other datasets and processes.

Requirements for the availability of an enterprise's services are derived from that enterprise's quality service policies. Barbacci, for example, notes that "availability is required for security critical aspects of any given system." [4] These requirements can be further decomposed into requirements for usability, reliability, and compatibility. Resources can be rendered unusable by malicious or inadvertent denial of service attacks: bombardments of resources with more requests than they can handle. [4]

Requirements for the accountability of an enterprise's services can be derived from an enterprise's policies for employees responsible for their actions. Requirements for accountability can be decomposed into requirements for logging, monitoring, and reporting users' actions. Logging allows the enterprise to determine which user may have breached system confidentiality or affected data integrity. For example, medical record systems will log medical providers who access patient information. This log can be used to determine if a provider breaches patient confidentiality by accessing records of individuals in ways that are not allowed by government regulations.

Since this first-level decomposition proved too coarse-grained to use as a basis for developing questions for requirements elicitation, these four NFRs were decomposed a second

time. This second round of decomposition and the increased level of complexity that ensued from it suggested a need for strategies for documenting the decomposition. Chung et al. [8] suggest two, both of which are used in what follows. One employs a graph that uses cloud symbols to denote NFRs and lines to represent inter-dependencies. A single line forms an arch with the subtypes to indicate that the four subtypes should be met to satisfy the security NFR. An example of the notation with an Account is given in figure 1.

Figure 1: Security NFR Subtypes

The other uses a longhand notation in the form:

> confidentiality [Topic] AND integrity [Topic] AND accountability [Topic] AND availability [Topic] SATIFICE security [Topic]

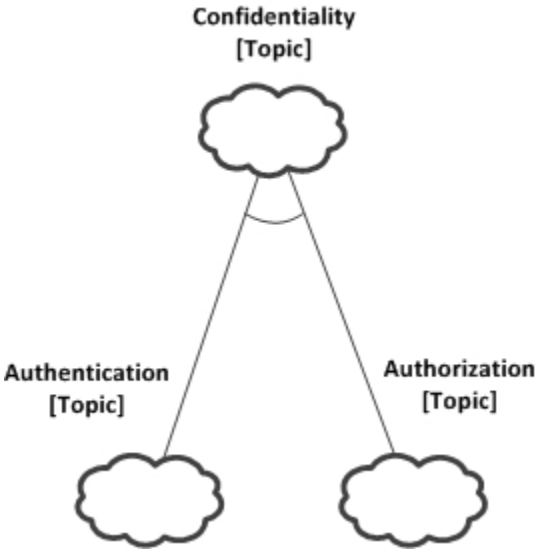
An example of the notation with an Account would be

> confidentiality [Account]

Confidentiality

Using Chung's method, confidentiality was decomposed into authentication and authorization subtypes (cf. 2). Authentication is the confirming of a user's or entity's identity.

Authorization can be done through physical confirmation, using devices such as biometrics or key card access, or specialized information like a secure password or a unique authentication key. Authorization is the granting of access to system resources such as accounts, files, or processes. Authorization is typically granted after a user has been authenticated. What resources a user is authorized to use may be stored in an access control list or based on a user's role within the system. [8] Figure 2 shows confidentiality decomposed using Chung's notation.



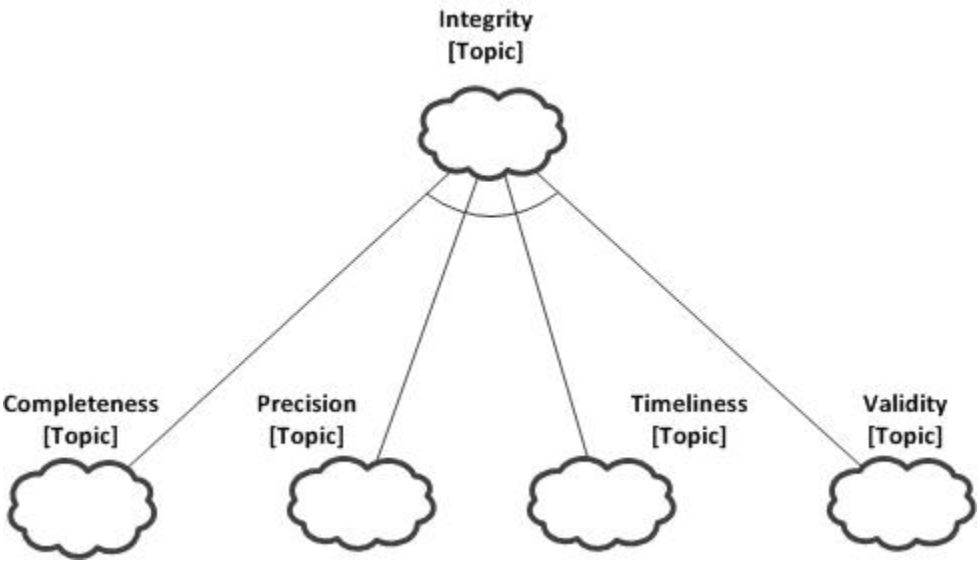
authentication [Topic] AND authorization [Topic] SATIFICE confidentiality [Topic]

Figure 2: Confidentiality Subtypes

Integrity

Using Chung's method, integrity was decomposed into completeness, precision, timeliness, and validity subtypes (cf. 3). Completeness is the storing of all information that a system is required to have. Precision is the degree to which stored information models true or accepted values. Timeliness is the currency of the system's information. Validity is the

correctness of that information, relative to the entities that it is intended to model. [8] Figure 3 shows integrity decomposed using Chung's notation.

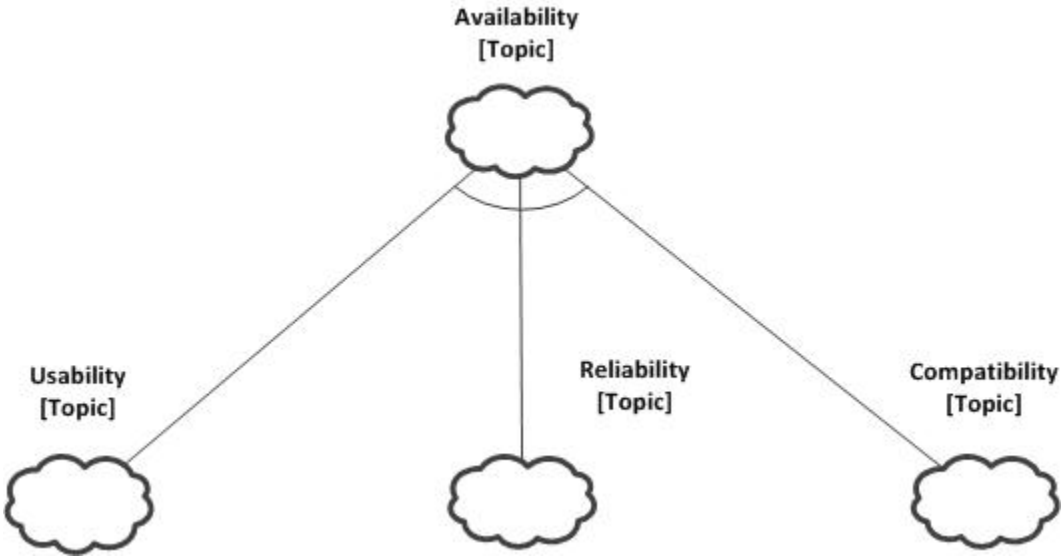


completeness [Topic] AND precision [Topic] AND timeliness [Topic] AND validity [Topic] SATIFICE integrity [Topic]

Figure 3: Integrity Subtypes

Availability

Using Chung's method, availability was decomposed into usability, reliability, and compatibility subtypes (cf. 4). Usability is the amount of time and effort required to use a program. Reliability is a system's ability to function in adverse circumstances, including attacks and component failures. Compatibility is the degree to which a program operates securely in a given environment. This includes the hiding of sensitive information from users who are not authorized to have that information. [8] Figure 4 shows availability decomposed using Chung's notation.

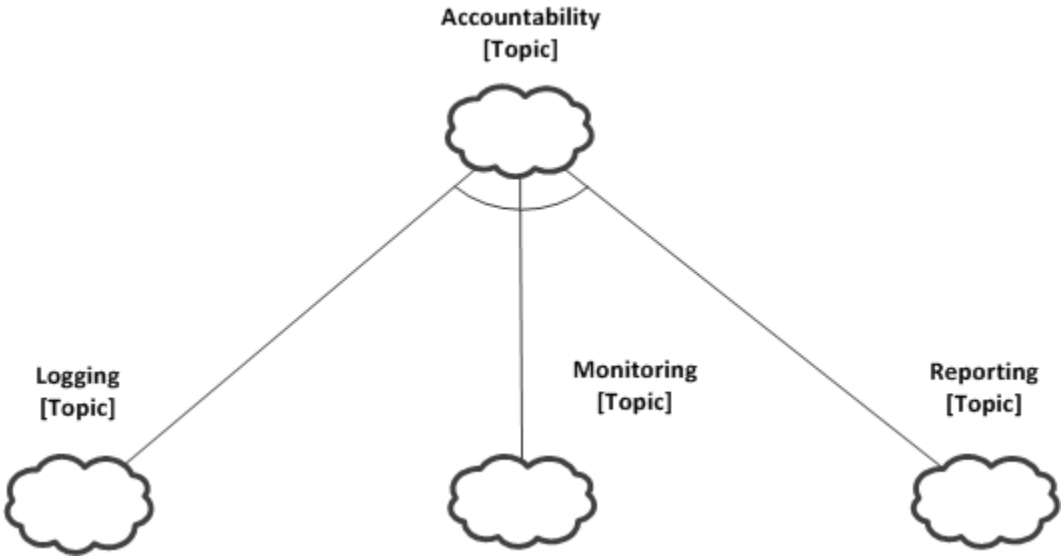


usability [Topic] AND reliability [Topic] AND compatibility [Topic] SATIFICE availability [Topic]

Figure 4: Availability Subtypes

Accountability

Using Chung's method, accountability was decomposed into logging, monitoring, and reporting subtypes (cf. 5). Logging is the recording of actions performed in or by a software system. Monitoring is a system's awareness of actions that may require notification to a third party. Reporting is the relaying of information to a third party. [8] Figure 5 shows accountability decomposed using Chung's notation.



logging [Topic] AND monitoring [Topic] AND reporting [Topic] SATIFICE accountability [Topic]

Figure 5: Accountability Subtypes

Soliciting NFRs

This decomposition yielded a list of detailed concerns to address when soliciting a system's security requirements, together with a list of sample questions for soliciting them. The questions below, one of this work's contributions, should be detailed enough to elicit useful responses from a system's stakeholders. Each question was derived from the definition of its corresponding NFR.

Authentication Should the system authenticate a user before allowing access to the system?

Authorization Should the system require a user to be authorized to access data before allowing the user to access sensitive data?

Completeness For each type of data that a system stores, under what circumstances must

that data be stored in full? For those types of data that can sometimes be stored in part, when is it acceptable to do so? Also, what degree of incompleteness is acceptable?

Precision How precise, in significant digits, should numeric values be stored within a system? Should names include middle names?

Timeliness For each type of data that a system presents to its users, under what circumstances must the system present the most recent version of that type to its users? For those types of data for which the system may present an older version of that data, how old may that version be?

Validity For each type of data that a system stores, under what circumstances may that data be invalid?

Logging What events that occur within a system, if any, should the system track?

Monitoring For each type of resource that the system supports, should the system provide monitoring for that resource? And, if so, how timely should the data be?

Reporting To which third parties, if any, should the system report events or other information about its operation? What sorts of information should be reported?

Usability Should the system allow users to view functionality that they are not authorized to use? If so, should they be allowed to access this functionality, then be subjected to errors when using unauthorized functionality?

Reliability Under what circumstances is it acceptable for a system that is under attack to be taken offline for any length of time?

Compatibility In what environments (e.g., Linux and Windows) must the system run?

The decomposition of security into a hierarchy simplifies the process of soliciting security NFRs. The following chapter describes a second use for this hierarchy: the classification of security design patterns relative to the concerns they are intended to address.

CHAPTER 3

SECURITY DESIGN PATTERNS

Software design patterns are solutions to problems that arise regularly during software design. They are meant to serve as readily applicable, time-saving strategies for software development. The structured documentation that accompanies a properly defined pattern allows developers to quickly identify and apply patterns to a given problem. [11]

A review of the literature on design patterns conducted for this research from August 2012 to December 2012 yielded 30 patterns that are concerned specifically with secure systems design. This chapter catalogues patterns identified during the review. These patterns have been categorized according to whether they're concerned with a system's architecture, design, or implementation. Within these levels, patterns are further categorized as to whether they're concerned with the confidentiality, integrity, authorization, or accountability subtypes of the security NFR, as discussed in chapter 2.

Overview of Design Patterns

The notion of a design pattern was developed by Christopher Alexander in his work on reusable strategies for architecting space and structure. Alexander's patterns [1] characterize what he referred to as "timeless ways of building": recurring practices in structural composition. Alexander says, "each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice". [1]

In 1987, Beck and Cunningham [5] applied Alexander's idea of patterns to the development of graphical user interfaces in an object-oriented language, Smalltalk. Cunningham and Beck developed five patterns for developing a Smalltalk user interface. They published their

results at the OOPSLA-87 workshop. [5] Between 1990 and 1992, Eric Gamma, Ralph Johnson, John Vlissides, and Richard Helm, a.k.a. the Gang of Four (GoF), compiled a catalog of patterns. These patterns were subsequently published as what became known as the Gang of Four (GoF) book [11]: the first and highly influential book of software design patterns. This book was followed by the publication of many subsequent pattern collections, such as Grady Booch's Handbook of Software Architecture [6], which currently includes about 2,000 patterns. Patterns are also the theme of a series of worldwide conferences sponsored by the Pattern Language of Programming group. The conferences' proceedings are published as Pattern Languages of Program Design (PLoPD). [7]

According to [11], a software design pattern should consist of four elements. These elements include a **pattern name**, a **problem** described by a set of scenarios, a **solution** of how that pattern may be implemented, and a discussion of the **consequences** of applying that pattern.

A pattern's **name**, according to [11], should describe "a design problem, its solutions, and consequences" in a few words. Naming a common design strategy gives developers a common vocabulary that facilitates communication during software development. While a pattern may be difficult to name, giving a clear name is crucial for communicating that pattern's significance.

The **problem**, a set of scenarios, describes the situations in which the pattern is needed. This description may restrict when the pattern can be used and how it may be applied.

The **solution**, a characterization of the pattern's implementation, describes the design's component parts and their responsibilities; how these parts collaborate; and the relationships between the parts. This characterization is an abstract description of how the pattern's elements are generally arranged. Since the patterns are not language dependent, the solution might not give a concrete implementation. Pattern developers often document their

patterns using the Unified Modeling Language (UML). This description, however, is often supplemented with examples in specific languages.

The **consequences** are the "results and trade-offs" of applying that pattern. Alternative designs are evaluated to document the costs and benefits of using the pattern.

The literature on design patterns includes interactions between patterns known as pattern relationships. Buschmann et al. [7] detail three common pattern relationships: **pattern complements**, **pattern compounds**, and **pattern sequences**.

Pattern complement is providing missing parts or contrasts other patterns by providing an alternative solution. One example is using the DISPOSAL METHOD, to compliment the FACTORY METHOD, to manage the creation and destruction of objects in the same design.

Pattern compound captures repetitive subdomains of common and identifiable patterns that can be viewed as a single decision to a common problem. An example of pattern compounds is implementing COMMAND as a COMPOSITE, resulting in a cohesive pattern, COMPOSITE COMMAND.

Pattern sequences are the generalization of a "progression of patterns and the way a design can be established by joining predecessor patterns" to form an overall solution for a given context. One example of this is the creation of communication middleware by joining BROKER, LAYERS, WRAPPER, FACADE, REACTOR, and other patterns. [7]

Patterns can also be aggregated to form **pattern languages**, a term first used by Christopher Alexander. [1] In the context of software engineering, a pattern language "defines a collection of patterns and the rules to combine them into an architectural style. Pattern languages describe software frameworks or families of related systems". [9] Bushmann et al. [7] use the term pattern system in place of pattern language. A pattern system defines a collection of patterns used during architecture design. Included with the design pattern

are descriptions of how to implement and combine patterns. Pattern systems should help a software system fulfill both functional and non-functional requirements. According to Buschmann et al. [7], a collection of patterns must meet five requirements to be a pattern system:

- Comprise a sufficient base of patterns
- Describe all patterns uniformly
- Expose the various relationships between patterns
- Support the construction of software systems
- Support its own evolution

Security Design Patterns

Security patterns are software design patterns that describe security mechanisms such as logging and access control. [10] The first use of the term "security patterns" occurred in 1997 paper by Yoder and Barcalow. [22] Prior to this paper, researchers had published several models for secure systems but had not referred to them as "patterns". More recent work includes books by Schumacher [16] and papers by Viega [21]

The following catalogue of security design patterns could have been presented using the name, problem, solution, and consequences template for pattern description developed by the GoF. For brevity, however, the list below provides just an abbreviated description of each pattern, along with a reference to the pattern's original source. Patterns have been grouped by the level of software artifact to which they pertain: i.e., architectural-level patterns, which specify how large-scale components interact; design-level patterns, which specify how elements of a single component (e.g.., class) interact; and implementation-level patterns,

which are applied to low-level security concerns. Within each level, patterns have been grouped according to the type of second-level NFR to which they apply: i.e., confidentiality, integrity, availability, or accountability. Classification was initially attempted at the third level of the NFR tree but was unsuccessful, since patterns tended to apply to several NFRs at that level. Table 1 gives the patterns ordered alphabetical and table 2 gives the patterns ordered according to the associated NFR category and Design Level.

Table 1: Catalog of Security Design Patterns (Alphabetical Order)

Pattern Name	**NFR Category**	**Design Level**	**Page**
Authenticator [16]	Confidentiality	Design	30
Authorization [16]	Confidentiality	Design	31
Check Point [22]	Confidentiality	Design	32
Clear Sensitive Information [10]	Confidentiality	Implementation	33
Controlled Object Factory [16]	Integrity	Design	34
Defer to Kernel [10]	Confidentiality	Architectural	35
Distrustful Decomposition [10]	Integrity	Architectural	36
Full View with Errors [22]	Availability	Design	37
Information Obscurity [16]	Confidentiality	Implementation	38
Input Validation [10]	Integrity	Implementation	39
Limited View [22]	Availability	Design	40
Multilevel Security [16]	Confidentiality	Architectural	40
Pathname Canonicalization [10]	Integrity	Implementation	42
Privilege Separation [10]	Integrity	Architectural	43
Resource Acquisition is Initialization (RAII) [10]	Availability	Implementation	44
Role Rights Definition [16]	Confidentiality	Architectural	45
Role-Based Access Control [16]	Confidentiality	Architectural	41
Roles [22]	Confidentiality	Design	45
Secure Access layer [22]	Integrity	Architectural	59
Secure Builder Factory [10]	Integrity	Design	46
Secure Chain of Responsibility [10]	Integrity	Design	48
Secure Channels	Confidentiality	Implementation	60
Secure Directory [10]	Integrity	Implementation	57
Secure Factory [10]	Integrity	Design	49
Secure Logger [10]	Accountability	Implementation	51
Secure Session [22]	Integrity	Design	59
Secure State Machine [10]	Confidentiality	Design	52
Secure Strategy Factory [10]	Integrity	Design	53
Secure Visitor [10]	Integrity	Design	55
Single Access Point [22]	Confidentiality	Design	57

Table 2: Catalog of Security Design Patterns (NFR/Level Order)

Pattern Name	**NFR Category**	**Design Level**	**Page**
Secure Logger [10]	Accountability	Implementation	51
Full View with Errors [22]	Availability	Design	37
Limited View [22]	Availability	Design	40
Resource Acquisition is Initialization (RAII) [10]	Availability	Implementation	44
Defer to Kernel [10]	Confidentiality	Architectural	35
Multilevel Security [16]	Confidentiality	Architectural	40
Role-Based Access Control [16]	Confidentiality	Architectural	41
Role Rights Definition [16]	Confidentiality	Architectural	45
Authenticator [16]	Confidentiality	Design	30
Authorization [16]	Confidentiality	Design	31
Check Point [22]	Confidentiality	Design	32
Roles [22]	Confidentiality	Design	45
Secure State Machine [10]	Confidentiality	Design	52
Single Access Point [22]	Confidentiality	Design	57
Clear Sensitive Information [10]	Confidentiality	Implementation	33
Information Obscurity [16]	Confidentiality	Implementation	38
Secure Channels	Confidentiality	Implementation	60
Distrustful Decomposition [10]	Integrity	Architectural	36
Privilege Separation [10]	Integrity	Architectural	43
Secure Access layer [22]	Integrity	Architectural	59
Controlled Object Factory [16]	Integrity	Design	34
Secure Builder Factory [10]	Integrity	Design	46
Secure Chain of Responsibility [10]	Integrity	Design	48
Secure Factory [10]	Integrity	Design	49
Secure Session [22]	Integrity	Design	59
Secure Strategy Factory [10]	Integrity	Design	53
Secure Visitor [10]	Integrity	Design	55
Input Validation	Integrity	Implementation	39
Pathname Canonicalization [10]	Integrity	Implementation	42
Secure Directory [10]	Integrity	Implementation	57

Authenticator

The Authenticator pattern verifies that a subject is who that subject claims to be. This pattern supports the use of different authentication algorithms to accommodate different users. This pattern keeps authentication information separated to increase security but at the cost of complexity. [16]

Authenticator can handle many types of users that require different authentication algorithms while storing sensitive information in a secure area. Algorithm complexity can be varied to accommodate the use of protocols based on combinations of what a user knows; what a user has; what a user is (i.e., biometrics), or where a user is. As a rule, adding authentication protocols will increase execution time and complexity.

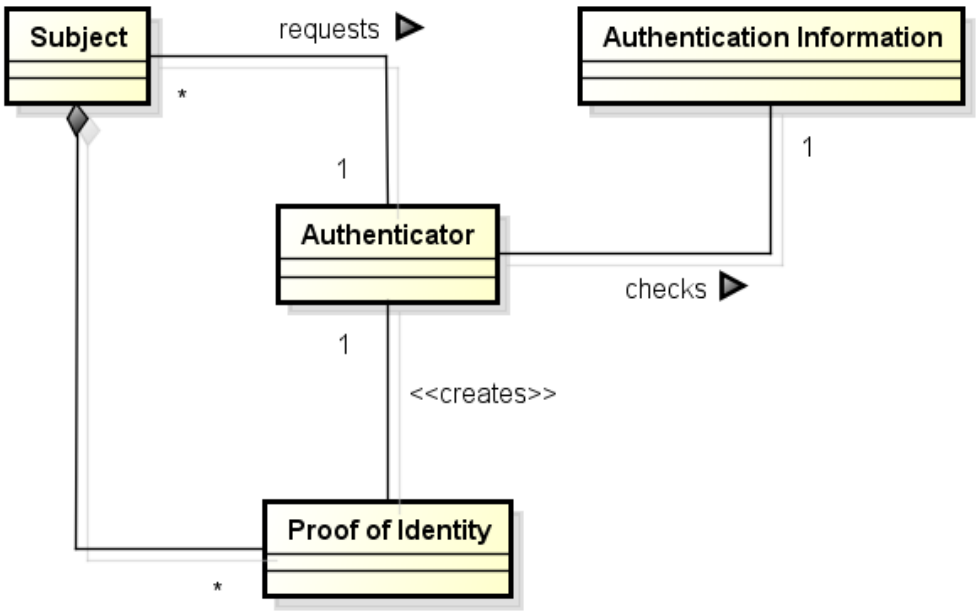


Figure 6: Authenticator Pattern

Figure 6 illustrates the Authenticator pattern with four participants: the Subject, Proof of Identity, Authenticator, and Authentication Information. The Subject is the entity that

needs to be authenticated before given access to the requested resources. Proof of Identity is the object or token given to the Subject once it has been authenticated. The Authenticator is the object encapsulating the algorithm that uses the Authentication Information to authenticate the subject and the creator of the Proof of Identity. [16]

Authorization

The Authorization pattern identifies the subjects that may access a given resource, along with a subject's access privileges for that resource. [16] This pattern separates permissions from the subject and resources. It is flexible enough to handle a variety of subjects, resources, and privileges. Security policies should define privilege sets for subjects that will be encapsulated in authorization rules. These authorization rules will be associated with a subject and resource.

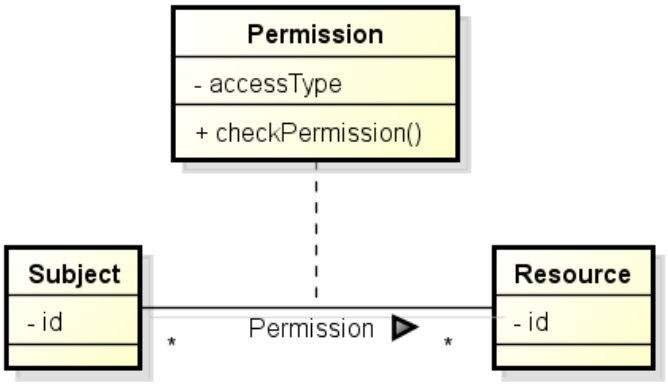


Figure 7: Authorization Pattern

Figure 7 illustrates the Authorization pattern with three participants: the Subject, Permission, and Resource. The Subject is the entity that needs to access the resource. Each Permission represents one way in which the Subject can access the Resource and provides mechanisms to check access. [16]

Check Point

The Check Point pattern prevents users from getting access to confidential information and protects against malicious acts against a system's data. [22] Check points are entities that encapsulate an enterprise's security policy. The Check Point pattern can be implemented using the strategy pattern described in Gamma et al. [11] This use of the strategy pattern allows different check points to be applied at different points of a system's operation, allowing security measures to vary by client. This is especially useful for developers who can stub algorithms for security policies before they are known. An example of a specific check point is a user logging into a system. During development, this check point can be stubbed to aid programmers in testing. [22]

The Check Point pattern can be varied based on how a user logs into a system, how many failed login attempts occurred, or what time of the day it is. This pattern separates complex security checks into a single class to make it easier to modify and allows different strategies to be used based on the application state. It can be used to authenticate or authorize a user. Check points can be used to set up global information such as the current user's id. [22]

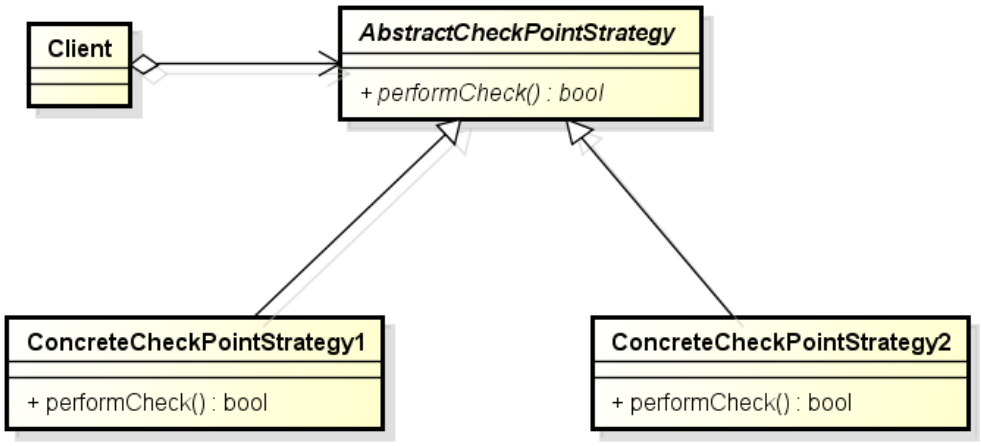


Figure 8: Check Point Pattern

Figure 8 illustrates the Check Point pattern where the primary participant is the Check Point. Check Point encapsulates an enterprise's security policy for use at any time during a program's run-time. This pattern separates security logic from regular application logic, allowing security checks to be interchangeable and reusable. [22]

Clear Sensitive Information

The Clear Sensitive Information pattern prevents access to sensitive information through a reusable resource that should have been cleared. [10] This pattern should be used if a system's processes have access to sensitive data through resources like memory, caches, and disks. When a resource is released, it may still have information that can be accessed by unauthorized users.

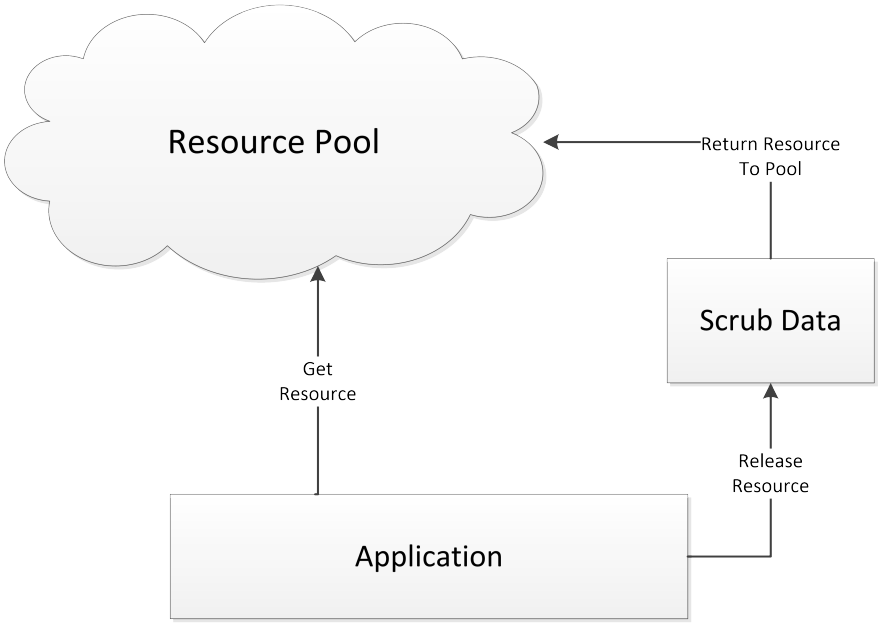


Figure 9: Clear Sensitive Information Pattern

Figure 9 illustrates the Clear Sensitive Information pattern with four participants: Resource, Application, Resource Pool, and Scrub Data. The Resource represents the reusable resource that the Application uses. The Application passes sensitive information to the Re-

source. The Scrub Data clears the sensitive information from the Resource before returning it to the Resource Pool. [10]

Controlled Object Factory

The Controlled Object Factory assists in managing process permissions for objects by determining and setting permissions at the time of object creation. [16] Object creation is facilitated using the Factory or Factory Method pattern described in [11]. If permissions are invariant for each object, the least privilege principle cannot be enforced. The lack of dynamic permissions can leave objects vulnerable to misuse. This pattern defines a list of subjects and privileges when objects are created.

This pattern should be used when security policies describe who can access objects. One method for implementing this pattern includes the association of access control lists (ACL) with objects. In order for systems to be flexible, they must allow dynamic changes to an object's permissions. When using this pattern, all objects must have permissions defined since there will be no default permissions. Objects retrieved from resource pools have permissions set dynamically. Issues with this pattern include creation overhead and unclear initial permissions.

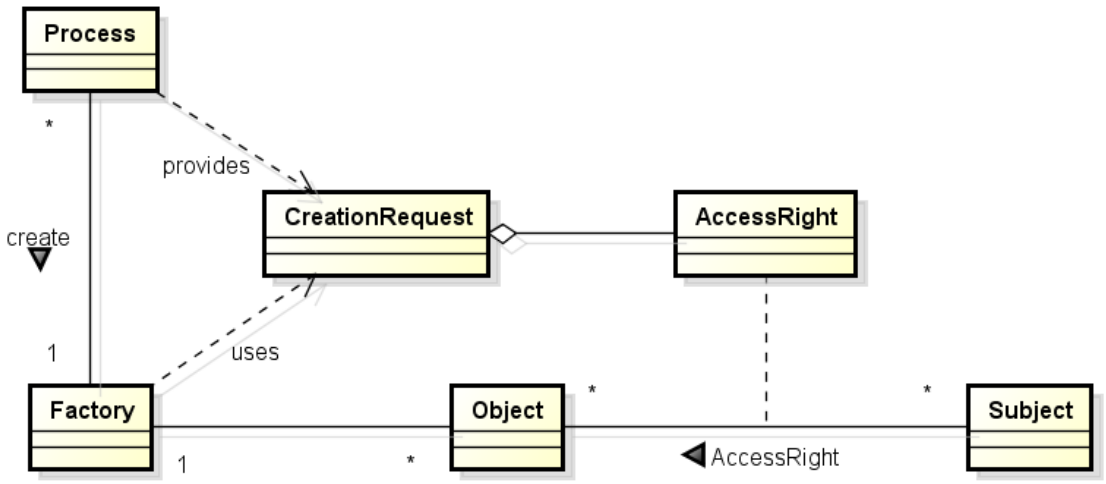


Figure 10: Controlled Object Factory Pattern

Figure 10 illustrates the Controlled Object Factory with six participants: the Process, Factory, CreationRequest, Object, Subject, and Access Right. The Process requests that the Object be created. The Factory creates the Object using the Creation Request. The Creation Request is sent by the Process and contains the Access Right between the Object and Subject. The Subject represents other objects that may want to use the Object. The Controlled Object Factory ensures that objects have the appropriate privileges set for all subjects that will be using it. [16]

Defer to Kernel

The Defer to Kernel pattern separates functionality that requires elevated privileges from unprivileged functionality. A kernel is a major component of an operating system that manages the system's key resources in ways that protect those resources from inappropriate patterns of use. Defer to Kernel uses kernel-level security mechanisms in lieu of user-developed functionality, thereby reducing time for application development and testing. [10]

This pattern can be used when a system can run an application using a user's privileges:

e.g., on UNIX-like systems, which assign each running application a user identifier (UID). This pattern should not be used if the system lacks this ability or if the application does not require elevated privilege. [10]

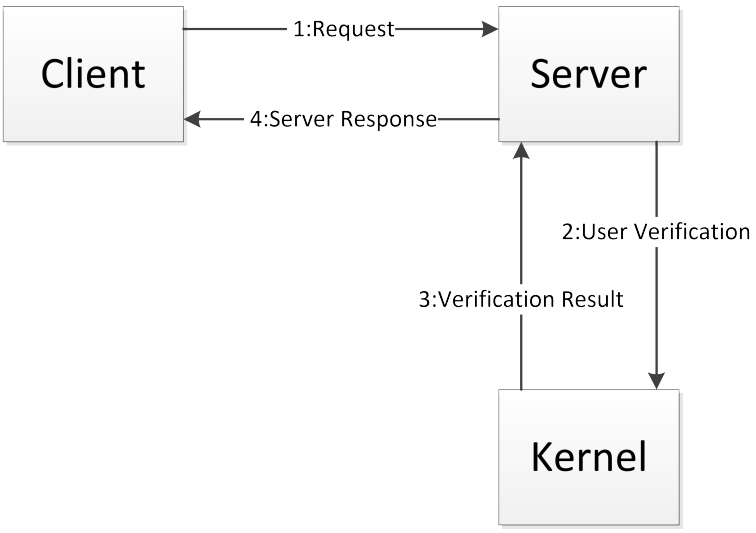


Figure 11: Defer to Kernel Pattern

The Defer to Kernel pattern, illustrated in figure 11, can be implemented using a basic client-server architecture where the server runs at elevated privileges. The pattern's participants include Client, Server, and Kernel. The Client, which runs at the user's privilege level, sends jobs that require elevated privileges to the Server. The Server handles requests from the Client while using the Kernel to determine if the user has the appropriate permissions. The Kernel provides verification for the user and provides communication mechanisms for the client and server. [10]

Distrustful Decomposition

The Distrustful Decomposition pattern separates functionality into mutually untrusting parts to reduce the attack surface of an individual part and to reduce data exposed if the part is compromised. This pattern should be used to defend against attacks on vulnerable

applications that run with elevated permissions. Examples include attacks against instances of Internet Explorer that run with administrator privileges and buffer overflow attacks that attack telnet daemons that run as root to execute unsafe code. This pattern is useful if a system performs several high-level functions that require varying privilege levels. [10]

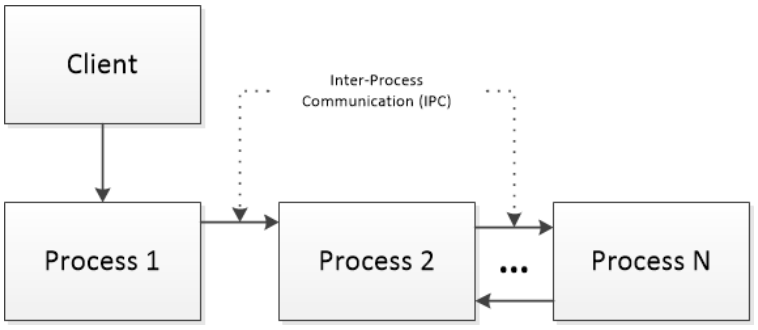


Figure 12: Distrustful Decomposition Pattern

The Distrustful Decomposition pattern, illustrated in figure 12, breaks an application into independent programs that run as separate processes. Each process manages a subset of the original application's functionality. [10] The division proper typically minimizes the amount of code that needs to run at elevated levels of privilege while restricting the privileges that any one process gets on a "need to have" basis. These processes communicate using some inter-process communication (IPC) protocol. The pattern's participants include Processes , a Client, and some IPC. Processes represent the partitioned program . The Client will send requests to the appropriate Process. The IPC is used as a communication mechanism for the processes. [10]

Full View with Errors

The Full View with Errors pattern allows developers to ignore permissions when developing a user interface. [22] This pattern allows every user to see each of a program's options, including options that that user is not allowed to invoke. Once a user selects the desired op-

tion, the system determines if the user may use that operation and respond accordingly. This architectural pattern prevents users from performing illegal operations and is very useful for users with almost all privileges.

This pattern should be built with an error handling framework that can respond to errors that also has a logging module. The use of this pattern allows training materials to be consistent for all types of users. This pattern is easy to implement since one view can be presented to all users. It may, however, confuse users by bombarding them with error messages. Other patterns that work well with this pattern include Checks for performing security checks on the operations and Roles for determining if a user has permission to perform the desired operation. [22]

Information Obscurity

The Information Obscurity pattern encrypts sensitive data in insecure environments with the goal of protecting the data from theft. [16] This pattern should be used when data is frequently passed between internal and/or external systems. Security policies should define the level of sensitivity of data in order to determine what data to encrypt and how aggressively to encrypt that data. Since encryption and decryption can be time-consuming, it is best to limit their use where possible. This pattern addresses that issue by using the level of sensitivity to determine what needs to be obscured through encryption.

This pattern requires key storage mechanisms, encryption mechanisms, and a secure location to store encryption keys and algorithms. [16] When using this pattern, data should be categorized to determine its sensitivity. This categorization should be used to determine what content needs to encrypted. Benefits for using the pattern include improved security and decreased performance impact, since information obscurity is selective instead of all inclusive. [16]

Input Validation

The Input Validation pattern ensures that data input by the subject is valid and free from malicious text. [10] This pattern should be used when a subject's input cannot be trusted. Failure to validate user input could expose systems to attacks like SQL injection and overflow attacks. To implement Input Validation, developers must identify what input is untrusted and validate this input before using it. While input should be validated at a trusted server, it can be also done client-side to decrease communication. [10]

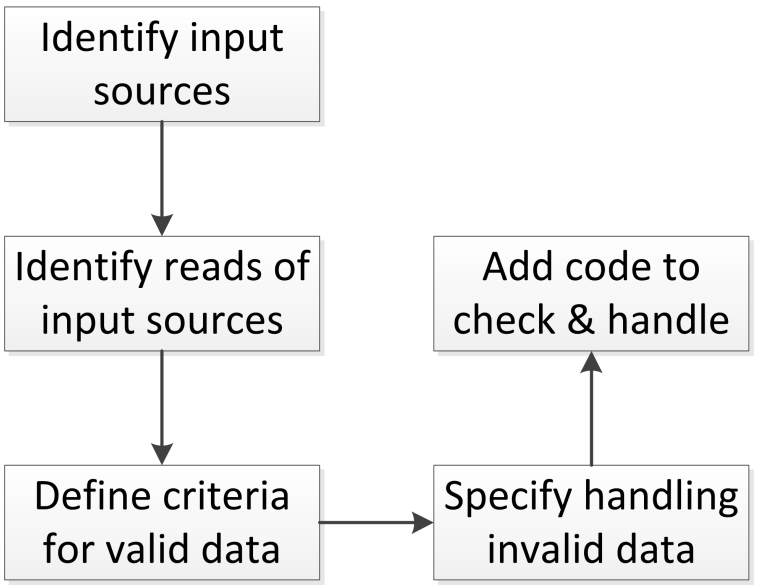


Figure 13: Input Validation Process

The process, illustrated in figure 13, to implement Input Validation has five stages. First, identify all input sources, including resources from which the system will get data. [10] Second, identify all locations within the code from which the input sources are read. Third, define validation rules for each location from the previous step. Fourth, specify how invalid data is handled. Fifth, add the validation code and invalid data handling code to each location identified in step two. Gervasio [12] gives a realization using the Strategy pattern by Gamma et al. [10]

Limited View

The Limited View pattern enables only functions that the current user is privileged to access. [22] This pattern should be used when most users have permissions to a limited number of operations. When using this pattern, security checks are performed before building the view. Limited View uses the current session's user's permissions and builds a user interface tailored to the current user. Limited View can be implemented using composites or builders to generate the user interface or by using a state machine to represent different views.[22]

Using Limited View results in a cleaner design with fewer security checks and a less cluttered user interface. [22] While minimizing error messages for invalid operations, users may be confused by the presence of unexpected options. This pattern also requires that training material be customized for each type of user. Other patterns that may be used by Limited View include Session to track the current user's permissions and Roles for configuring views. The builder and strategy can be used to manage different views. [22]

Multilevel Security

The Multilevel Security pattern determines access permissions for data objects by partitioning users and data into categories, based on patterns of acceptable use. [16] It uses trusted processes to change these classifications at need. Classifications for users are called clearances, while classifications for data are called sensitivity levels.

The use of classifications for users and data protects the system's confidentiality and integrity. This pattern's use should be limited to organizations where permissions are based on rank or position within an organization: e.g., in the United States military. [16]

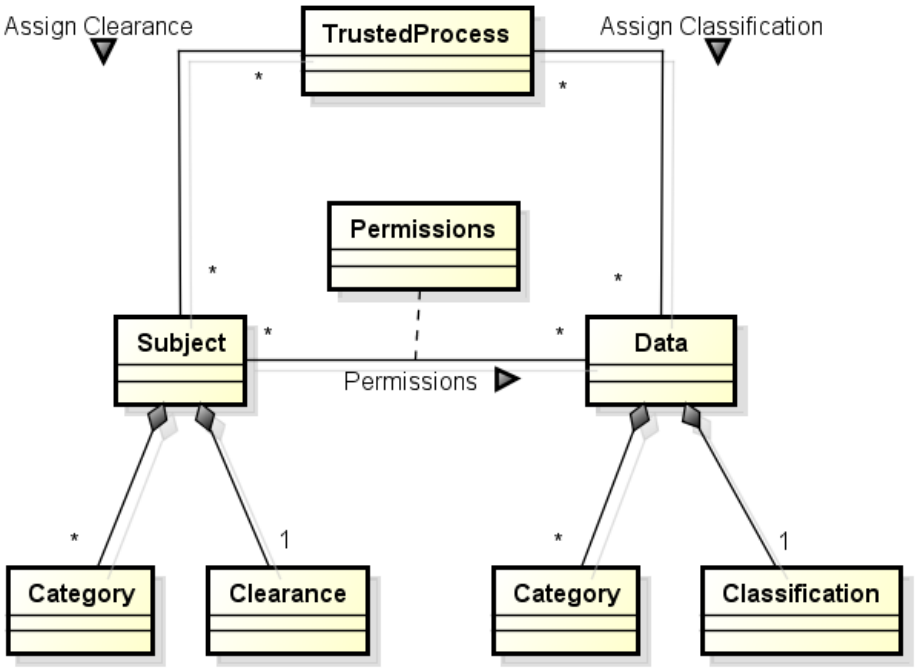


Figure 14: Multilevel Security Pattern

Figure 14 illustrates the Multilevel Security pattern with seven participants: Subject, Data, TrustedProcess, Category, Clearance, Category, and Classification. The Subject contains the Category for the organizational unit to which it belongs and the Clearance. Clearance represents the Subject's clearance level; it is used to determine access permissions. The Data also contains the Category for the organizational unit to which it belongs and the Classification. Classification represents the Data's sensitivity level; it is used to determine the access permissions. The TrustedProcess is the entity that can change the Clearance level for the Subject and Classification level for the Data. [16]

Role-Based Access Control

The Role-Based Access Control pattern associates permissions with users based on their system-assigned roles. [16] Associating user with roles and roles with permissions eliminates

the work of associating individual users with individual sets of permissions. This pattern is appropriate when a large number of users or a large number of resources share related access privileges.

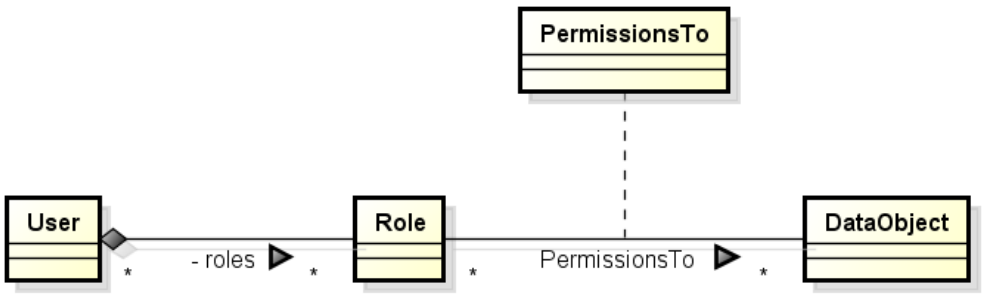


Figure 15: Role-Based Access Control Pattern

Figure 15 illustrates the Role-Based Access Control pattern with four participants: User, Role, PermissionsTo, and DataObject. User is the class that represents the system's current user. [16] Role represents the current role the user has assumed. DataObject represents the data the user wishes to access. An association class, PermissionsTo, determines the role's permissions for the DataObject. Note that the User has a many-to-many relationship to the Role, and Role has a many-to-many relationship with DataObject. [16]

This pattern reduces the workload for managing users and simpler security, since there are many more users than roles. This pattern also allows users to switch between roles, making the application easier to use. [16]

Pathname Canonicalization

The Pathname Canonicalization pattern ensures that the path to given files and directories is valid and free of links to other locations.[10] This pattern should be used when an application accepts pathnames from its users. This pattern protects against directory traversal vulnerabilities. To implement this pattern, the application must use the OS spe-

cific pathname canonicalization function to retrieve the canonicalized pathname from a given path. This canonicalized pathname is then used to access file system resources. This ensures that the resource is not a link. [10]

This pattern has two participants: Application and File System. The Application processes a request from the user to open a file system resource. The Application uses the File System to determine if the given file path is secure. This pattern prevents the user from accessing file system resources outside of the appropriate environment. [10]

Privilege Separation

The Privilege Separation pattern limits the amount of code that runs at an elevated privilege without affecting program functionality. [10] This pattern is a specialization of the Distrustful Decomposition pattern. This pattern should be used when there is a small subset of functionality that requires elevated privileges. This pattern is useful if the application has a large attack surface and system functionality requires user to authentication.

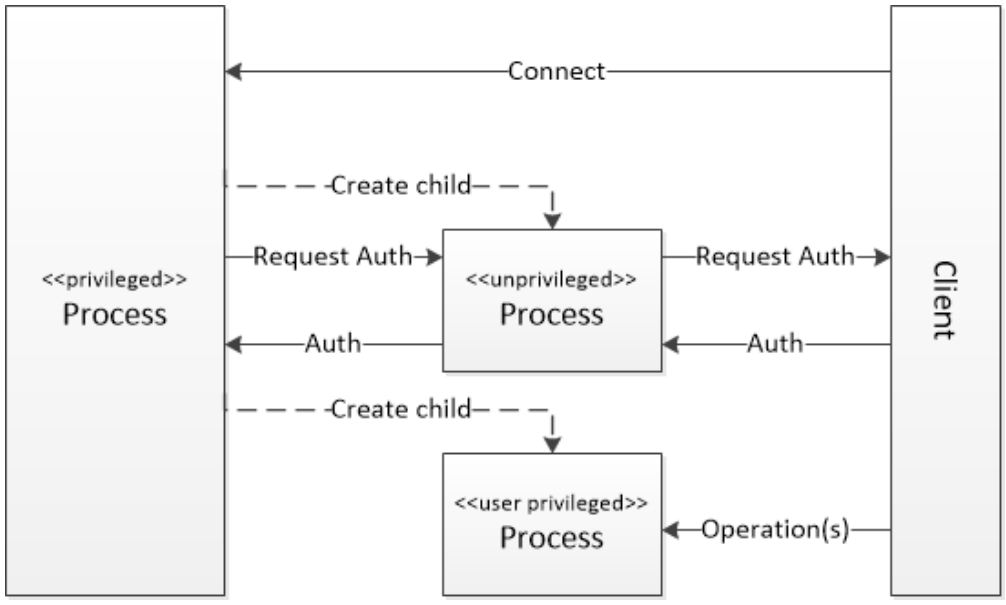


Figure 16: Privilege Separation Pattern Example

The example given in figure 16 has four participants: Privileged Process, Unprivileged Process, Client, and User Privileged Process. [10] The Privileged Process handles initial requests from the Client. A client can be a user or another process. Once a request is received, an Unprivileged Process is created to handle authentication with the client. After authentication is completed, the Privileged Process creates a User Privileged Process. This User Privileged Process handles all further requests from the client. If an attacker gains control of the User Privileged Process, it will be confined within the limitations of the process. This protects the system's Privileged Process from attackers. By separating responsibility into different processes, code is easier to review and test. [10]

To generalize the implementation of Privilege Separation, create a server with elevated privileges to handle initial user requests. Once a request has been received, spawn an unprivileged process to handle authentication. Finally, once authentication has completed, spawn another process with limited privileges based on the client's credentials to handle the clients requests. [10]

Resource Acquisition Is Initialization

The Resource Acquisition Is Initialization (RAII) pattern ensures that an object's resources are properly allocated and deallocated by using the object's constructor and destructor instead of relying on the client. [10] This pattern should be used when resource usage could be intensive enough to trigger memory reallocation. This can expose data that should have been cleared before reuse.

Implementing RAII ensures that the resources needed by an object are allocated in the constructor and deallocated in the object's destructor, instead of relying on the client to manage them. Alternatives to this approach include the use of garbage collection or dependency injection paradigm. [10]

Role Rights Definition

The Role Rights Definition pattern upholds the principle of "least privilege" by basing the assignment of permissions on use cases. [16] This pattern should be used when roles correspond to functional tasks, permissions needs to be assigned using the "least privilege" principle, roles change often, and/or permissions should be independent of implementation. Sequence diagrams are used to determine what operations a role can perform in a system. These operations are then used to determine that role's permissions. This is performed for all actors of a use case diagram. [16]

Role Rights Definition is typically used with other role based permission patterns such as Role Based Access Control (RBAC). This pattern requires the development of use cases and the analysis of sequence diagrams to determine minimal permissions for roles relative to their responsibilities. [16]

Roles

The Roles pattern consolidates security policies that apply to multiple users into a single entity. [22] Associating privileges with roles instead of individual users makes it easier to manage groups of users with common privileges. A strategy for implementing Roles is to allow a user to have multiple roles and a role to have multiple privileges. This implementation allows each role to represent a stakeholder and to capture the stakeholder's appropriate privileges. Role-based inheritance relationships can also be used to form a hierarchy of roles.

One consequence of the Roles pattern is that administrators manage user-role and role-privilege relationships instead of user-privilege relationships. Associating multiple users to a role is simpler than associating users with individualized sets of privileges. Other consequences include the system having a good way to group privileges and it being more convenient to administer. These benefits come at the cost of adding an extra layer of com-

plexity to the system. One well known application of Roles is UNIX's owner-group-other role system. Roles are also known as Actors, Groups and Profiles. [22]

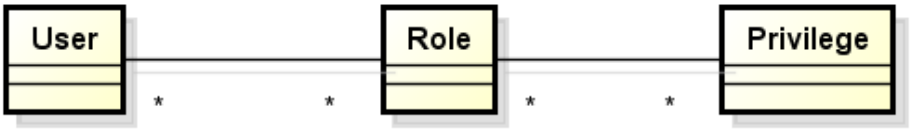


Figure 17: Roles Pattern

Figure 17 illustrates the Roles pattern with three participants: User, Role, and Privilege. A User is associated with a Role. Once the user is associated with a Role, that user gains Privileges associated with the Role. [22]

Secure Builder Factory

The Secure Builder Factory pattern separates security logic from the creation of complex objects: objects that are constructed from several other objects. [10] This pattern can be used when security credentials dictate a complex object's composition. This pattern applies if the system uses the Builder pattern given by [11] and the Builder is dependent on the user security privileges.

The Secure Builder Factory pattern provides an easy way to select a strategy object based on the security credentials. This pattern is extended from Secure Factory Pattern and uses the strategy pattern given in [11]. This pattern should be used when behavior needs to vary based on the user logged into the system and can only be determined by the user's credentials.

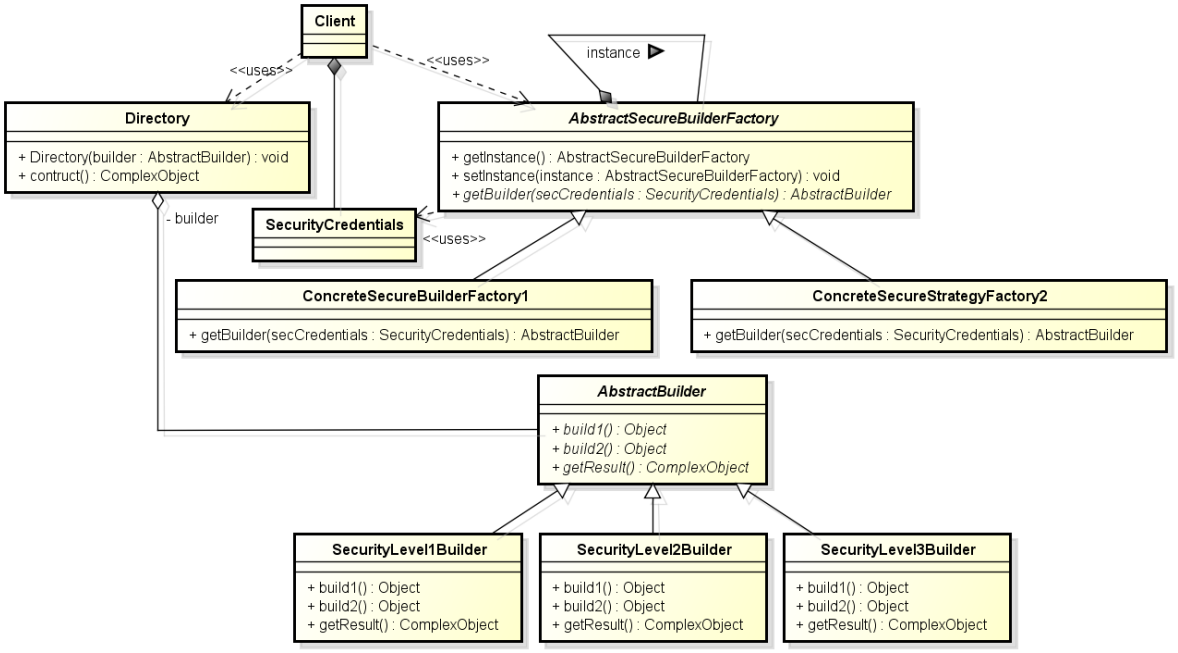


Figure 18: Secure Builder Factory Pattern

Figure 18 illustrates the Secure Builder Factory pattern with seven participants: Abstract Secure Builder Factory, Concrete Secure Builder Factory, Abstract Builder, Security Level Builder, Director, Security Credentials and Client. [22] Abstract Secure Builder Factory is the primary participant: it provides an instance of the builder factory, provides a function to change the builder factory instance at run-time, and defines the interface for retrieving an object to be implemented in the concrete secure builder factory. Concrete Builder Factory represents the concrete implementations of the abstract factory: it implements the getBuilder method. Abstract Builder defines the required methods for all Security Level Builder classes. Security Level Builder implements the required algorithms for the user with the appropriate security level. [22] The Director is used by the Client to build and return complex objects using the Security Level Builder. Directors are better described in [11]. The Client tracks a user's security credentials: it uses the getInstance method to retrieve a concrete instance of the builder and then gets a builder for the given security credentials using the concrete

factory's getBuilder method. Security Credentials represents the current's user credentials.

The Secure Builder Factory pattern separates and hides security logic from the Client resulting in more concise code that is easier to test. The Secure Builder Factory acts as a black box that allows security logic to change independently of the Client's behavior. [10]

Secure Chain of Responsibility

The Secure Chain of Responsibility pattern separates security checks from their associated actions. [10] This pattern can be used when a handler in a chain of responsibility uses security credentials to determine if that handler should run. An example given by [10] is the implementation of a role-based access system where each role is represented as a handler in the chain. When a request is passed down the chain, the user's credentials are passed down with it.

Secure Chain of Responsibility is almost identical to the Chain of Responsibility given by [11], except that a method to check credentials is implemented by each handler. This method is used in the handleRequest method before handling the request or passing the request to the next handler.

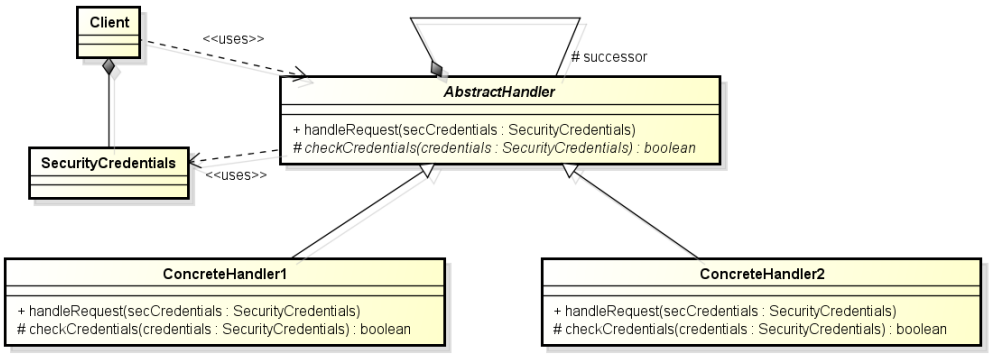


Figure 19: Secure Chain of Responsibility Pattern

Figure 19 illustrates the Secure Chain of Responsibility pattern with four participants:

Abstract Handler, Concrete Handler, Security Credentials, and Client. Abstract Handler defines the interface that the Concrete Handlers must implement. Concrete Handlers invoke requests that they are designed to manage, conditional upon a request's passing the security check. This check differentiates this pattern different than the one given by [11]. The Client tracks a user's security credentials and gives the request to the chain's first handler. Security Credentials represents the current's user credentials. [10]

The Secure Chain of Responsibility pattern separates the security logic into a method within the handler making it easier to modify and test. The chain acts as a black box that allows the system to change how it responds to requests independently of the code that makes the request. [10]

Secure Factory

The Secure Factory pattern separates the security logic for selecting an object from the creation of that object. [10] The pattern is an extension of the Abstract Factory pattern given by [11]. Abstract Factory is used for its ability to transparently change the Concrete Factory during run-time. The primary motivation for using this pattern is to make it easier to test and verify the logic used to select or create the object. This also makes it easier to modify the logic for selecting an object based on security requirements.

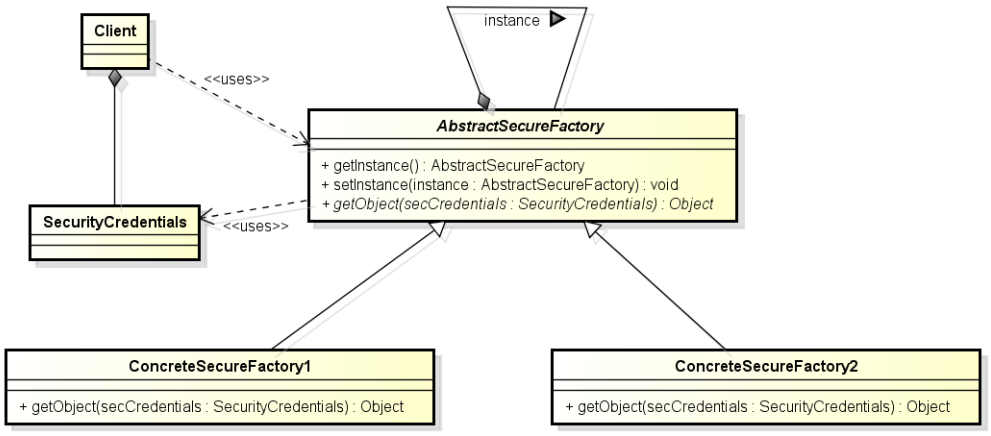


Figure 20: Secure Factory Pattern

Figure 20 illustrates the Security Factory pattern with four participants: Abstract Secure Factory, Concrete Secure Factory, Security Credentials, and Client. Abstract Secure Factory is the primary participant: it provides an instance of the Secure Factory, provides a function to change the Secure Factory instance at run-time, and defines the interface for retrieving an object to be implemented in the Concrete Secure Factory classes. Concrete Secure Factory represents the concrete implementations of the abstract factory that implements the getObject method. The Client class tracks a user's security credentials: it uses the getInstance method to retrieve a concrete instance of the factory and then gets an object for the given security credentials using the concrete factory's getObject. Security Credentials represents the current's user credentials. [10]

The Secure Factory pattern separates and hides security logic from the Client resulting in more concise code that is easier to test. The Secure Factory acts as a black box that allows security logic to change without having to change the behavior of the Client. [10]

Secure Logger

The Secure Logger pattern protects a system's logs from access by potential attackers and prevents attackers from altering logs to hide their activity. [10] This pattern should be used when the system stores logging information externally and when these logs contain sensitive information.

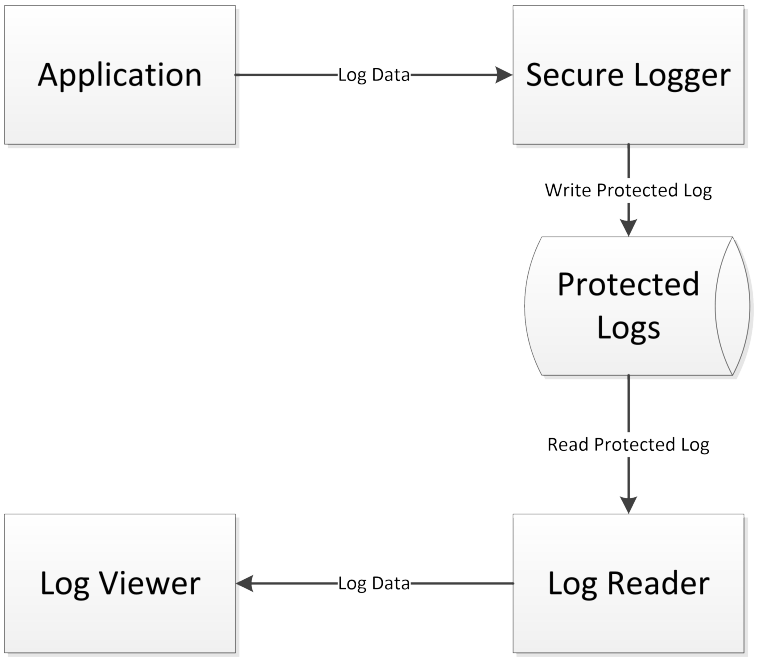


Figure 21: Secure Logger Process

Figure 21 illustrates the Secure Logger pattern with four participants: Application, Secure Logger, Log Reader, and Log Viewer. The Application creates the logs that will be handled by the Secure Logger. Secure Logger encrypts the logs to make it difficult for attackers to access the logs. Log Reader decrypts the logs for the Log Viewer, which limits log access to authorized users. Log Reader can be a part of the Secure Logger system. The Secure Logger pattern ensures that logs acquired by attackers are nearly impossible to decipher and that modification to the logs are detectable. [10]

Secure State Machine

The Secure State Machine pattern provides a "clear separation between security mechanisms and user-level functionality by implementing the security and user-level functionality in two different state machines". [10] By separating security and user-level functionality, designers increase the design's cohesion, making it easy to test, review, and verify security properties. This decreases the likelihood of introducing vulnerabilities into the system. The pattern also decreases coupling between the security and user-level functionality, making future modifications to the system easier.

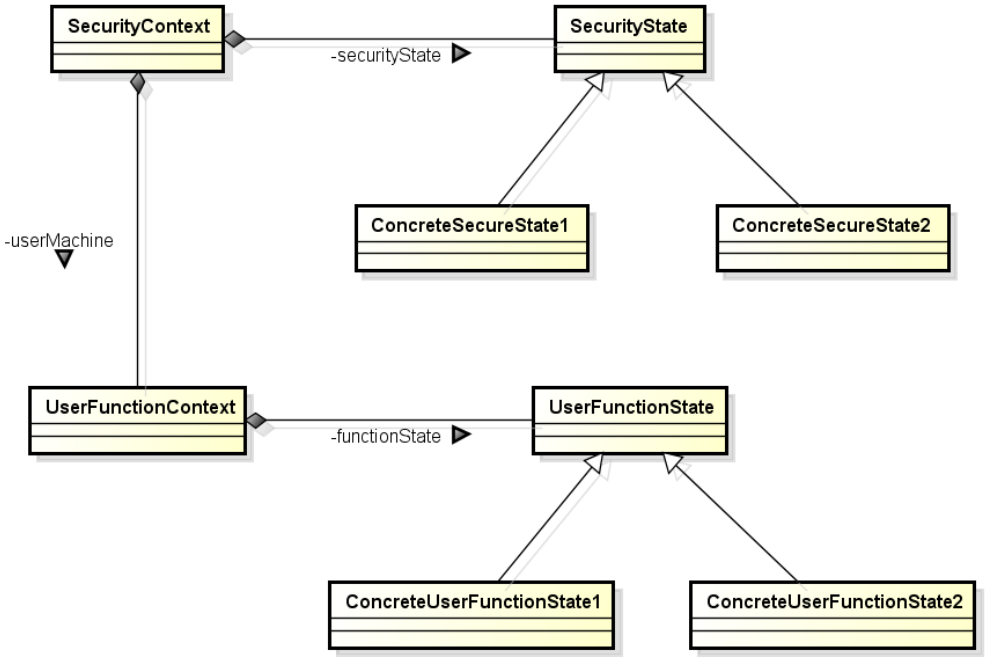


Figure 22: Secure State Machine Pattern

Figure 22 illustrates the Secure State Machine pattern with four participants: Secure Context, Secure State, User Function Context, and User Function State. [10] Secure Context, the primary participant, provides all operations to the client and acts as a proxy to the User Function Context. Secure State is an abstract class that must define all functionality handled

by the secure state machine. Secure State will be subclassed by several concrete classes that represent how the security state changes based on the system's current user. User Function Context is a class whose functionality must match the Secure State class so that requests can be forwarded to the User Function State when security requirements are met. User Function Context can only be created by Security Context. User Function State is an abstract class that must have the same interface as the Secure State. The Secure State Machine pattern "separate[s] security mechanics from user-level functionality" and "prevents programmatic access to the user-level functionality that avoids security" checks. [10]

Secure Strategy Factory

The Secure Strategy Factory pattern provides an easy way to select a strategy object based on the user's security credentials. [10] This pattern combines Secure Factory with the Strategy pattern as given in [11]. This pattern should be used when behavior needs to vary based on the user logged into the system and can only be determined by the user's credentials.

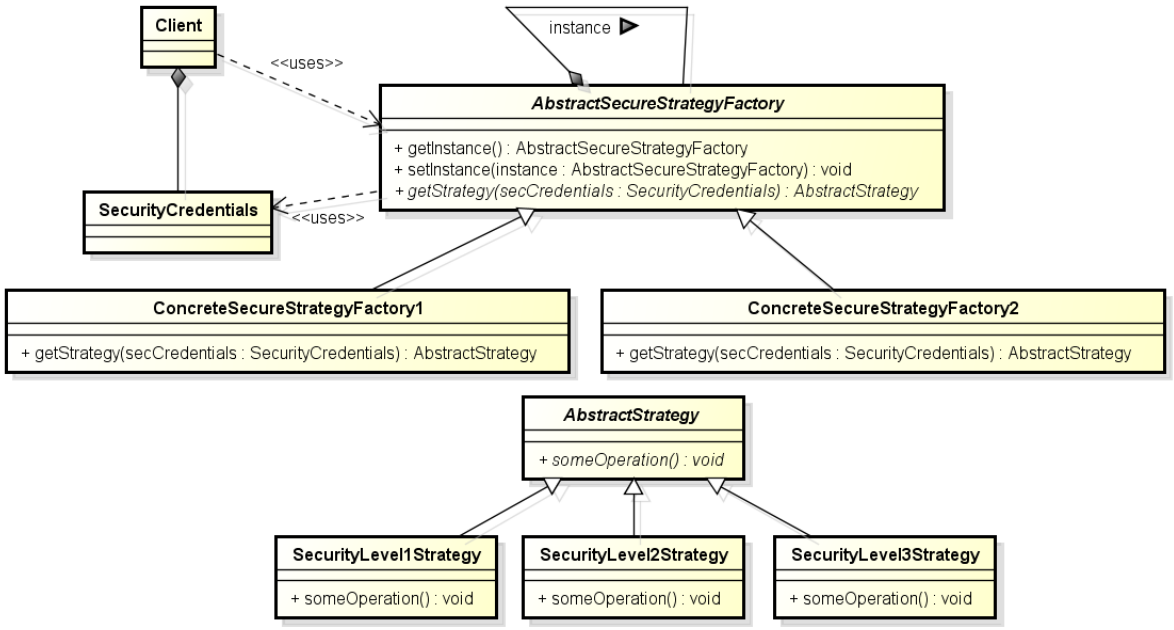


Figure 23: Secure Strategy Factory Pattern

Figure 23 illustrates the Secure Strategy Factory pattern with six participants: Abstract Secure Strategy Factory, Concrete Secure Strategy Factory, Abstract Strategy, Security Level Strategy, Security Credentials and Client. [10] Abstract Secure Strategy Factory is the primary participant: it provides an instance of the strategy factory, provides a function to change the security factory instance at run-time, and defines the interface for retrieving an object to be implemented in the concrete secure strategy factory. Concrete Secure Factory represents the concrete implementations of the Abstract Factory that implement the getStrategy method. Abstract Strategy defines the required methods for all Security Level Strategy classes. Security Level Strategy implements the required algorithms for the user with the appropriate security level. The Client tracks the user's security credentials. The Client uses the getInstance method to retrieve a concrete instance of the factory and then gets a strategy for the given security credentials using the concrete factory's getStrategy method. Security Credentials represents the current's user credentials. [10]

The Secure Strategy Factory pattern separates and hides security logic from the Client, resulting in more concise code that is easier to test. The Secure Strategy Factory acts as a black box that allows security logic to change independently of the Client's behavior. [10]

Secure Visitor

The Secure Visitor pattern gives designers the tools to deny access to a node if a user lacks the appropriate privileges. [10] When using the Visitor pattern given by [11], a hierarchy is used to structure data into different nodes. This means the node is locked to all visitors except those who have the proper security credentials. This results in a clear separation of security logic from user functionality.

As with the Visitor pattern given by [11], Secure Visitor should be used with a system that has its data organized hierarchically. This pattern also places different access restrictions on each node.

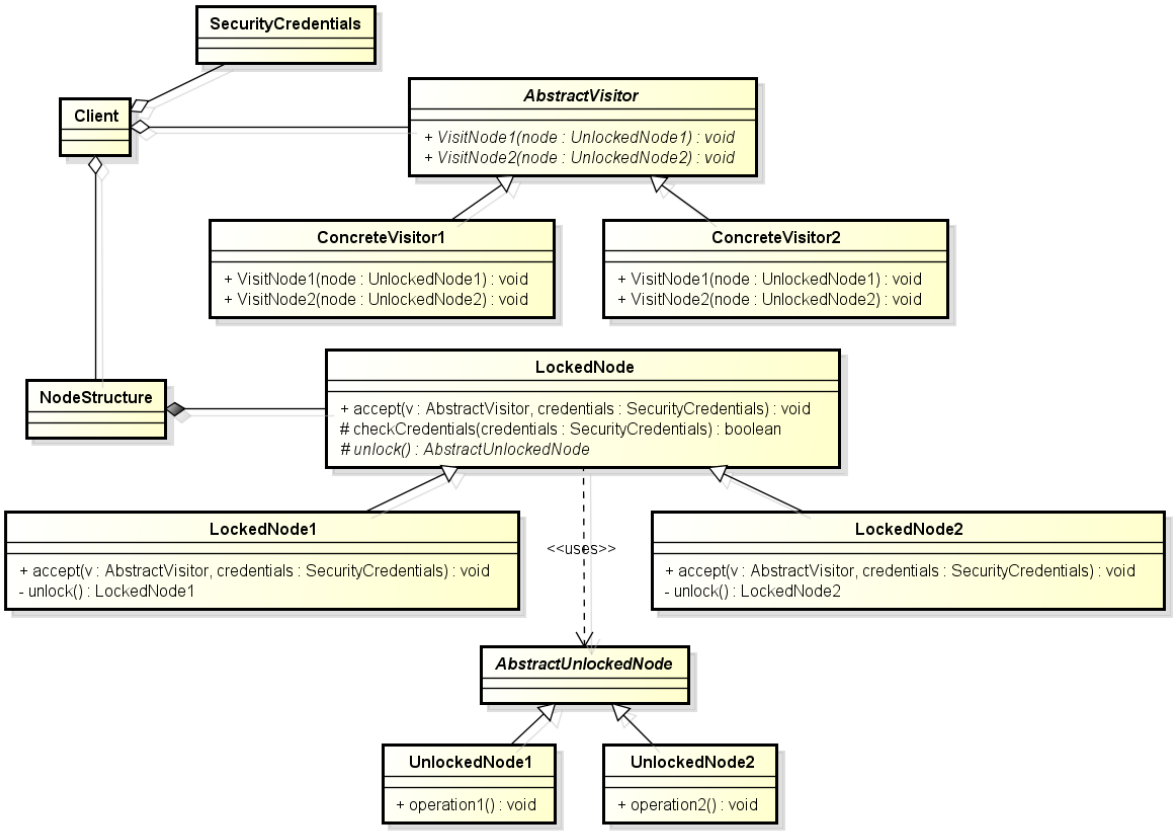


Figure 24: Secure Visitor Pattern

Figure 24 illustrates the Secure Visitor pattern with nine participants: Abstract Visitor, Concrete Visitor, Base Locked Node, Locked Node, Abstract Unlocked Node, Unlocked Node, Node Structure, Security Credentials, and Client. [10] The Abstract Visitor is exactly the same as [11]'s Visitor pattern except that its visit method takes an Unlocked Node. Concrete Visitor implements the methods defined in Abstract Visitor. Base Locked Node defines an accept method that takes a Visitor and Security Credentials. This participant also provides a method to check the user's credentials and defines an abstract method for unlocking the node. The Locked Node implements the accept and unlock methods, which test the user's credentials: if the credentials are correct the Locked Node will provide the appropriate Unlocked Node to the Visitor's visitNode method. The Abstract Unlocked Node

represents the functionality for the Locked Node. The Unlocked Node is the implementation of the Abstract Unlocked Node and contains the functionality needed by the associated Locked Node that will be unlocked once security check is passed. The Client maintains the user Security Credentials, the instantiated Visitor, and the Node Structure containing a hierarchy of Locked Nodes. The Locked Node acts as a security checkpoint to the actual node functionality that is contained in the Unlocked Node. This pattern clearly separates security logic from the user functionality and prevents access to this functionality without proper security checks. [10]

Secure Directory

The Secure Directory pattern protects files from being manipulated by an attacker. [10] This pattern should be used when an application uses files for an extended time of period. It protects against malicious file modification, including the use of race conditions to obtain unauthorized access to files. Secure Directory limits a program's access to those resources that that program's user may access. This is useful for applications that need to read and/or write to files in an insecure environment.

Secure Directory forces a program to find the canonical pathname for a file's directory and check if the directory is secure. This pattern has two participants: Program and File System. The Program is ensures the directory on the File System can only be written to by the user of the program. [10]

Single Access Point

The Single Access Point pattern restricts access into an application to one entry point. [22] Security requirements are complicated by the need to support multiple external interfaces. This pattern removes the need to validate users at multiple entry points and duplicate

security-related code throughout an application. The pattern's lone entry point must collect all information about a user needed throughout the system. The entry point must also launch all sub-applications and may force the user to enter information that is unnecessary for the user's current session. [22]

This pattern may be used when a system is a composition of other applications that could result in duplicated code for logging in a user. Multiple entry points could need different data requiring the user to provide unnecessary information when gaining access to different parts of the system. This results in a larger attack surface, making the system less secure and a simpler control flow, making the system easier to test. [22]

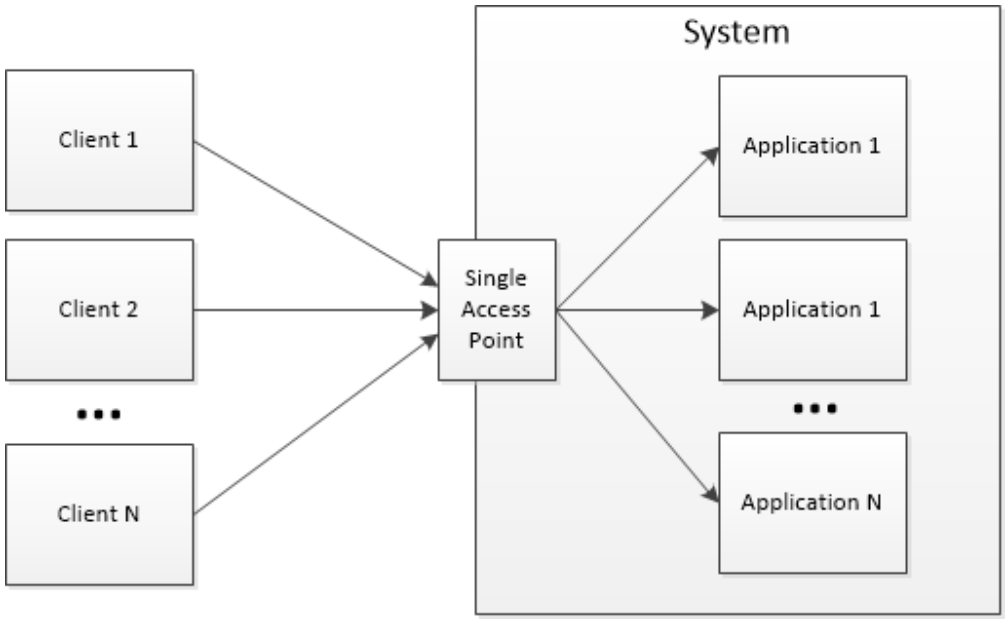


Figure 25: Single Access Point Pattern

Figure 25 illustrates the Single Access Point pattern with four participants: Client, Single Access Point, System, and Application. The Client is the user or process that wants to gain access to the System. This is done by going through the Single Access Point. The System can be comprised of the Application the Client wishes to gain access to. [22]

Secure Session

The Secure Session pattern can be used to store globally relevant information, such as a user's username and roles, for use throughout an application. Session can be implemented as a Singleton pattern [11]. Session is often used in web applications to store a user's information between requests. Since HTTP is stateless, using Session allows a user to interact with an application without having to reenter access credentials.

The Secure Session provides a location to store common data that is made available to all components. It may, unfortunately, become complex if it must store a large amount of information. [22]

Secure Access Layer

The Secure Access Layer pattern provides a secure method for communicating with external systems. [10] System integration often leaves weaknesses at the boundaries between the initial systems' APIs. Implementing security checks on both sides of these boundary points may yield duplicate code, making the integrated system more difficult to maintain and test. This pattern should be used when interfacing with external systems that may not be secure. [10]

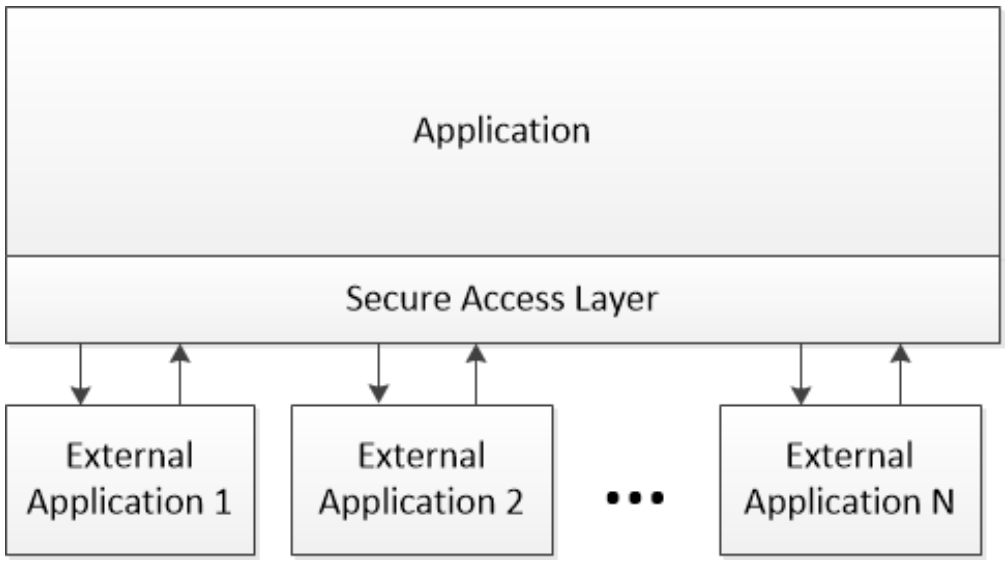


Figure 26: Secure Access Layer Pattern

The Secure Access Layer pattern, illustrated in figure 26 can isolate communications between different applications, making the system easier to maintain and test. By using Secure Access Layer, a system can be ported to a different platform more easily. [10]

Secure Channels

The Secure Channel pattern ensures that connections between clients and servers are secure, especially when communicating over public networks. [16] The Secure Channels pattern encrypts sensitive data while transmitting other data in the clear in order to reduce encryption overhead. This pattern requires the client and server to establish secure connections.

Benefits for using this pattern include improved security and minimal development time, due to the use of existing technology to implement secure communications without impacting the exchange of non-sensitive data. Issues with this pattern include decreased performance from encryption time, scalability of client communication, and increased cost. [16]

CHAPTER 4

CASE STUDIES

Three case studies were undertaken to evaluate the effectiveness of the mapping from security NFRs to security patterns developed in Chapters 2 and 3. The basis for these case studies, the Online Tenure and Promotion System (OLTP), is a system that was developed for managing documents and other artifacts related to tenure and promotion. This system, as described below, has nontrivial requirements that determine when these documents can be uploaded to the system and who is authorized to upload and view them.

Each of these three studies was conducted using the following five-step methodology. First, four security non-functional requirements (NFRs) were selected based on their importance to the OLTP. Second, these NFRs were mapped to security issues in the hierarchy created in Chapter 2. Third, three patterns were selected, based on the related security issues, from the catalogue in Chapter 3. Fourth, a security mechanism implemented in the original system was replaced with a security pattern selected from the catalogue. Finally, the original and new design were compared to determine how well they satisfied the selected security NFRs.

Online Tenure and Promotion System

The Online Tenure and Promotion System (OLTP) is an electronic program that replaces a paper process for application for tenure and promotion. This system was originally developed for East Tennessee State University (ETSU). It provides a platform to manage and document tenure and/or promotion for faculty members who meet the appropriate requirements. Tenure is a status given to faculty that protects them from being terminated without just cause. A faculty member can be promoted through four different levels, starting at the instructor or assistant professor level and progressing through associate professor and full

professor.

The tenure and promotion process takes a full calendar year and has many stages and types of users. The stages follow a timeline that associates different users with different permissions at different stages of the process. The tenure and promotion process involves the management of several types of electronic artifacts: documents uploaded as PDF files, recommendations entered as a yes or no, committee votes, and electronic report signatures for each candidate. High level requirements for the OLTP system state that users should be able to perform tasks required to create the artifacts for their given role(s).

The OLTP's requirements for user-dependent, stage-based functionality created three broad classes of security-related challenges for the system designers. The first involved tracking the various stages of the tenure and promotion process. For this, the idea of a timeline was developed that was used to check the current date against the Tenure and Promotion deadlines. The second involved determining a user's permissions: a concern that was met by associating each user with a role. The third involved tracking the completeness of the artifacts related to individual cases of tenure and promotion.

A timeline consisting of nine stages, detailed in Table 3, was developed for the OLTP. Each stage was associated with role information and that stage's required actions.

Table 3: Online Tenure and Promotion Timeline

Stage	Date Range	User Type (Role)	Actions
System Setup	Aug 1 - Aug 15	Administrator	Add candidates to system Set up department and college committees
Candidate	Aug 16 - Sep 15	Candidate	Upload Supporting Document, Narrative Statements, and CV
		Department Chair	Upload Course Load, Peer Review
Department Committee Review	Sep 16 - Oct 07	Department Committee Chair	Upload Department Committee Report Enter Committee Votes Sign Committee Report
		Department Committee Member	Review Committee Report Sign Committee Report
Department Chair Review	Oct 08 - Oct 15	Department Chair	Upload Department Chair Report Enter Recommendation Sign Chair Report
College Committee Review	Oct 15 - Dec 15	College Committee Chair	Upload College Committee Report Enter Committee Votes Sign Committee Report
		College Committee Member	Review Committee Report Sign Committee Report
College Dean Review	Dec 16 - Feb 01	College Dean	Upload College Dean Report Enter Recommendation Sign Dean Report
Vice President Review	Feb 02 - Mar 01	Vice President	Upload Vice President Report Enter Recommendation Sign VP Report
President	Mar 02 - Apr 01	President	Upload President Report Enter Recommendation Sign President Report

At each stage, the system's users can invoke actions that are specific to that stage and their role. Other actions that can be performed by all roles at their given stage include viewing documents, recommendations, votes, and signatures from the previous stage. In

order to view the artifacts from a previous stage, all actions that are required for that stage must have been completed. This includes the uploading of required reports, the recording of required recommendations and votes, and the signing of all required reports by all associated users.

Users of the system should only see candidates that they have permission to view. These permissions are determined by a user's membership in colleges, departments, department committees, and college committees. This includes two administrative roles that monitor the system. The college administrator and university administrator can view candidates in their college and the university respectively.

Initial OLTP Design

The designers considered several security NFRs requiring certain elements to be well defined. A role represents a certain type of system user. For example, the candidate role is associated with individuals who are applying for tenure and/or promotion. Chairs and members who are on a committee will get the appropriate committee chair and committee member role. A file type represents a type of file. Specific types of files include candidate files, committee reports, and administrative reports. Upload privileges for file types are specific to roles. For example, the candidate role uploads candidate files such as supporting documents, department chairs upload peer reviews and the department chair report (administrative), and committee chairs upload committee reports.

The system uses four types of artifacts: files that represent committee reports and administrative reports, votes that are associated with committee reports, recommendations that are associated with a chair or dean administrative reports, and signatures that are associated with all reports uploaded. Timeline elements determine when a role can upload and view files, enter and view committee votes and recommendations, sign reports, and view

signatures.

The system designers considered four main security related non-functional requirements. First, system designers must provide a mechanism for authenticating users using the university's authentication services and saving user information for later use (persistence). Second, users must be associated with one to many roles. It is possible for a committee member also be a candidate; this has to be considered when setting up committees and user permissions. Third, user functionality must be available during the appropriate stage, such as file upload, votes/recommendation entry forms, and report signing. Fourth, completed artifacts should only be viewable after the associated stage has been completed, a requirement that involves file and artifact statuses as well as timelines. Figure 27 is a partial class diagram showing the original OLTP design with only four roles represented.

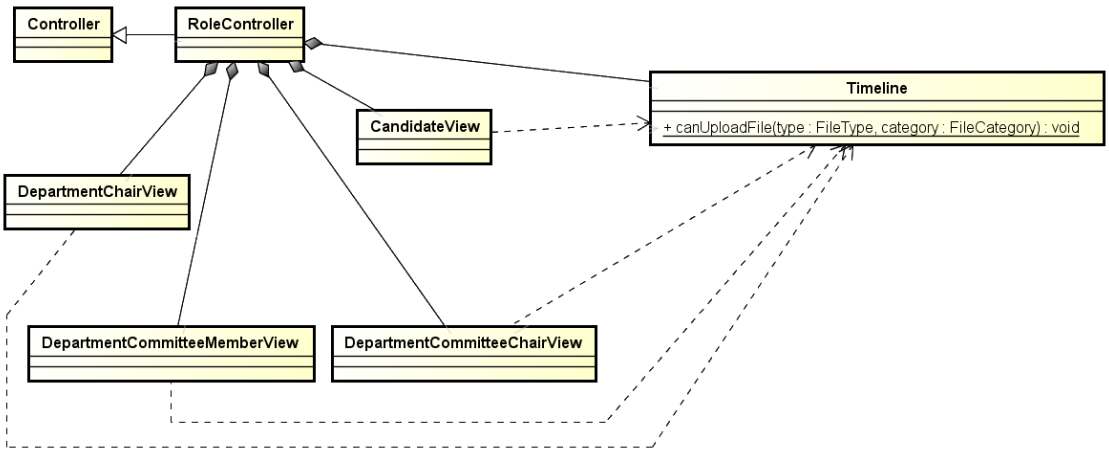


Figure 27: Original OLTP Class Design Simplified

The first NFR is met by authenticating users with LDAP efficiently but fails in how user information is made persistent once logged in. Currently, user information is stored in a global space and is not limited to the current user's instantiation of the system.

The second security concern is handled in a way that clutters the user interface and makes

debugging difficult. When a user is logged into the system, that user sees all functionality that is available to all of that user's roles. This makes the user interface confusing and debugging difficult, because there is no way to distinguish what functionality is associated with what role.

The third and fourth concerns are managed using a single class, the timeline class. This class has a high complexity and much repeated code. This makes it difficult to debug and modify as new roles and file types are added to the system. Using one complex class to handle all timeline security concerns increased the probability of inducing errors when modifying the system and made it difficult to verify that a security requirement has been met. This occurred several times when altering the system to fit the clients' changing needs. Table 4 lists the selected security NFRs along with the current implementation's deficiency.

Table 4: Online Tenure and Promotion Selected NFRs

OLTP Security NFR		**Current Implementation Deficiency**
OLTP-NFR-1	authenticate users with university's authentication services	once logged in, user information is stored in a global space
OLTP-NFR-2	user can fulfill one to many roles	user is presented with all functionality associated with all roles he/she has. This clutters the interface and increases debugging ability
OLTP-NFR-3	user functionality is dependent on current stage of timeline	all decisions based on timeline and artifact state is made in one class, the timeline class, that results in high complexity and repeated code. This makes debugging and modifying roles difficult
OLTP-NFR-4	artifacts are only viewable during appropriate stages if artifact is complete and user has permission	

Case Study - Limited View

To assess the effectiveness of the Limited View design pattern for satisfying non-functional requirements (NFR), its use was considered in the context of the OLTP's design. This pattern is ideal for the OLTP due to the number of different operations it supports. Limited View presents users with only those operations that they may access, dynamically building the GUI at run time. This pattern is the alternative to the Full View with Errors pattern. To compare the effectiveness of the Limited View and Full View patterns relative to the OLTP, a mockup of a Limited View implementation was constructed, as follows:

- Each user type, represented by role, was limited to the functionality available to that role.
- User functionality was limited to a subset provided by the real system includes canUploadFile, uploadFile, and canViewFile.
- Security considerations were analyzed and are described in section 4.

Five Roles were considered for this exercise: Candidate, Department Chair, Department Committee Chair, College Dean, and Administrator. Mockups were created as if the user were logged into the system as a department chair and the process had entered the "department chair review" stage. The department chair can perform the following functions: Upload Department Chair Report, Enter Department Chair Recommendation, and Sign Department Chair Report. Chairs may also delete the file that he/she uploaded by selecting the red delete icon next to the file.

The first of the three images presented below uses the Full View with Errors showing all options available for the given screen to all users. The second image highlights the functionality only available to the current user. All other options trigger an access denied

error. The third image shows the same interface using the Limited View pattern. The user's functionality is available to the user by the use of buttons that is grouped with other functionality illustrated in figure 28. Figure 29 shows the functionality highlighted while figure 30 shows how the GUI looks with Limited View.

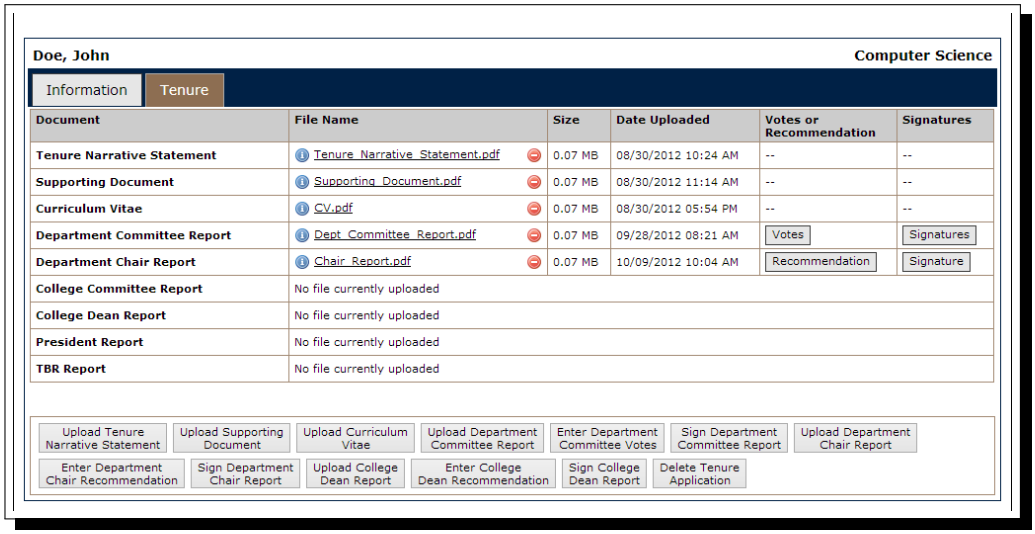


Figure 28: GUI using Full View with Errors Pattern

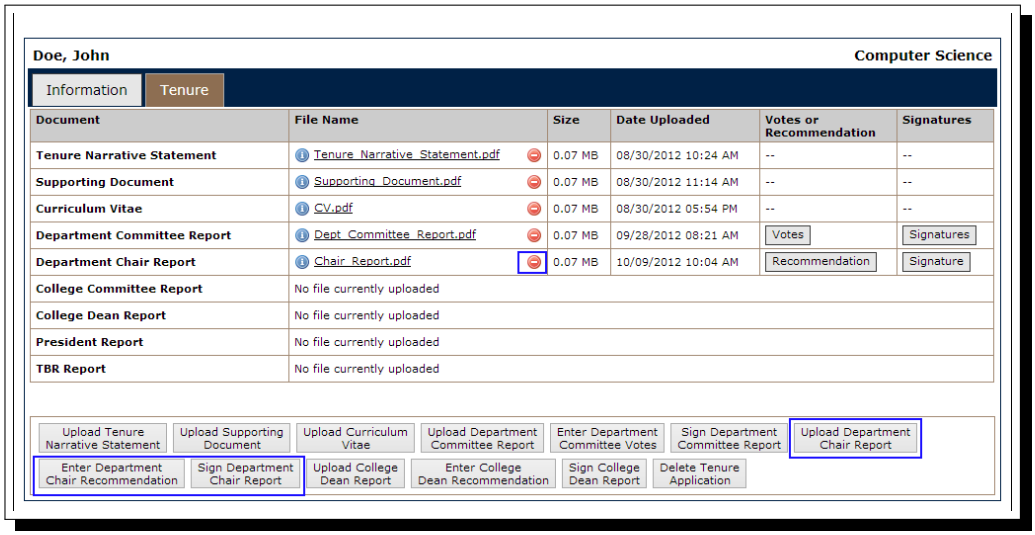


Figure 29: GUI with Relevant Functionality Highlighted

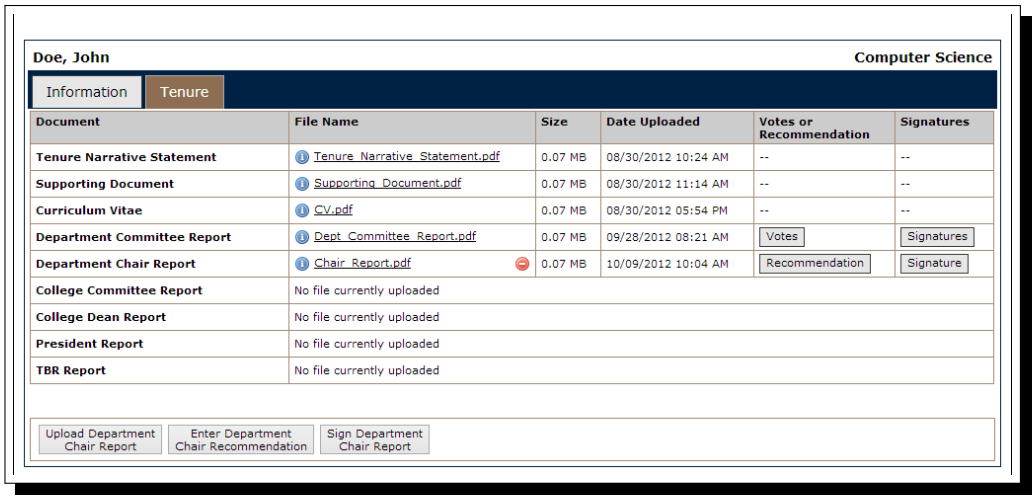


Figure 30: GUI using Limited View Pattern

Result

Using the Limited View pattern unclutters the user interface, making it easier to use. Even though it is more difficult to limit the functionality presented, the user is less confused when using the system. Feedback shows that the application of the Limited View pattern resolves issues that arose due to user playing multiple roles within the system detailed in the description of OLTP-NFR-2. This pattern makes the verification of security requirements easier.

Case Study - Role-Based Access Control

The Role-Based Access Control (RBAC) design pattern has been applied to a user case from the OLTP system to assess its effectiveness. The use case handles the following behavior:

- Each user can be associated with one to many roles.
- Functionality within the system is determined based on the user's current role.
- Artifacts represented in this design only include File and Votes.

- Security considerations are described in table 4.

Five Roles were considered for this exercise: Candidate, Department Chair, Department Committee Chair, College Dean, and Administrator. The class diagram in figure 15 shows design patterns original elements. The class diagram in figure 31 shows the OLTP partially redesigned using the Role-Based Access Control pattern.

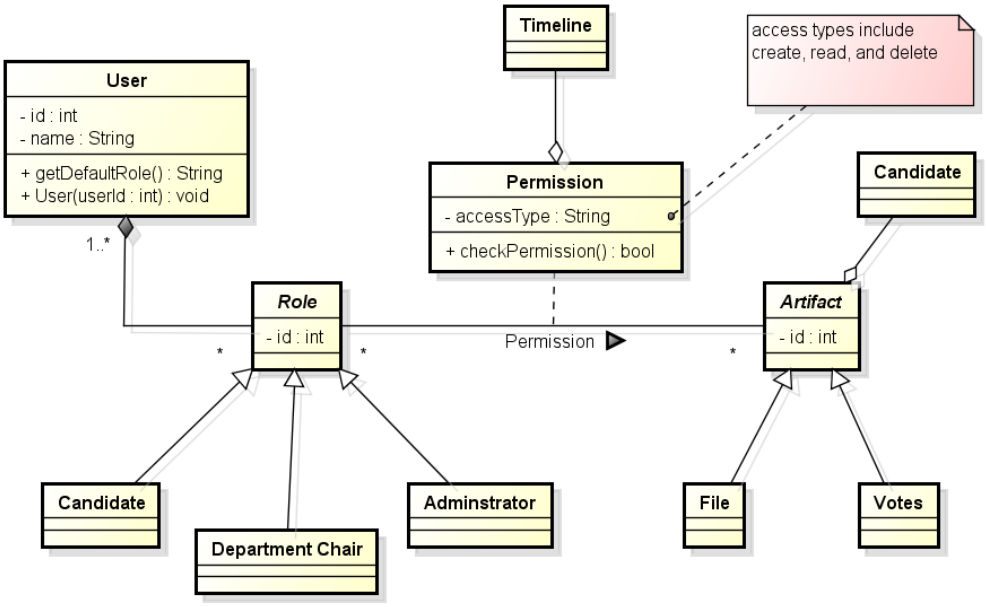


Figure 31: Class diagram using the Role-Based Access Control pattern

Result

The RBAC pattern isolates security checks into a Permission class. The Permission class represents the multiplicity of permissions associated between a Role and an Artifact. The Permission Class uses the Timeline class to determine which access action is allowed on the Artifact. Access types include creation of a new artifact, reading (accessing) the artifact, and deleting the artifact. The pattern associates multiple roles with a user, allowing the system to switch a user's permissions by switching the active role.

By separating the permissions from the Role class within the system, developers can

create new permissions without having to modify code within each Role class. Application of the RBAC pattern to the OLTP resolves some deficiencies apparent in the initial system implementation. OLTP-NFR-2 is handled by associating a user with multiple roles and giving the user the ability to switch between roles. OLTP-NFR-3 and OLTP-NFR-4 were handled by placing the timeline and the security concerns into different classes: Permission, Timeline, and Artifact. Decoupling these concerns resulted in the Permission class using the Timeline to make decisions based on what stage the process was in. This separation decreased code duplication and simplified the security code. This pattern fits the application's needs because the OLTP has many users with relatively few roles. The small number of roles makes managing permissions by role easier than managing permissions by individual users.

Case Study- Secure State Machine

The Secure State Machine pattern has been applied to Online Tenure and Promotion System (OLTP) in a use case. The following behavior in the use case has been designed and implemented using the Secure State Machine pattern:

- A user must log in before using the system.
- Each type of user, represented by role, is limited to the functionality available to that role.
- User functionality was limited to a subset provided by the real system includes canUploadFile, uploadFile, and canViewFile.
- Security considerations are described in table 4.

The class diagram in figure 22 shows design patterns original elements. The class diagram in figure 22 show the OLTP partially redesigned using the Secure State Machine pattern. Figure 32: Class diagram using the Secure State Machine pattern.

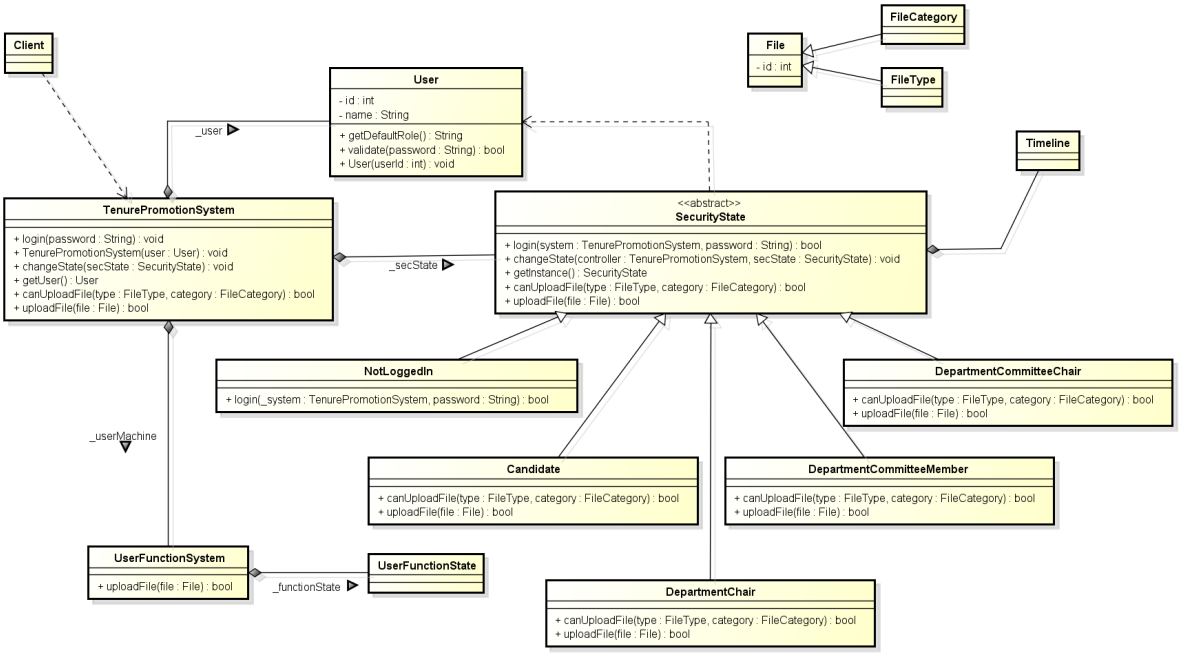


Figure 32: Class diagram using the Secure State Machine pattern

Result

Using different concrete SecurityStates separates security concerns for different system users into different classes. Each concrete SecurityState has a matching timeline that determines what functionality is available to the specific role based on that process's stage. The SecurityState will encapsulate the user information once he/she is authenticated. This separation allows for the system to easily accommodate security changes that the client may request. To modify a given role's security concerns, programmers can modify the relevant SecurityState class instead of digging through a very large class that would handle all security concerns. Adding a new role to the system is achieved by adding a new SecurityState and Timeline. Smaller, concise concrete classes reduce errors when adding new roles to the system.

Application of the SecureState pattern to the OLTP resolves deficiencies apparent in the initial system implementation. OLTP-NFR-1 is resolved by storing user information in the

concrete SecurityState class as opposed to the global space. This hides user information from other parts of the system. OLTP-NFR-2 is handled by forcing the user to assume one role at a time. This limits the functionality presented to the user current role instead of presenting all functionality available to a user. In addition to having the login functionality, the system will require the ability for users to switch between roles. This was not included in the design to simplify diagrams. OLTP-NFR-3 and OLTP-NFR-4 were handled by splitting the timeline and the security concerns into two classes: Timeline and SecurityState. Decoupling these concerns resulted in the SecurityState class using the Timeline to make decisions based on the process's current stage. This separation decreased code duplication and simplified the security code.

CHAPTER 5

SECURITY TEACHING MODULES

Modules for teaching security in a computer science curriculum have been developed to demonstrate techniques and security design patterns to meet the NFRs detailed in Chapter 2. These modules include a description, objective, and activities for each security module. Modules are listed with the NFR category it covers along with security design patterns that could be used in the module. Similiar modules have been developed in [20], [19], and [2].

The modules presented here, as a rule, give students practice in security-related concepts by tasking them with two to three assignments. The first assignment presents students with a discussion worksheet that explains the security issue at hand, along with questions to answer. The second assignment presents students with an application to test and attempt to break in a way that relates to the security issues. The third assignment presents students with the source code for the application in previous assignment. The student is instructed to modify the code in an attempt to address the security issue. Table 5 lists the courses covered along with the concentrations the courses are in, the year the course is taught, and the security module developed for it.

Table 5: Catalog of Security Design Patterns

Course	**Concentration**	**Year**	**Security Module**
CS1: Introduction to Programming I	CS, IT, IS	Freshman	Integer Overflow
CS2: Introduction to Programming II	CS, IT, IS	Freshman	File Access
WEB1: Introduction to Web Development	IT, IS	Freshman	HTTPS
DB: Database Fundamentals	CS, IT, IS	Sophomore	SQL Injection
WEB2: Server-side Development	IT, IS	Sophomore	Input Validation

CS1: Introduction to Programming I

Introduction to Programming I is the first programming intensive course in the ETSU Computer Science Department undergraduate curriculum. It teaches the basic programming concepts, including data types, control flow, classes, and methods. A security assignment for this course needed to be simple enough for beginning computer science students to understand. Instead of adding a new security module to the course and increasing the course material, an existing topic, the integer data type, was extended to include a unit on integer overflow. Integer overflow occurs when an arithmetic operation generates a value that is too large to store relative to a target data type. Overflow can result in erroneous computations, due to result truncation and/or rollover, or program failure, depending on a language's implementation. The concept of overflow also applies to other primitive types such as Java's short and long data types. The teaching module for integer overflow includes three activities for students to perform.

The first, discussion assignment for the CS1 unit presents students with a short narrative and questions covering integer overflow. Students are then given a Java program that simulates a bank account. Students are asked to break the program by using values that exceed what the data type (short) used to hold the balance can store and asked questions about the assignment. Finally, students are given the code for the simulation from the previous assignment. They are asked to modify the program to fix the security flaw caused by overflow. This module also includes the answer key for each assignment and source code for use by the facilitator.

This module demonstrates an example of data integrity vulnerability. There are no patterns given in the security design pattern catalog that could address this vulnerability.

CS2: Introduction to Programming II

Introduction to Programming II is a continuation of CS1. It covers topics that include advanced object oriented programming (OOP) paradigm, exception handling, graphical user interface (GUI) development, and file processing. The security component for CS2 proposed here extends the file processing module to show the security issues involved with using files in a program.

Security concerns related to file access include vulnerabilities involving pathnames, permissions, and race conditions. Improperly managed pathnames can afford access to parts of a file system that should be off-limits to the pathname's user, via symbolic links or directory walking using "..", the path to a file system object's parent directory. Improperly configured file permissions may allow a user to delete files without proper authorization. Race conditions can allow two or more processes to interfere with each other's operation by updating common files in incompatible ways.

The teaching module for file access includes three activities. The first is a discussion assignment on pathname canonicalization, the practice of ensuring that all files are referred to by a valid canonical path. The second gives students a Java program that simulates a bank account. This program will use a file to store account information. Students are asked to break the program by using filepaths that access files the program is not intended to access. Students are asked to investigate security issues associated with pathname vulnerabilities. The third gives students the code for this exercise. It asks them to modify the code to fix the security flaw caused by pathname vulnerability. This module also includes the answer key for each assignment and source code for use the facilitator

This module demonstrates examples of security vulnerabilities involving integrity or confidentiality. Improper file access could allow an attacker to gain access to a resource he/she

is not authorized to access or allow an attacker to pass parameters to an application he/she may have modified. Pathname canonicalization is an implementation level pattern that can help address some issues with file access.

WEB1: Introduction to Web Development

Introduction to Web Development tasks students with creating static websites using hypertext markup language (HTML) and cascading style sheets (CSS). Students are taught web design principles, graphics, forms, communication protocols, and so on. For this course, a new teaching module was created to teach the differences between the standard hypertext transfer protocol (HTTP) and the secure version of this protocol. The latter, Hypertext Transfer Protocol Secure (HTTPS), extends HTTP by placing the HTTP layer on top of the Secure Socket Layer/Transport Layer Security (SSL/TLS) layer. By using HTTP on top of SSL/TLS, web servers can use authentication to check the validity of requests, to protect against man-in-the-middle attacks. The teaching module for HTTPS includes only one activity: a narrative explaining HTTPS with the advantage and disadvantages of using HTTPS. The student is given several questions to answer about this narrative, along with some questions that will require internet research. This module also includes the answer key for the assignment.

DB: Database Fundamentals

Database Fundamentals, an introductory course on databases, teaches students how to create databases, connect to databases, and interact with databases using Structured Query Language (SQL). Students also learn what a RDBMS is, what primary and foreign keys are, how to build SQL queries, how to aggregate data, and how to debug SQL. For this course, a new teaching module was created to teach students what SQL injection is and how to

protect against it. SQL injection involves the use of incorrectly preprocessed input to insert malicious code into requests made to a database. Failure to account for SQL injection can result in unauthorized access to or the corruption or destruction of database content.

The teaching module for SQL injection includes three activities. The first is a discussion assignment on SQL injection and the practice of sanitizing user input to protect databases from attackers. The second gives students a Java program that simulates a bank account. This program uses a database to store account information. The student is asked to delete all accounts using SQL injection. The third gives students the simulation's code. It asks them to modify the code to fix the security flaw associated with SQL injection. This module also includes the answer key for the assignment.

This module demonstrates a data integrity vulnerability that can be addressed using the input validation pattern from the security design pattern catalogue.

WEB2: Server-side Development

Server-side Development, an introductory course on server application development, teaches students how to develop and maintain web applications, object-oriented PHP, sessions, database integration, and web site security. Students learn server-side scripting languages that include PHP and Coldfusion. For this course, a new teaching module was created to teach students proper techniques for validating user input. Improper validation can lead to attackers gaining control of the system, modifying underlying databases, and injecting malicious code.

The teaching module for Input Validation includes two activities. The first is a discussion assignment on Input Validation and the need to validate user provided data. The second gives students a PHP script that simulates a sign-up form for an online account. It asks them to test the form and add proper input validation. This module also includes the answer

key for the assignment. This module demonstrates a data integrity vulnerability that can be addressed using the input validation pattern from the security design pattern catalogue.

Status of Future Modules

The modules presented in this chapter cover freshman and sophomore level courses, but include few examples of actual security design pattern use. Modules for junior and senior level courses are under development. These modules will demonstrate more uses for patterns provided in the security design catalogue since students will have advanced skills to solve complex problems.

CHAPTER 6

CONCLUSION

Strategies for software development often slight security-related considerations during systems development. Three possible reasons for this include the difficulty of developing realizable security requirements, the difficulty of applying appropriate techniques, and a lack of training in secure systems design. The research presented in this thesis proposes a three-part strategy for addressing these concerns. Part 1 involves the use of questions for eliciting precise, realizable requirements for system security is presented in Chapter 2. These questions are derived from a two-level characterization of system security, based on work by Chung et. al. [8], that defines security in terms of a 2-level hierarchy, with confidentiality, integrity, availability, and accountability at level 1. Part 2 involves the use of a novel framework for relating these level-2 attributes to previously published, carefully documented strategies, a.k.a. patterns, for secure software development presented in Chapter 3. Case studies are included that suggest the framework's effectiveness, involving the application of three patterns for secure design (Limited View, Role-Based Access Control, Secure State Machine), presented in Chapter 4. Part 3 involves the development of teaching modules for integrating security design patterns into lower-division computer science courses presented in Chapter 5. Here, five such modules, related to integer overflow, input validation, HTTPS, files access, and SQL injection, are proposed for helping future developers become aware of patterns and their potential value in secure software development.

Decomposing security requirements into a hierarchy yields a three-level hierarchy. Security, at the top, is decomposed into four non-functional requirements (NFR): confidentiality, integrity, availability, and accountability. Further decomposition of these four NFRs results in the third level with a total of eleven NFRS. Confidentiality is decomposed into authentication and authorization. Integrity is decomposed into completeness, precision, and validity.

Accountability is decomposed into logging, monitoring, and reporting. Availability is decomposed into usability, reliability, and compatibility. This hierarchy can be used to solicit a system's quality attributes from stakeholders and to develop that system, based on associations between security NFRs and patterns that realize these requirements, relative to different stages in the software development process

To evaluate the effectiveness of security design patterns, several patterns were applied to the problem of redesigning a real-world application, the Online Tenure and Promotion (OLTP) system. The use of these patterns in this system's design yielded qualitative improvements in the redesigned system, as documented in Chapter 4.

Finally, the NFRs and security design patterns were used to develop teaching models that illustrate the importance of security. These security teaching modules were developed for five lower-division courses in ETSU's computer science curriculum. The modules are intended to illuminate material that is already being presented in those courses. Security aspects with related practices or security design patterns are given with multiple activities and objectives. All teaching modules have a discussion activity that requires students to answer questions and do a modest amount of research. For classes that are programming intensive, the teaching modules have programming activities that require students to break a program by taking advantages of security flaws, then fix the program by applying knowledge from the discussion activity.

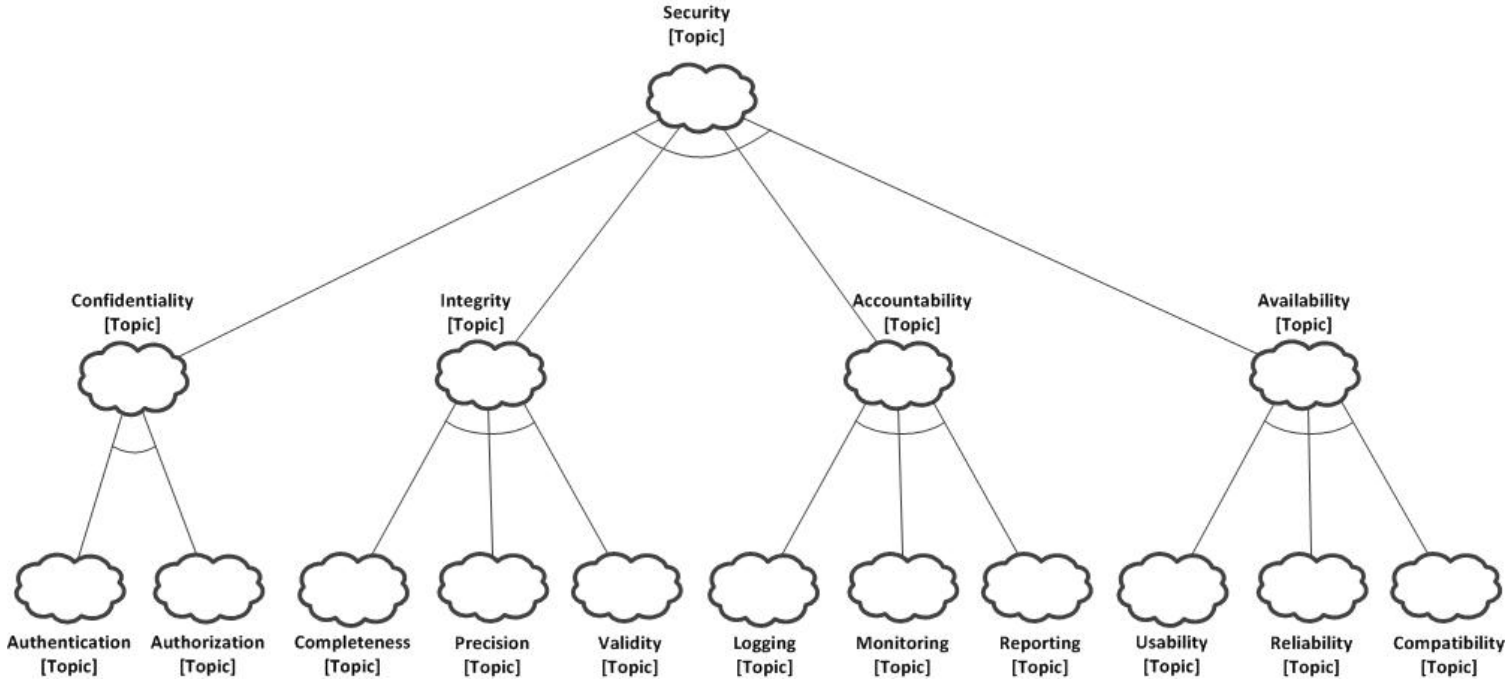
Further Research

Additional research could be performed in several areas discussed in chapter two through five. First, security issues could be decomposed at finer levels of detail to obtain more focused questions for NFR elicitation. Second, additional patterns could be extracted from the literature or developed to fill gaps in the security hierarchy. Third, all security design patterns

could be tested to evaluate their effectiveness. Security design patterns could be evaluated on how well they work together when addressing security non-functional requirements. Finally, the teaching modules should be evaluated to assess their effectiveness. An analysis could be performed on students that have taken courses with the security modules compared to those that have taken classes without the security modules. This would require several semesters to complete since modules cover several years of course work. Alternatively, analysis can be performed with two classes being taught in the same term with one class having the security modules and the other class not.

APPENDIX A

SECURITY HIERARCHY



APPENDIX B

SECURITY TEACHING MODULES

Security Module	**Course**	**Page**
Integer Overflow	CS1: Introduction to Programming I	85
File Access	CS2: Introduction to Programming II	95
HTTPS	WEB1: Introduction to Web Development	107
SQL Injection	DB: Database Fundamentals	118
Input Validation	WEB2: Server-side Development	126

Security Module: Integer Overflow

Description

Integer overflow is a security issue caused by overflowing a data type's storage limitations. Integer overflow occurs when arithmetic operation give a numeric value that is to big to be stored in integer variable.

Objective

The objectives of this security module are to illustrate the limitations that a data type can have and show the consequences of integer overflow can have on your system data.

Activities

There are three activities for this module. The first is a discussion assignment, where students are given a short narrative and questions covering integer overflow. Second, students are given a Java program they can run that simulates a bank account. The program uses a command line interface. The account starts with an initial balance of 0 dollars. The user has 3 options in the menu: deposit money, withdraw money, and exit. The data type for storing the balance will be a `short`. Ask the student to attempt to break the program by depositing and/or withdrawing money. Third, students are given the code for the previous assignment. They are asked to modify the program to fix the security flaw caused by overflow. They can do this in two ways: change the data type to be a long or check for max values before adding/subtracting the deposit/withdraw amount.

Module Contents

- Assignment 1: Integer Overflow Discussion
- Assignment 2: Break the Program
- Assignment 3: Fix the Program w/source code
- Answer key for each assignment
- Source code for IntegerOverflow class
- Source code for Menu class (used in IntegerOverflow program)

Assignment 1: Integer Overflow Discussion

What is Integer Overflow?

Integer overflow is a security issue is caused by overflowing a data type's storage limitations. Integer overflow occurs when arithmetic operations result in a numeric value that is to big to be stored in integer variable. A physical example of this problem can be seen with your car's odometer, which has only 6 dials. If your car's odometer is reading 999500 for having traveled 99,950.0 miles, what happens when you travel another 1000 miles? The odometer will roll over to 000500, resulting in the appearance of the car having only ever traveled 500 miles. This is an mechanical example of the the integer overflow problem. Even though it's the "Integer Overflow" problem, it apples to all primitive data types.

This same principal can be applied to primitive data types, such as Java's short data type. Short can only store values between -32,768 and 32,767 inclusively. Let say you have the variable `shortOdometerReading` defined as a `short`. If `shortOdometerReading` has a value of 31,890 and 3,000 was added to it. What would be the value stored in `shortOdometerReading`? -30646 When storing a value the system will only use the least significant bits that can be stored in a variable. Since the `short` data type can only store 16 bits anything over 16 bits is ignored. This cause the value to wrap around to the lowest number it can store and start counting up from there. Since the lowest value is -32,768 and the highest value is 32,767 the variable will wrap around when it hits 32,768.

To demonstrate the limitations of the data type `short` the below code is given for you to analyze and answer questions about.

```
1  short myShort = 32765;
2
3  myShort = myShort + 1;
4  System.out.println(myShort);
5
6  myShort = myShort + 1;
7  System.out.println(myShort);
8
9  myShort = myShort + 1;
10 System.out.println(myShort);
11
12 myShort = myShort + 1;
13 System.out.println(myShort);
14
15 myShort = myShort + 1;
16 System.out.println(myShort);
```

1. Choose the two answers that equal the minimum and maximum values that a `short` can store?

   (a) `-2,147,483,648`

   (b) `-32,768`

   (c) `2,147,483,647`

   (d) `32,767`

2. What will be the result of the `System.out.println(myShort)` statements on lines 4, 7, 9, 13, 16?

3. Explain what happens when the value being stored in `short` is to large to fit in the data type.

Assignment 2: Break the Program

In this assignment, you will be given a Java program for a simple bank account. This program will track the balance for your account. You can deposit and withdraw money from your account using the command line interface. To deposit money into your account, use option 1. The prompt will ask you "How much money to deposit:". You must input a positive number that will be added to your balance. To withdraw money out of your account, use option 2. The prompt will again ask you, "How much money to withdraw". You must enter a positive number that will subtracted from your balance.

Your assignment is to try and break the simple bank program. As you try to break the program, keep track of the amount you deposit or withdraw along with the expected balance and the actual balance. Hint: try depositing or withdrawing large amounts of money.

1. Did you have an actual balance that did not match your expected balance?

2. If yes, what was your actual balance and your expected balance?

3. Based on the values from the previous question, can you determine the date type being used to store the balance? What is that type?

Assignment 3: Fix the Program

The next page has the source code from the previous assignment, where you were asked to break the simple bank program. Your assignment is to identify what needs to be modified in the source code to fix the overflow problem. Use your answers from the previous assignment to help you identify three locations in the code needing modification. When asked to identify a line of code, provide the line number.

1. What data type is used to store the balance?

2. What other data types could be used to store the balance and why?

3. What is the first line of code that needs modified? What do you suggest it be changed to?

4. What is the second line of code that needs modified? What do you suggest it be changed to?

5. What is the third line of code that needs modified? What do you suggest it be changed to?

```
1  import java.util.Map;
2  import java.util.HashMap;
3  import java.util.ArrayList;
4  import java.util.Scanner;
5
6  class IntegerOverflow
7  {
8      public static void main(String[] args)
9      {
10         Scanner in = new Scanner(System.in);
11         short balance = 0;
12
13         Menu menu = new Menu("Main Menu");
14         menu.addOption(1, "Deposit Money");
15         menu.addOption(2, "Withdraw Money");
16         menu.addOption(9, "Exit");
17
18         while(true)
19         {
20             System.out.print("\n\n\n\n\nCurrent balance: " + balance);
21             switch(menu.getOption())
22             {
23                 case 1:
24                 {
25                     System.out.print("How much to deposit: ");
26                     short depositAmount = in.nextShort();
27                     balance += depositAmount;
28                     break;
29                 }
30
31                 case 2:
32                 {
33                     System.out.print("How much to withdraw: ");
34                     short withdrawAmount = in.nextShort();
35                     balance -= withdrawAmount;
36                     break;
37                 }
38
39                 case 9:
40                 {
41                     System.out.println("Exiting");
42                     System.exit(0);
43                     break;
44                 }
45             }
46         }
47     }
48 }
```

Answer Key

Assignment 1: Integer Overflow Discussion

1. Choose the two answers that equal the minimum and maximum values that a `short` can store?

   (b) -32,768 and (d) 32,767

2. What will be the result of the `System.out.println(myShort)` statements on lines 4, 7, 9, 13, 16?

   32766, 32767, -32768, -32767, -32766

3. Explain what happens when the value being stored in `short` is to large to fit in the data type.

   Arithmetic operations that result in a number to large for a given data type result in high order bits being discarded. The short data type only has 16 bits to represent a value. If a value needs more than 16 bits to be store, everything over 16 bits is discarded when being stored. This can corrupt your data.

Assignment 2: Break the Program

1. Did you have an actual balance that did not match your expected balance?

   The student should deposit or withdraw money until this occurs.

2. If yes, what was your actual balance and your expected balance?

   The student will have a discrepancy when he/she its the short's storage limit (-32,768 or 32767)

3. Can you determine the date type being used to store the balance? What is that type?

   The student should be able to determine that the data type being used is a short.

Assignment 3: Fix the Program

1. What data type is used to store the balance?

   short

2. What other data types could be used to store the balance and why?

   integer or double, both would allow for a much larger number to be stored

3. What is the first line of code that needs modified? What do you suggest it be changed to?

   Line 11 should be changed to `short int = 0`

4. What is the second line of code that needs modified? What do you suggest it be changed to?

   Line 26 should be changed to `int depositAmount = in.nextInt()`

5. What is the third line of code that needs modified? What do you suggest it be changed to?

   Line 34 should be changed to `int withdrawAmount = in.nextInt()`

IntegerOverflow Source Code

```
import java.util.Map;
import java.util.HashMap;
import java.util.ArrayList;
import java.util.Scanner;

class IntegerOverflow
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        short balance = 0;

        Menu menu = new Menu("Main Menu");
        menu.addOption(1, "Deposit Money");
        menu.addOption(2, "Withdraw Money");
        menu.addOption(9, "Exit");

        while(true)
        {
            System.out.print("\n\n\n\n\nCurrent balance: " + balance);
            switch(menu.getOption())
            {
                case 1:
                {
                    System.out.print("How much to deposit: ");
                    short depositAmount = in.nextShort();
                    balance += depositAmount;
                    break;
                }

                case 2:
                {
                    System.out.print("How much to withdraw: ");
                    short withdrawAmount = in.nextShort();
                    balance -= withdrawAmount;
                    break;
                }

                case 9:
                {
                    System.out.println("Exiting");
                    System.exit(0);
                    break;
                }
            }
        }
    }
}
```

Menu Class Source Code

```
class Menu
{
    class Option
    {
        private int id;
        private String desc;

        public Option(int id, String desc)
        {
            this.id = id;
            this.desc = desc;
        }
        public int getId()
        {
            return this.id;
        }
        public String getDesc()
        {
            return this.desc;
        }
    }

    private int _subMenuKey;
    private ArrayList<Option> _options;
    private String _title;

    public Menu(String title)
    {
        this._subMenus = new HashMap<Integer, Menu>();
        this._options = new ArrayList<Option>();

        this._subMenuKey = 1;
        this._title = title;
    }

    public void addOption(int id, String desc)
    {
        this._options.add(new Option(id, desc));
    }

    public void clearOptions()
    {
        this._options.clear();
    }

    public int getOption()
    {
        Scanner in = new Scanner(System.in);
```

```
        while(true)
        {
            System.out.println("\n------------------------------------");
            System.out.println(" " + this._title + "\n");

            for (Option option : this._options) {
                System.out.println(" " + option.getId() + " - " + option.getDesc());
            }
            System.out.println("------------------------------------");

            System.out.print("Select option: ");
            int optionInt;
            String optionStr = in.nextLine();

            //--- Attempt to convert the inputed string to an integer
            try {
                optionInt = Integer.parseInt(optionStr);
            } catch (NumberFormatException ex) {
                System.out.println("Error: It appears you did not input a option number");
                continue;
            }

            //--- Check if the option exists in our menu
            for (Option option : this._options) {
                if (option.getId() == optionInt) {
                    return optionInt;
                }
            }
            System.out.println("Error: Invalid option selected");
        }
    }
}
```

Security Module: File Access

Description

File access has many security issues including pathnames, permissions, and race conditions. Pathnames can be vulnerable to symbolic links or directory walking using ".." , the path to the directory above. Permissions may allow a user to delete files without proper authorization. Race conditions can allow other processes to insert data into a system by writing to application files while a program is processing the same file.

Objective

The objectives of this security module are to illustrate and explain security issues associated with file access and the consequences file access can have on your system's data.

Activities

There are three activities for this module. The first is a discussion assignment on pathname canonicalization, the practice of ensuring that all files are referred to by a valid canonical path. Second, students are given a Java program that simulates a bank account. This program will use a file to store account information. The program uses a command line interface. The user has 3 options in the menu: open account file, display account info, and exit. The student will investigate security issues associated with pathname vulnerabilities. Third, students are given the code for the simulation above. They are asked to modify the program to fix the security flaw caused by pathname vulnerable.

Module Contents

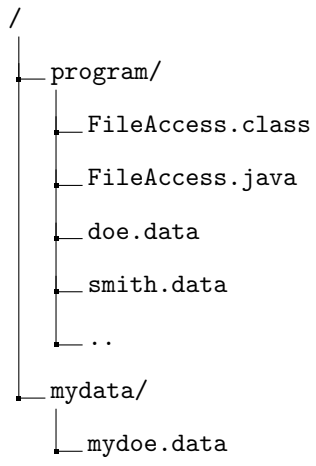
- Assignment 1: File Access Discussion
- Assignment 2: Break the Program
- Assignment 3: Fix the Program w/source code
- Answer key for each assignment
- Source code for FileAccess class
- Source code for Menu class (used in FileAccess program)
- Complete source code for solution to Assignment 3

Assignment 1: File Access Discussion

File Access

File Access can be a very important part of an application. It can be used load configuration, take input, or store user data. There are a number of security issues when dealing with files, such as race conditions, pathnames, and file permissions. Race conditions can occur if you have multiple programs using the same file. If two programs read data from the same file, then perform calculations on the data, the last program to write back the calculated value will win. Pathname issues can expose data that the user may not be intended to access. For example, if we trust the user to input the pathname for a file to be read, this could allow the user to walk up the directory tree to other files. File permissions can cause programs to crash if it does not have the appropriate permissions to open the file.

This assignment will concentrate on vulnerabilities involving pathnames. Use the following directory structure to answer questions.



1. What is the absolute path to the folder that contains the `doe.data` file?

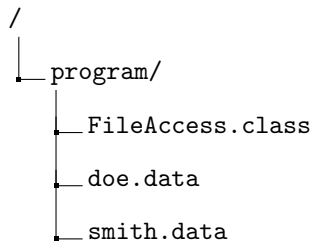
2. What is the absolute path to the folder that contains the `mydoe.data` file?

3. If you are in the `/program/` folder, what is relative path used to change to the `/mydata/` folder?

4. If you run the FileAccess java program what is the absolute path to your working directory?

Assignment 2: Break the Program

In this assignment, you will be given a Java program for a simple bank account. This program uses files to store account information. You can load account files and display account information using the command line. To load an account file, use option 1. The prompt will ask you for the "Account file name:". You must input a file name that represents the account you wish to load. To display the account information, use option 2. Once the program displays the account information, you will be returned to the menu, where you can load a different account file. You will be given a zip archive containing the simple bank account program. Below is the directory structure contained in the zip archive. The .data files located in the program directory represent the files where account information is stored. The .data files have the account holder's name in the first line and the account balance in the second line. As users of the program, the .data files are not supposed to be accessible to you.



Your assignment is to trick the program into loading data files that we do have access to. To accomplish this create a new folder in the root folder, not in the program folder. Then, create a `.data` file with your name as in the first line and your desired balance in line two. For this assignment, keep the balance below $1 million dollars. When the program asks you to enter the account's file name, use `doe.data` the first time. Then attempt to have the program load the file you created.

1. What is the balance of John Doe's account when you used the `doe.data` file?

2. What is the filename you uses to get the program to use your version of `doe.data`? (The previous assignment should be a hint)

3. What balance does the program think John Doe's account has when using your version of the `doe.data`?

Assignment 3: Fix the Program

In this assignment, you will fix the source code from the previous assignment to account for file path vulnerabilities. There may be several approaches to this problem. The one suggested here is to compare the absolute directory to ensure it matches the expected path. To do this you will use the Path class provided by Java. This class is the java.nio package available only in Java 7 or higher. Java provides a tutorial on Path operations here - http://docs.oracle.com/javase/tutorial/essential/io/pathOps.html. The javadocs for Path is located here - http://docs.oracle.com/javase/7/docs/api/java/nio/file/Path.html. Pseudo code for the solution is presented below.

The solution will need to be added after the program creates the File object on line 33. Use the `strFilePath` to create a Path object via the Paths.gets() method. Pay attention to the parameters needed by the Paths.gets() method. Use the websites provided earlier for documentation.

```
GET FILE PATH FOR FILE (Hint: use Paths.gets() method)
GET ABSOLUTE PATH FOR FILE PATH
GET PARENT DIR FOR THE FILE ABSOLUTE PATH

GET APP PATH FOR WORKING DIRECTORY
GET ABSOLUTE PATH FOR APP PATH
GET PARENT DIR FOR APP ABSOLUTE PATH

IF FILE'S PARENT DIR NOT EQUAL APP'S PARENT DIR
— DISPLAY ERROR
— BREAK
```

1. Test your solution to ensure that it works.

2. Turn in a screen shot of the program accessing the original `doe.data` file.

3. Turn in a screen shot of the program showing that you **cannot** access the `doe.data` file you created earlier.

Answer Key

Assignment 1: File Access Discussion

1. What is the absolute path to the folder that contains the `doe.data` file?

   /program/doe.data

2. What is the absolute path to the folder that contains the `mydoe.data` file?

   /mydata/doe.data

3. If you are in the `/program/` folder, what is relative path used to change to the `/mydata/` folder?

   ../mydata

4. If you run the FileAccess java program what is the absolute path to your working directory?

   ../program

Assignment 2: Break the Program

1. What is the balance of John Doe's account when you used the `doe.data` file?

   $1 hundred dollars

2. What is the filename you uses to get the program to use your version of `doe.data`?

   ../mydata (assuming the student used the same structure as assignment 1)

3. What balance does the program think John Doe's account has when using your version of the `doe.data`?

   (will be different per student)

Assignment 3: Fix the Program

The following code is one possible solution. A full solution is given at the end of this module.

```
1     //--- Get absolute path of the folder containing the file given
2     Path filePathObject = Paths.get(fileIn.getPath());
3     Path fileLocation = filePathObject.toAbsolutePath().getParent();
4     //--- Get absolute path to the folder the application is running in
5     Path appPathObject = Paths.get(".");
6     Path appLocation = appPathObject.toAbsolutePath().getParent();
7     //--- If the app location doesn't match the file location, error and break
8     if ( !appLocation.equals(fileLocation) ) {
9         System.out.print("\nERROR: the file you loaded is outside of program directory");
10        break;
11    }
```

FileAccess Source Code

```
import java.util.*;
import java.io.*;
import java.nio.charset.*;

class FileAccess {

    public static void main(String[] args) {

        Scanner in = new Scanner(System.in);

        Menu menu = new Menu("Main Menu");
        menu.addOption(1, "Open Account File");
        menu.addOption(2, "Display Account Info");
        menu.addOption(9, "Exit");

        String name = null;
        String balance = null;

        while(true)
        {
            switch(menu.getOption())
            {
                case 1:
                {
                    System.out.print("Account file name: ");
                    String fileName = in.nextLine();

                    try {
                        File fileIn = new File(fileName);
                        String strFilePath = fileIn.getPath();
                        System.out.print("\nAttempting to open file at " + strFilePath);

                        name = file.nextLine();
                        balance = file.nextLine();
                    } catch (Exception e) {
                        System.out.print("\nError: failed loading file");
                    }
                    break;
                }

                case 2:
                {
                    if ( null == name ) {
                        System.out.print("\nError: account file not loaded");
                    } else {
                        System.out.print("\nName : " + name);
                        System.out.print("\nBalance : " + balance);
                    }
                    break;
```

```
                }

                case 9:
                {
                    System.out.println("Exiting");
                    System.exit(0);
                    break;
                }
            }
        }
    }
}
```

Content for doe.data file

John Doe

100

Menu Class Source Code

```
class Menu
{
    class Option
    {
        private int id;
        private String desc;

        public Option(int id, String desc)
        {
            this.id = id;
            this.desc = desc;
        }
        public int getId()
        {
            return this.id;
        }
        public String getDesc()
        {
            return this.desc;
        }
    }

    private int _subMenuKey;
    private ArrayList<Option> _options;
    private String _title;

    public Menu(String title)
    {
        this._subMenus = new HashMap<Integer, Menu>();
        this._options = new ArrayList<Option>();

        this._subMenuKey = 1;
        this._title = title;
    }

    public void addOption(int id, String desc)
    {
        this._options.add(new Option(id, desc));
    }

    public void clearOptions()
    {
        this._options.clear();
    }

    public int getOption()
    {
        Scanner in = new Scanner(System.in);
```

```
        while(true)
        {
            System.out.println("\n-----------------------------------");
            System.out.println(" " + this._title + "\n");

            for (Option option : this._options) {
                System.out.println(" " + option.getId() + " - " + option.getDesc());
            }
            System.out.println("-----------------------------------");

            System.out.print("Select option: ");
            int optionInt;
            String optionStr = in.nextLine();

            //--- Attempt to convert the inputed string to an integer
            try {
                optionInt = Integer.parseInt(optionStr);
            } catch (NumberFormatException ex) {
                System.out.println("Error: It appears you did not input a option number");
                continue;
            }

            //--- Check if the option exists in our menu
            for (Option option : this._options) {
                if (option.getId() == optionInt) {
                    return optionInt;
                }
            }
            System.out.println("Error: Invalid option selected");
        }
    }
}


import java.util.*;
import java.io.*;
import java.nio.charset.*;
import java.nio.file.*;

class FileAccess {

    public static void main(String[] args) {

        Scanner in = new Scanner(System.in);

        Menu menu = new Menu("Main Menu");
        menu.addOption(1, "Open Account File");
        menu.addOption(2, "Display Account Info");
        menu.addOption(9, "Exit");

        String name = null;
        String balance = null;
```

```
while(true)
{
    switch(menu.getOption())
    {
        case 1:
        {
            System.out.print("Account file name: ");
            String fileName = in.nextLine();

            try {
                File fileIn = new File(fileName);
                String strFilePath = fileIn.getPath();
                System.out.print("\nAttempting to open file at " + strFilePath);

                //--- Get absolute path of the folder containing the file given
                Path filePathObject = Paths.get(fileIn.getPath());
                Path fileLocation = filePathObject.toAbsolutePath().getParent();
                //--- Get absolute path to the folder the application is running in
                Path appPathObject = Paths.get(".");
                Path appLocation = appPathObject.toAbsolutePath().getParent();
                //--- If the app location doesn't match the file location, error and break
                if ( !appLocation.equals(fileLocation) ) {
                    System.out.print("\nERROR: the file you loaded is outside of program
                        directory");
                    break;
                }

                name = file.nextLine();
                balance = file.nextLine();
            } catch (Exception e) {
                System.out.print("\nError: failed loading file");
            }
            break;
        }

        case 2:
        {
            if ( null == name ) {
                System.out.print("\nError: account file not loaded");
            } else {
                System.out.print("\nName : " + name);
                System.out.print("\nBalance : " + balance);
            }
            break;
        }

        case 9:
        {
            System.out.println("Exiting");
            System.exit(0);
```

```
                    break;
                }
            }
        }
    }
}
```

Security Module: HTTPS

Description

Hypertext Transfer Protocol Secure (HTTPS) extends the Hypertext Transfer Protocol (HTTP) by placing the HTTP layer on top of the Secure Socket Layer/Transport Layer Security (SSL/TLS) layer. By using HTTP on top of SSL/TLS, web servers can use authentication to check validity of requests to protect against man-in-the-middle attacks.

Objective

The objective of this security module is to discuss security issues associated with making requests over the internet using HTTPS and the vulnerability of sending requests using HTTP.

Activities

There is only one activity for this module. The activity involves a narrative explaining HTTPS with the advantage and disadvantages of using HTTPS. The student will be given several questions to answer about this narrative along with some questions that will requires internet research.

Module Contents

- Assignment 1: HTTPS Discussion
- Answer key the assignment

Assignment 1: HTTPS Discussion

Hypertext Transfer Protocol Secure (HTTPS) is an extension of the Hypertext Transfer Protocol (HTTP) that adds security by using Secure Socket Layer/Transport Layer Security SSL/TLS. By using HTTPS, users of the internet can verify that servers are who they say they are, and they can be protected from man-in-the-middle attacks. These attacks include eavesdropping to steal sensitive information, such as a user's bank account credentials, and inject new messages that may contain spyware or malware. HTTPS uses the same uniform resource identifier (URI) scheme as HTTP with the exception of the scheme identifier at the beginning of the URL. When using `https://` instead of `http://`, the broswer will encrypt requests via the SSL/TLS layer. The request URL, query parameters, headers, and cookies are all encrypted. The addresses and ports numbers can not be encrypted, since they are used to route the request to the appropriate server.

To use HTTPS, the server is required to have a public key certificate created by an administrator. This certificate will be signed using a private key of a trusted certificate authority (CA). When a request is made by the web broswer (client), the certificate is retrieved and then verified using the public key for the CA. Web browsers will be installed with all major CAs' public keys. The web browser then creates a shared key that is sent back to the server to ensure all communication between the web browser (client) and server is encrypted and therefore protected. The CA is providing services that prevent man-in-the-middle attacks.

Since HTTPS has to encrypt sent messages and decrypt received messages, communication between the client and server can be slower than it would be using just HTTP. This is especially true because all pages of a website need to use HTTPS if the connection needs to be secure. This includes retrieving all images, cascading style sheets (CSS), and Javascript files.

The following questions can be answered using the above narrative or by a little online research.

1. In a few sentences, describe what HTTPS is.

2. Why should one use HTTPS over HTTP?

3. How does HTTPS work?

4. What does HTTPS protect us from? Give at least two things.

5. What is needed to use HTTPS?

6. Name at least one major trusted certificate authority.

7. What ports do HTTP and HTTPS use?

8. Give one difference between HTTP and HTTPS?

Answer Key

Assignment 1: HTTPS Discussion

1. In a few sentences, describe what HTTPS is.

   HTTPS is an extension of HTTP to add security. This is done by layering HTTP on top of SSL/TLS to encrypt requests sent between clients and server.

2. Why should one use HTTPS over HTTP

   If a site only uses HTTP then sensitive information is passed around in plain text. Attackers can intercept messages and view information such as your username/password or bank account information.

3. How does HTTPS work?

   When using a secure connection with HTTPS everything is encrypted. This includes the request URL, query parameters, headers, and cookies. The host address and port numbers can not be encrypted since they are needed to route the request to the appropriate server.

4. What does HTTPS protect us from? Give at least two things.

   HTTPS protects us from man-in-the-middle attacks that can inject information into or change requests. Requests could have malware or ads injected by attackers. Packet sniffers can also steal information from request such as username/passwords, bank account information, etc.

5. What is needed to use HTTPS?

   The server must have a public key certificate that is signed by a trusted certificate authority. The client (browser) must have the abilities to retrieve the key from the server and use certificate authorities to validate key. This implies that the client must trust the certificate authority.

6. Name at least one major trusted certificate authority.

   Symantec (VeriSign, Thawte,and Geotrust), Comodo, GoDaddy, and GlobalSign sign almost 90 percent of all certificates on the internet.

7. What ports do HTTP and HTTPS use?

   HTTP uses port 80 and HTTPS uses port 443.

8. What is one difference between HTTP and HTTPS

   HTTP use port 80 while HTTPS uses port 443. HTTP uses `http://` while HTTPS uses `https://`. HTTPS can be slower than HTTP.

Security Module: SQL Injection

Description

SQL injection is a security issue that involves inserting malicious code into requests made to a database. The security vulnerability occurs when user provided data is not correctly validated or filtered. Failure to account for SQL injection can result in sensitive data being be stolen or in unauthorized access.

Objective

The objectives of this security module are to illustrate and explain security issues associated with SQL injection and the consequences that not protecting against SQL injection can have on your system.

Activities

There are three activities for this module. The first is a discussion assignment on SQL injection and the practice of sanitizing user input to protect your database from attackers. Second, students are given a Java program that simulates a bank account. This program will use a database to store account information. The program uses a command line interface that displays the account available. The user has two options in the menu: view account and exit. The view account option will ask for the name of an account to view. The student is asked to add another account using SQL injection. Third, students are given the code for the simulation above. They are asked to modify the program to fix the security flaw caused by SQL injection.

Module Contents

- Assignment 1: SQL Injection Discussion
- Assignment 2: Break the Program
- Assignment 3: Fix the Program w/source code
- Answer key for each assignment
- Source code for SQLInjection class
- Source code for Menu class (used in SQLInjection program)
- Complete source code for solution to Assignment 3

Assignment 1: SQL Injection Discussion

SQL Injection

SQL injection is an attack techenique often used against data driven applications. This type of attack attempts to take advantage of poor programming practices to gain unauthorized access to databases. Attackers that gain access can alter or remove data from the database, affecting application operation. To protect against SQL injection attacks, developers must use best practices. The first practice for avoiding SQL injection is to *assume all input is evil*. This means do **not** trust input from users, files, or other programs. The second practice is to never use dynamic SQL, SQL that is constructed by concatenating SQL using user-provided values. The third practice is to never connect to the database with administrative level privileges that is, do not connect with root accounts. The fourth practice is do not store secrets in plain text. This means do not store user's passwords in plain text, encrypt the password before storing in the database. The fifth practice is that exceptions should display minimal information. Exceptions can display connection inforomation such as the database's username and password.

The following questions should be answered using the narrative above and using research conducted on the internet.

1. What are the five practices listed above?

2. What techniques could you use to implement the first practice?

3. What is dynamic SQL? Give an example using Java.

4. Explain the process of encrypting a user's password with a *salt*. Note: this password is stored in the database.

5. Why should you use a salt when encrypting passwords?

Assignment 2: Break the Program

In this assignment, you are given a Java program for a simple bank account. This program uses a database to store account information. You can load accounts and display account information using the command line interface. To load account information, use option 1. The prompt will display all acounts in the system. To load an account, you must input the account number. Once the account has been displayed, you will be returned to the main menu. The purpose of this assingment is to show how SQL injection can be used to alter a system's data. The database structure is given below for the account table.

Table	**Field**	**Data Type**	**Nulls**	**Unsign**	**P Key**	**Uniq**	**F Key**	**Comments**
account	id	int(11)		Y	Y			The account id
	name	varchar(45)						The account holder's name
	balance	double						The account balance

Your assignment will be to use SQL injection to modifty the database. Using SQL injection add an additional account with your name and an inital balance of $10,000 dollars. Verify that your account has been added to the system, then update your account to have $1,000,000 dollars.

1. At what point in the interface, can SQL injection be used?

2. What did you input to have the system insert your account?

3. What did you input to have the system update your account?

Assignment 3: Fix the Program

In this assignment, you will fix the source code from the previous assignment to protect the system against SQL injection. There may be serveral approaches to protect the system. Consider the first practice discussed in assignement 1: *assume all input is evil.* The only way to protect against all SQL injection is to use parameterized queries. Parameterized queires are prepared SQL statements that are built using parameters, not the user's input data. This technique should always be used when using user inputed data to retrieve data from a database.

Using Java's documentation on paramaterized query, alter the `getAccount` method to protect against SQL injection. http://docs.oracle.com/javase/6/docs/api/java/sql/PreparedStatement.html

1. Test your solution to ensure that it works.

2. Turn in a screen shot of the program with you trying to use SQL injection to add another account.

3. Turn in a screen shot of the program account list after your attempt to add another account. This should s

Answer Key

Assignment 1: SQL Injection Discussion

1. What are the five practices listed above?

   1) Assume all input is evil
   2) Never use dynamic SQL
   3) Never connect to database with admin level privileges
   4) Don't store secrets (password) in plain text
   5) Exceptions should display minimal information

2. What techniques could you use to implement the first practice?

   Sanatize user input at the application level.
   Use prepared statements for accessing the database.

3. What is dynamic SQL? Give an example using Java.

   Dynamic SQL is SQL that is created at runtime using user input.

4. Explain the process of encrypting a user's password with a *salt*.

   A salt is random data added to the user's password to increase security when encrypting the password to be stored.

5. Why should you use a salt when encrypting passwords?

   Adding a salt to the user's password before encrypting will protect agianst pre-compiled tables used to attack system. (Rainbow tables)

Assignment 2: Break the Program

1. At what point in the interface, can SQL injection be used?

   When the user is required to enter the account number.

2. What did you input to have the system insert your account?

   If selecting account with account number 1.

   ```
   1; insert into account values (5, 'John Doe', 10000);
   ```

3. What did you input to have the system update your account?

   ```
   1; update table set balance = 1000000000 where id = 5;
   ```

Assignment 3: Fix the Program

The following code is one possible solution. A full solution is given at the end of this module.

Alter the getAccount method to have the following code.

```
private static Account getAccount(String id) {
        Account account = null;

        PreparedStatement selectAccount = null;
        String selectAccountString = "select * from account where id = ?";

        try {
            Connection conn = getConnection();
            selectAccount = conn.prepareStatement(selectAccountString);
            selectAccount.setInt(1, Integer.parseInt(id)); // 1 references the first (and
                only) ? in the prepared stmt

            selectAccount.execute();
            ResultSet rs = selectAccount.getResultSet();
            if ( rs.next() ) {
                account = new Account(rs.getInt("id"), rs.getString("name"), rs.getInt(
                    "balance"));
            }
        } catch (Exception e) {
            System.out.println("Error getting the account: " + e.getMessage());
        }

}
```

SQL Injection Source Code

```
import java.util.*;
import java.sql.*;

class SQLInjection {

    public static void main(String[] args) {

        Scanner in = new Scanner(System.in);

        Menu menu = new Menu("Main Menu");
        menu.addOption(1, "View Account");
        menu.addOption(9, "Exit");

        while(true)
        {
            System.out.print("\n ");
            switch(menu.getOption())
            {
                case 1:
                {
                    System.out.printf("\n\n\t%-10s : %s", "Account Id", "Account Holder");
                    System.out.println("\n
                        ----------------------------------------");
                    for (Account account : getAccountList()) {
                        System.out.printf("\t%10s : %s\n", account.id, account.name);
                    }
                    System.out.print("\n\nSelect id for acount to be viewed: ");
                    String accountId = in.nextLine();

                    Account account = getAccount(accountId);
                    if ( account != null ) {
                        System.out.printf("\n\t%15s : %s", "Account Id", account.id);
                        System.out.printf("\n\t%15s : %s", "Account Holder", account.name);
                        System.out.printf("\n\t%15s : %s", "Balance", account.balance);
                    } else {
                        System.out.print("\nAccount with " + accountId + " does not exist");
                    }

                    System.out.print("\n\n\nPress ENTER to continue");
                    in.nextLine();

                    break;
                }

                case 9:
                {
                    System.out.println("Exiting");
                    System.exit(0);
```

```
                break;
            }
        }
    }
}

private static ArrayList<Account> getAccountList() {
    ArrayList<Account> accounts = new ArrayList<Account>();

    try {
        Connection conn = getConnection();
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("select id, name from account");
        while(rs.next()) {
            accounts.add(new Account(rs.getInt("id"), rs.getString("name")));
        }
    } catch (Exception e) {
        System.out.println("Error getting account list: " + e.getMessage());
    }

    return accounts;
}

private static Account getAccount(String id) {
    Account account = null;

    try {
        Connection conn = getConnection();
        Statement stmt = conn.createStatement();
        String strStmt = "select id, name, balance from account where id = " + id;
        stmt.execute(strStmt);
        ResultSet rs = stmt.getResultSet();
        if ( rs.next() ) {
            account = new Account(rs.getInt("id"), rs.getString("name"), rs.getInt("balance"))
                ;
        }
    } catch (Exception e) {
        System.out.println("Error getting the account: " + e.getMessage());
    }

    return account;
}

private static Connection getConnection() {
    Connection conn = null;

    try{
        Class.forName("com.mysql.jdbc.Driver");
        conn = DriverManager.getConnection("jdbc:mysql://localhost/injection?"
          + "user=root&password=password&allowMultiQueries=true");
    } catch (Exception e) {
```

```
            System.out.println("Error getting the connection: " + e.getMessage());
        }
        return conn;
    }
}

class Account {
    public int balance = 0;
    public int id;
    public String name;
    public Account(int i, String n, int b) {
        this.id = i;
        this.name = n;
        this.balance = b;
    }
    public Account(int i, String n) {
        this.id = i;
        this.name = n;
    }
}
```

Menu Class Source Code

```
class Menu
{
    class Option
    {
        private int id;
        private String desc;

        public Option(int id, String desc)
        {
            this.id = id;
            this.desc = desc;
        }
        public int getId()
        {
            return this.id;
        }
        public String getDesc()
        {
            return this.desc;
        }
    }

    private int _subMenuKey;
    private ArrayList<Option> _options;
    private String _title;

    public Menu(String title)
    {
        this._subMenus = new HashMap<Integer, Menu>();
        this._options = new ArrayList<Option>();

        this._subMenuKey = 1;
        this._title = title;
    }

    public void addOption(int id, String desc)
    {
        this._options.add(new Option(id, desc));
    }

    public void clearOptions()
    {
        this._options.clear();
    }

    public int getOption()
    {
        Scanner in = new Scanner(System.in);
```

```
        while(true)
        {
            System.out.println("\n--------------------------------------");
            System.out.println(" " + this._title + "\n");

            for (Option option : this._options) {
                System.out.println(" " + option.getId() + " - " + option.getDesc());
            }
            System.out.println("--------------------------------------");

            System.out.print("Select option: ");
            int optionInt;
            String optionStr = in.nextLine();

            //--- Attempt to convert the inputed string to an integer
            try {
                optionInt = Integer.parseInt(optionStr);
            } catch (NumberFormatException ex) {
                System.out.println("Error: It appears you did not input a option number");
                continue;
            }

            //--- Check if the option exists in our menu
            for (Option option : this._options) {
                if (option.getId() == optionInt) {
                    return optionInt;
                }
            }
            System.out.println("Error: Invalid option selected");
        }
    }
}


import java.util.*;
import java.sql.*;

class SQLInjection {

    public static void main(String[] args) {

        Scanner in = new Scanner(System.in);

        Menu menu = new Menu("Main Menu");
        menu.addOption(1, "View Account");
        menu.addOption(9, "Exit");

        while(true)
        {
            System.out.print("\n ");
            switch(menu.getOption())
            {
```

```
            case 1:
            {
                System.out.printf("\n\n\t%-10s : %s", "Account Id", "Account Holder");
                System.out.println("\n
                    ----------------------------------------");
                for (Account account : getAccountList()) {
                    System.out.printf("\t%10s : %s\n", account.id, account.name);
                }
                System.out.print("\n\nSelect id for acount to be viewed: ");
                String accountId = in.nextLine();

                Account account = getAccount(accountId);
                if ( account != null ) {
                    System.out.printf("\n\t%15s : %s", "Account Id", account.id);
                    System.out.printf("\n\t%15s : %s", "Account Holder", account.name);
                    System.out.printf("\n\t%15s : %s", "Balance", account.balance);
                } else {
                    System.out.print("\nAccount with " + accountId + " does not exist");
                }

                System.out.print("\n\n\nPress ENTER to continue");
                in.nextLine();

                break;
            }

            case 9:
            {
                System.out.println("Exiting");
                System.exit(0);
                break;
            }
        }
    }
}

private static ArrayList<Account> getAccountList() {
    ArrayList<Account> accounts = new ArrayList<Account>();

    try {
        Connection conn = getConnection();
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("select id, name from account");
        while(rs.next()) {
            accounts.add(new Account(rs.getInt("id"), rs.getString("name")));
        }
    } catch (Exception e) {
        System.out.println("Error getting account list: " + e.getMessage());
    }

    return accounts;
```

```
    }

    private static Account getAccount(String id) {
        Account account = null;

        PreparedStatement selectAccount = null;
        String selectAccountString = "select * from account where id = ?";

        try {
            Connection conn = getConnection();
            selectAccount = conn.prepareStatement(selectAccountString);
            selectAccount.setInt(1, Integer.parseInt(id)); // 1 references the first (and only) ? in the
                 prepared stmt

            selectAccount.execute();
            ResultSet rs = selectAccount.getResultSet();
            if ( rs.next() ) {
                account = new Account(rs.getInt("id"), rs.getString("name"), rs.getInt("balance"))
                    ;
            }
        } catch (Exception e) {
            System.out.println("Error getting the account: " + e.getMessage());
        }

        return account;
    }

    private static Connection getConnection() {
        Connection conn = null;

        try{
            Class.forName("com.mysql.jdbc.Driver");
            conn = DriverManager.getConnection("jdbc:mysql://localhost/injection?"
              + "user=root&password=password&allowMultiQueries=true");
        } catch (Exception e) {
            System.out.println("Error getting the connection: " + e.getMessage());
        }
        return conn;
    }
}

class Account {
    public int balance = 0;
    public int id;
    public String name;
    public Account(int i, String n, int b) {
        this.id = i;
        this.name = n;
        this.balance = b;
    }
    public Account(int i, String n) {
```

```
        this.id = i;
        this.name = n;
    }
}
```

Security Module: Input Validation

Description

Input validation is a technique for validating all input before using it. An application that assumes input from the user is valid opens itself up to security vulnerabilities. Issues with data consistency render a system's data useless. User input that queries a database may contain malicious SQL that can alter the database in undesirables ways.

Objective

The objectives of this security module are to illustrate and explain the importance of input validation and the consequences that not validating data can have on a your system.

Activities

There are two activities for this module. This module is targeted at a server-side development class that uses PHP. The first activity is a discussion assignment over input validation and methods that can be used to achieve input validation. Second, students are given a PHP script that simulates a sign-up form for an online account. Students are asked to test the form and add input validation logic.

Module Contents

- Assignment 1: Input Validation Discussion
- Assignment 2: Add Proper Validation
- Answer key for each assignment
- Source code for InputValidation script
- Complete source code for solution to Assignment 3

Assignment 1: Input Validation Discussion

Input Validation

Input validation is a term use to describe the process of validating all input before using it. An application that assumes input from the user is valid opens itself up to security vulnerabilities. Issues with data consistency render a system's data useless. User input that queries a database may contain malicious SQL that can alter the database in undesirables ways. Input validation starts by defining the data that will be collected from users. By defining the data, developers can validate the input data to ensure that it meets specifications.

The following is an example specification used for defining a sign-up form.

Field Name	**Data Type**	**Required**	**Constraints**
Username	Text	Yes	Consist of characters a-z, A-Z, and 0-9 Length between 6 and 20 (inclusive)
Email Address	Text	Yes	Must be a valid email Username@domain.com
First Name	Text	Yes	Consist of characters a-z and A-Z Length between 2 and 24 (inclusive)
Last Name	Text	Yes	Consist of characters a-z and A-Z Length between 2 and 24 (inclusive)
Age	Number	Yes	Number that must between 18 and 110 inclusive Must be 18 or over
Address Line 1	Text	Yes	Consist of characters a-z and A-Z Length between 2 and 48 (inclusive)
Address Line 2	Text	No	Consist of characters a-z and A-Z Length between 2 and 48 (inclusive)
City	Text	Yes	Consist of characters a-z and A-Z Length between 2 and 48 (inclusive)
US Zipcode	Integer	Yes	Consist of numbers Length equal to 5
State	Text	Yes	Must be one of the 50 states

Table 1: Sign-up form specification

The following questions can be answered using the above narrative, specification table, and online PHP documentation.

1. What is the benefit of creating a specification for form data?

2. What PHP function can be used to check the length of a variable?

3. What PHP function can be used to perform a regex match? What is the regex to check for a-z, A-Z, or 0-9?

4. What PHP function can be used to check if variable is a number?

5. What PHP array function can be used to check if a value is in an array?

Assignment 2: Add Proper Validation

In this assignment, you will be given access to a fake sign-up form that will require certain information. Once you submit the form, the information will be presented back to you as you entered it. The form does no validation checks on the data provided. Your assignment is to add proper validation logic and thoroughly test against the form specifications.

Field Name	**Data Type**	**Required**	**Constraints**
Username	Text	Yes	Consist of characters a-z, A-Z, and 0-9 Length between 6 and 20 (inclusive)
Email Address	Text	Yes	Must be a valid email Username@domain.com
First Name	Text	Yes	Consist of characters a-z and A-Z Length between 2 and 24 (inclusive)
Last Name	Text	Yes	Consist of characters a-z and A-Z Length between 2 and 24 (inclusive)
Age	Number	Yes	Number that must between 18 and 110 inclusive Must be 18 or over
Address Line 1	Text	Yes	Consist of characters a-z and A-Z Length between 2 and 48 (inclusive)
Address Line 2	Text	No	Consist of characters a-z and A-Z Length between 2 and 48 (inclusive)
City	Text	Yes	Consist of characters a-z and A-Z Length between 2 and 48 (inclusive)
US Zipcode	Integer	Yes	Consist of numbers Length equal to 5
State	Text	Yes	Must be one of the 50 states

Table 2: Sign-up form specification

1. Test your solution to ensure that it works.

2. Turn in a screen shot of the sign-up form with invalid data showing that the form accepts invalid data.

3. Turn in a screen shot of the sign-up form with valid data.

Answer Key

Assignment 1: Input Validation Discussion

1. What is the benefit of creating a specification for form data?

   A specification defines what validation needs to be performed on form fields.

2. What PHP function can be used to check the length of a variable?

   `strlen`

3. What PHP function can be used to perform a regex match? What is the regex to check for a-z, A-Z, or 0-9?

   `preg_match`

4. What PHP function can be used to check if variable is a number?

   `is_numeric`

5. What PHP array function can be used to check if a value is in an array?

   `is_array` **`^[a-zA-Z ]*$` (Notice: a space is after Z)**

Assignment 2: Add Proper Validation

A full solution is given at the end of this module.

Input Validation Source Code

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/
    DTD/xhtml1-strict.dtd">
<?php
    $states = array( "Alabama", "Alaska", "Arizona", "Arkansas", "California", "Colorado", "
        Connecticut", "Delaware", "District Of Columbia", "Florida", "Georgia", "Hawaii", "Idaho", "
        Illinois", "Indiana", "Iowa", "Kansas", "Kentucky", "Louisiana", "Maine", "Maryland", "
        Massachusetts", "Michigan", "Minnesota", "Mississippi", "Missouri", "Montana","Nebraska","
        Nevada","New Hampshire","New Jersey", "New Mexico", "New York", "North Carolina", "North
         Dakota", "Ohio", "Oklahoma", "Oregon", "Pennsylvania", "Rhode Island", "South Carolina", "
        South Dakota","Tennessee", "Texas", "Utah", "Vermont", "Virginia", "Washington", "West
        Virginia", "Wisconsin", "Wyoming" );

    $isPost = false;

    if ($_SERVER['REQUEST_METHOD'] === 'POST') {
        $isPost = true;

        //--- Get data
        $username  = $_POST['username'];
        $email   = $_POST['email'];
        $firstName  = $_POST['firstname'];
        $lastName  = $_POST['lastname'];
        $age   = $_POST['age'];
        $address1  = $_POST['address1'];
        $address2  = $_POST['address2'];
        $city   = $_POST['city'];
        $zipcode  = $_POST['zipcode'];
        $state   = $_POST['state'];
    }

?>
<html>
    <head>
        <title>Input Validation</title>
    </head>
    <body>
        <div>
            <!-- If the request is a POST then show data -->
            <?php if ($isPost) : ?>
                <table style="width: 400px; margin: 50px auto 0 auto;">
                    <tr>
                        <td>Username:</td>
                        <td><?php echo $username ?></td>
                    </tr>
                    <tr>
                        <td>Email Address:</td>
                        <td><?php echo $email ?></td>
                    </tr>
```

```
        <tr>
            <td>First Name:</td>
            <td><?php echo $firstName ?></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><?php echo $lastName ?></td>
        </tr>
        <tr>
            <td>Age:</td>
            <td><?php echo $age ?></td>
        </tr>
        <tr>
            <td>Address Line 1:</td>
            <td><?php echo $address1 ?></td>
        </tr>
        <tr>
            <td>Address Line 2:</td>
            <td><?php echo $address2 ?></td>
        </tr>
        <tr>
            <td>City:</td>
            <td><?php echo $city ?></td>
        </tr>
        <tr>
            <td>US Zipcode:</td>
            <td><?php echo $zipcode ?></td>
        </tr>
        <tr>
            <td>State:</td>
            <td><?php echo $state ?></td>
        </tr>
    </table>
<!-- Else show the form-->
<?php else: ?>
    <form action="" id="subscription-form" method="post" style="width: 400px;
        margin: 100px auto 0 auto;">
        <label for="username">Username</label><br />
        <input id="username" name="username" size="16" /><br /><br />

        <label for="email">Email Address</label><br />
        <input id="email" name="email" size="16" /><br /><br />

        <label for="firstname">First Name</label><br />
        <input id="firstname" name="firstname" size="16" /><br /><br />

        <label for="lastname">Last Name</label><br />
        <input id="lastname" name="lastname" size="16" /><br /><br />

        <label for="age">Age</label><br />
        <input id="age" name="age" size="8" /><br /><br />
```

```
                    <label for="address1">Address Line 1</label><br />
                    <input id="address1" name="address1" size="32" /><br /><br />

                    <label for="address2">Address Line 2</label><br />
                    <input id="address2" name="address2" size="32" /><br /><br />

                    <label for="city">City</label><br />
                    <input id="city" name="city" size="16" /><br /><br />

                    <label for="zipcode">US Zipcode</label><br />
                    <input id="zipcode" name="zipcode" size="16" /><br /><br />

                    <label for="state">State</label><br />
                    <select id="state" name="state">
                        <?php foreach ($states as $state) : ?>
                            <option value="<?php echo $state?>"><?php echo $state?></option>
                        <?php endforeach; ?>
                    </select><br /><br />

                    <input class="submit" type="submit" value="Submit" />
                </form>
            <?php endif; ?>
        </div>
    </body>
</html>
```

Input Validation Solution Source Code

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/
    DTD/xhtml1-strict.dtd">
<?php
    $states = array( "Alabama", "Alaska", "Arizona", "Arkansas", "California", "Colorado", "
        Connecticut",
            "Delaware", "District Of Columbia", "Florida", "Georgia", "Hawaii", "Idaho", "Illinois",
            "Indiana", "Iowa", "Kansas", "Kentucky", "Louisiana", "Maine", "Maryland", "
                Massachusetts",
            "Michigan", "Minnesota", "Mississippi", "Missouri", "Montana", "Nebraska", "Nevada",
            "New Hampshire", "New Jersey", "New Mexico", "New York", "North Carolina", "North
                Dakota",
            "Ohio", "Oklahoma", "Oregon", "Pennsylvania", "Rhode Island", "South Carolina", "South
                Dakota",
            "Tennessee", "Texas", "Utah", "Vermont", "Virginia", "Washington", "West Virginia",
            "Wisconsin", "Wyoming" );

    //--- Set default variables
    $username = "";
    $email  = "";
    $firstName = "";
    $lastName = "";
    $age  = "";
    $address1 = "";
    $address2 = "";
    $city  = "";
    $zipcode = "";
    $state  = "";
    $isPost = false;

    //--- Error messages
    $errorMessages = array();

    if ($_SERVER['REQUEST_METHOD'] === 'POST') {
        $isPost = true;

        //--- Get data
        $username = $_POST['username'];
        $email  = $_POST['email'];
        $firstName = $_POST['firstname'];
        $lastName = $_POST['lastname'];
        $age  = $_POST['age'];
        $address1 = $_POST['address1'];
        $address2 = $_POST['address2'];
        $city  = $_POST['city'];
        $zipcode = $_POST['zipcode'];
        $state  = $_POST['state'];

        //--- Validate username (is string; between 6 and 20; contains only a-z, A-Z, or 0-9
```

```
if ($username != null) {
    if (!is_string($username)) {
        $errorMessages[] = 'Username must be a string';
    }
    if (strlen($username) < 6 || strlen($username) > 20) {
        $errorMessages[] = 'Username must be between 8 and 20 characters long (inclusive)';
    }
    if (!preg_match('/^[a-zA-Z0-9]*$/', $username)) {
        $errorMessages[] = 'Username can only contain a-z, A-Z, or 0-9';
    }
} else {
    $errorMessages[] = 'Username is a required field';
}

//--- Validate email (is string; valid email address)
if ($email != null) {
    if (!is_string($email)) {
        $errorMessages[] = 'Email must be a string';
    }
    if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
        $errorMessages[] = 'Email must be a valid email address';
    }
} else {
    $errorMessages[] = 'Email Address is a required field';
}

//--- Validate firstName (is string; between 2 and 24; contains only a-z, A-Z
if ($firstName != null) {
    if (!is_string($firstName)) {
        $errorMessages[] = 'First Name must be a string';
    }
    if (strlen($firstName) < 2 || strlen($firstName) > 24) {
        $errorMessages[] = 'First Name must be between 2 and 24 characters long (inclusive)';
    }
    if (!preg_match('/^[a-zA-Z]*$/', $firstName)) {
        $errorMessages[] = 'First Name can only contain a-z or A-Z';
    }
} else {
    $errorMessages[] = 'First Name is a required field';
}

//--- Validate lastName (is string; between 2 and 24; contains only a-z, A-Z)
if ($lastName != null) {
    if (!is_string($lastName)) {
        $errorMessages[] = 'Last Name must be a string';
    }
    if (strlen($lastName) < 2 || strlen($lastName) > 24) {
        $errorMessages[] = 'Last Name must be between 2 and 24 characters long (inclusive)';
    }
    if (!preg_match('/^[a-zA-Z]*$/', $lastName)) {
        $errorMessages[] = 'Last Name can only contain a-z or A-Z';
```

```
    }
} else {
    $errorMessages[] = 'Last Name is a required field';
}

//--- Validate age (is number; between 18 and 110)
if ($age != null) {
    if (!is_numeric($age)) {
        $errorMessages[] = 'Age must be a number';
    }
    if ($age < 18 || $age > 110) {
        $errorMessages[] = 'Age must be between 18 and 110 characters long (inclusive)';
    }
} else {
    $errorMessages[] = 'Age is a required field';
}

//--- Validate address line 1 (is string, between 2 and 48; contains only a-z, A-Z, or 0-9)
if ($address1 != null) {
    if (!is_string($address1)) {
        $errorMessages[] = 'Address Line 1 must be a string';
    }
    if (strlen($address1) < 2 || strlen($address1) > 48) {
        $errorMessages[] = 'Address Line 1 must be between 2 and 48 characters long (inclusive)
            ';
    }
    if (!preg_match('/^[a-zA-Z0-9 ]*$/', $address1)) {
        $errorMessages[] = 'Address Line 1 can only contain a-z, A-Z, or 0-9';
    }
} else {
    $errorMessages[] = 'Address Line 1 is a required field';
}

//--- Validate address line 2 (is string, between 2 and 48; contains only a-z, A-Z, or 0-9)
//--- not a required field
if ($address2 != null) {
    if (!is_string($address2)) {
        $errorMessages[] = 'Address Line 2 must be a string';
    }
    if (strlen($address1) < 2 || strlen($address2) > 48) {
        $errorMessages[] = 'Address Line 2 must be between 2 and 48 characters long (inclusive)
            ';
    }
    if (!preg_match('/^[a-zA-Z0-9 ]*$/', $address2)) {
        $errorMessages[] = 'Address Line 2 can only contain a-z, A-Z, or 0-9';
    }
}

//--- Validate city (is string, between 2 and 48; contains only a-z or A-Z)
if ($city != null) {
    if (!is_string($city)) {
```

```
            $errorMessages[] = 'City must be a string';
        }
        if (strlen($city) < 2 || strlen($city) > 48) {
            $errorMessages[] = 'City must be between 2 and 48 characters long (inclusive)';
        }
        if (!preg_match('/^[a-zA-Z ]*$/', $city)) {
            $errorMessages[] = 'City can only contain a-z, A-Z, or 0-9';
        }
    } else {
        $errorMessages[] = 'City is a required field';
    }

    //--- Validate zipcode (is number, exactly 5 digits long);
    if ($zipcode != null) {
        if (!is_numeric($zipcode)) {
            $errorMessages[] = 'Zipcode must be a number';
        }
        if (strlen($zipcode) != 5) {
            $errorMessages[] = 'Zipcode must be exactly 5 digits long';
        }
    } else {
        $errorMessages[] = 'Zipcode is a required field';
    }

    //--- Validate state (is number, exactly 5 digits long;
    if ($state != null) {
        if (!in_array($state, $states)) {
            $errorMessages[] = 'State must be one of the 50 states or DC';
        }
    } else {
        $errorMessages[] = 'State is a required field';
    }

}

?>
<html>
    <head>
        <title>Input Validation</title>
    </head>
    <body>
        <div>
            <!-- If the request is a POST then show data -->
            <?php if ($isPost && !sizeof($errorMessages)) : ?>
                <table style="width: 400px; margin: 50px auto 0 auto;">
                    <tr>
                        <td>Username:</td>
                        <td><?php echo $username ?></td>
                    </tr>
                    <tr>
                        <td>Email Address:</td>
```

```
            <td><?php echo $email ?></td>
        </tr>
        <tr>
            <td>First Name:</td>
            <td><?php echo $firstName ?></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><?php echo $lastName ?></td>
        </tr>
        <tr>
            <td>Age:</td>
            <td><?php echo $age ?></td>
        </tr>
        <tr>
            <td>Address Line 1:</td>
            <td><?php echo $address1 ?></td>
        </tr>
        <tr>
            <td>Address Line 2:</td>
            <td><?php echo $address2 ?></td>
        </tr>
        <tr>
            <td>City:</td>
            <td><?php echo $city ?></td>
        </tr>
        <tr>
            <td>US Zipcode:</td>
            <td><?php echo $zipcode ?></td>
        </tr>
        <tr>
            <td>State:</td>
            <td><?php echo $state ?></td>
        </tr>
    </table>
<!-- Else show the form-->
<?php else: ?>
    <?php if ($errorMessages) : ?>
        <ul style="width: 400px; margin: 10px auto 0 auto; color: red;">
        <?php foreach ($errorMessages as $message) : ?>
            <li><?php echo $message; ?></li>
        <?php endforeach; ?>
        </ul>
    <?php endif; ?>
    <form action="" id="subscription-form" method="post" style="width: 400px;
        margin: 10px auto 0 auto;">
        <label for="username">Username</label><br />
        <input id="username" name="username" size="16" value="<?php echo
            $username;?>"/><br /><br />

        <label for="email">Email Address</label><br />
```

```
                    <input id="email" name="email" size="16" value="<?php echo $email;?>"/><
                        br /><br />

                    <label for="firstname">First Name</label><br />
                    <input id="firstname" name="firstname" size="16" value="<?php echo
                        $firstName;?>"/><br /><br />

                    <label for="lastname">Last Name</label><br />
                    <input id="lastname" name="lastname" size="16" value="<?php echo
                        $lastName;?>"/><br /><br />

                    <label for="age">Age</label><br />
                    <input id="age" name="age" size="8" value="<?php echo $age;?>"/><br /><
                        br />

                    <label for="address1">Address Line 1</label><br />
                    <input id="address1" name="address1" size="32" value="<?php echo $address1
                        ;?>"/><br /><br />

                    <label for="address2">Address Line 2</label><br />
                    <input id="address2" name="address2" size="32" value="<?php echo $address2
                        ;?>"/><br /><br />

                    <label for="city">City</label><br />
                    <input id="city" name="city" size="16" value="<?php echo $city;?>"/><br
                        /><br />

                    <label for="zipcode">US Zipcode</label><br />
                    <input id="zipcode" name="zipcode" size="16" value="<?php echo $zipcode;?>
                        "/><br /><br />

                    <label for="state">State</label><br />
                    <select id="state" name="state">
                        <?php foreach ($states as $state) : ?>
                            <option value="<?php echo $state?>"><?php echo $state?></option>
                        <?php endforeach; ?>
                    </select><br /><br />

                    <input class="submit" type="submit" value="Submit" />
                </form>
            <?php endif; ?>
        </div>
    </body>
</html>
```

BIBLIOGRAPHY

[1] C. Alexander, S. Ishikawa, and M. Silverstein. A Pattern Language: Towns, Buildings, Construction. Center for Environmental Structure Series. Oxford University Press, 1977. URL: http://books.google.com/books?id=hwAHmktpk5IC.

[2] Shiva Azadegan, M. Lavine, Michael O'Leary, Alexander L. Wijesinha, and Marius Zimand. An undergraduate track in computer security. In ITiCSE, pages 207–210, 2003.

[3] H. Bagheri and S.-H. Mirian-Hosseinabadi. Injecting security as aspectable nfr into software architecture. In Software Engineering Conference, 2007. APSEC 2007. 14th Asia-Pacific, pages 310 –317, dec. 2007.

[4] Mario Barbacci, Mark H. Klein, Thomas A. Longstaff, and Charles B Weinstock. Quality attributes. Technical report, Software Engineering Institute, Carnegie Mellon, December 2005. URL: hhttp://www.sei.cmu.edu/reports/95tr021.pdf.

[5] Kent Beck and Ward Cunningham. Using pattern languages for object oriented programs. In Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 1987.

[6] Grady Booch. Software archeology and the handbook of software architecture. In Rainer Gimnich, Uwe Kaiser, Jochen Quante, and Andreas Winter, editors, Workshop Software Reengineering, volume 126 of LNI, pages 5–6. GI, 2008. URL: http://dblp.uni-trier.de/db/conf/wsr/wsr2008.html#Booch08.

[7] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. Past, present, and future trends in software patterns. IEEE Softw., 24(4):31–37, July 2007. URL: http://dx.doi.org/10.1109/MS.2007.115.

[8] L. Chung, B.A. Nixon, and E. Yu. Non-Functional Requirements in Software Engineering. The Kluwer International Series in Software Engineering. Kluwer Academic, 2000. URL: http://books.google.com/books?id=IgV_nRpf5tUC.

[9] James O. Coplien. Software patterns. URL: http://www.hillside.net/component/content/article/50-patterns/222-design-pattern-definition.

[10] Chad Dougherty, Kirk Sayre, Robert C. Seacord, David Svoboda, and Kazuya Togashi. Secure design patterns. Technical report, Software Engineering Institute, Carnegie Mellon, October 2009. URL: http://www.sei.cmu.edu/reports/09tr010.pdf.

[11] E. Gamma, R. Johnson, J. Vlissides, and R. Helm. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995. URL: http://books.google.com/books?id=iyIvGGp2550C.

[12] Alejandro Gervasio. Validating user input with the strategy pattern, March 2007. URL: http://www.devshed.com/c/a/PHP/Validating-User-Input-with-the-Strategy-Pattern/.

[13] Info Sec Island. Siemans patches scada system vulneerabilities, June 2011. URL: https://www.infosecisland.com/blogview/14417-Siemens-Patches-SCADA-System-Vulnerabilities.html.

[14] R McMillian. Siemens fixes industrial flaws found by hacker, June 2011. URL: http://www.computerworld.com/s/article/9217547/Siemens_fixes_industrial_flaws_found_by_hacker.

[15] James David Moody. Categorizing non-functional requirements using a hierarchy in uml, 2003.

[16] M. Schumacher. Security patterns: integrating security and systems engineering. Wiley series in software design patterns. John Wiley & Sons, 2006. URL: http://books.google.com/books?id=gtpQAAAAMAAJ.

[17] Mathew J. Schwartz. Sony data breach cleanup to cost 171 million, May 2011. URL: http://www.informationweek.com/security/attacks/sony-data-breach-cleanup-to-cost-171-mil/229625379.

[18] P Seybold. Update on playstation network and qriocity, April 2011. URL: http://blog.us.playstation.com/2011/04/26/update-on-playstation-network-and-qriocity/.

[19] Blair Taylor and Shiva Azadegan. Threading secure coding principles and risk analysis into the undergraduate computer science and information systems curriculum. In Proceedings of the 3rd annual conference on Information security curriculum development, InfoSecCD '06, pages 24–29, New York, NY, USA, 2006. ACM. URL: http://doi.acm.org/10.1145/1231047.1231053.

[20] Blair Taylor and Shiva Azadegan. Moving beyond security tracks: integrating security in cs0 and cs1. In Proceedings of the 39th SIGCSE technical symposium on Computer science education, SIGCSE '08, pages 320–324, New York, NY, USA, 2008. ACM.

[21] John Viega and Gary McGraw. Building Secure Software - How to Avoid Security Problems the Right Way. Addison-Wesley, September 2002.

[22] Joseph Yoder and Jeffrey Barcalow. Architectural patterns for enabling application security. In Fourth Conference on Pattern Languages of Programs, 1998.

VITA

JEREMIAH Y. DANGLER

Personal Data:	Date of Birth: May 26, 1981. Place of Birth: Cartersville, Georgia. Email: jdangler@gmail.com
Education:	A.S. Computer Networking and Service Technology, Dalton State College, 2004. A.S. Drafting and Design Technology, Dalton State College, 2004. B.S. Computer Science, East Tennessee State University, 2011. M.S. Computer Science, East Tennessee State University, 2013.
Professional Experience:	IT/IS Coordinator, DA Technologies; Atlanta, GA. 2000 - 2010 Developer, Emerging Technology Center: Johnson City, TN. 2008 - 2012 Research Assistant, Emerging Technology Center: Johnson City, TN. 2011 – 2013
Academic Activities:	Webmaster, Association of Computing Machinery 2011 - 2012 President, Association of Computing Machinery 2012 - 2013 President, Upsilon Pi Epsilon 2011 – 2013
Honors and Awards:	Dean’s List 2008 - 2012 Outstanding Senior in Computer Science 2011 East Tennessee State University Unsung Hero Award 2011 Outstanding Graduate Student in Computer Science 2013