Chapter 15
An Integrated Secure Software Engineering Approach for Functional, Collaborative, and Information Concerns

J. A. Pavlich-Mariscal
Pontificia Universidad Javeriana, Colombia

S. Berhe
University of Connecticut, USA

A. De la Rosa Algarín
University of Connecticut, USA

S. Demurjian
University of Connecticut, USA

ABSTRACT
This chapter explores a secure software engineering approach that spans functional (object-oriented), collaborative (sharing), and information (Web modeling and exchange) concerns in support of role-based (RBAC), discretionary (DAC), and mandatory (MAC) access control. By extending UML with security diagrams for RBAC, DAC, and MAC, we are able to design an application with all of its concerns, and not defer security to a later time in the design process that could have significant impact and require potentially wide-ranging changes to a nearly completed design. Through its early inclusion in the software design process, security concerns can be part of the application design process, providing separate abstractions for security via new UML diagrams. From these new UML diagrams, it is then possible to generate security policies and enforcement code for RBAC, DAC, and MAC, which separates security from the application. This modeling and generation allows security changes to have less of an impact on an application. The end result is a secure software engineering approach within a UML context that is capable of modeling an application’s functional, collaborative, and information concerns. This is explored in this chapter.

DOI: 10.4018/978-1-4666-6026-7.ch015
1 INTRODUCTION

The software development process has had significant improvements of the past forty plus years, from the introduction of the waterfall model (Winston, 1970) to the iterative model (Larman and Basili, 2002) in the late 70s to the spiral model (Boehm, 1986) in the mid-1980s to the unified process model (Scott, 2001) to the agile development lifecycle (Craig, 2003) in the early 21st century. Despite this progress, there remain many challenges when one attempts to design and develop large-scale applications, where there are a myriad of concerns such as user interfaces, server functionality, database support, logging and historical tracking, and secure information modeling, access, and enforcement. Rather than separation, there is often an entanglement of these different concerns, e.g., in an object-oriented application, code to read/write the database can be spread across multiple classes even if the database is abstracted via Hibernate. Also consider that security can be realized across the entire application, with security checks and enforcement at the GUI level, the server level, the database level, the network communications level, etc. All of these different concerns end up being tangled with one another, and spread out across the application’s varied components. As a result, the traceability of security in terms of an application’s functional, collaborative, and information concerns cannot be easily isolated; in such a situation, changes to the security policy often requires code-level alternations which are not acceptable in practice.

The intent of this chapter is to elevate security to a primary and early priority in the software development process to provide a secure engineering approach that encompasses functional, collaborative, and information concerns. To place this into a proper perspective, Figure 1 conceptualizes a secure software engineering approach for functional, collaborative, and information concerns via UML to visually model access control security. Over the past five years, our focus has been on extending UML with new diagrams that supports secure software engineering for role-based access control (RBAC) (Ferraiolo, et al., 2001), discretionary access control (DAC) (DoD, 1985), and mandatory access control (MAC) (Bell & LaPadula, 1976). In this chapter, we bring together our work for secure software engineering in three areas. First, from a functional perspective that focuses on object-oriented design, we have developed a framework of composable security features that preserves separation of security concerns from models to code through the extension of UML with new diagrams for RBAC, DAC, and MAC with the automatic generation of enforcement code in AspectJ that allowed the security definitions to be separated (untangled) from the code (Pavlich-Mariscal, 2005, 2010a, 2010b). Second, from a collaboration perspective, we have developed a framework for secure, obligated, coordinated, and dynamic collaboration that extends the RBAC to allow for the definition and enforcement of security for collaborative RBAC applicable to situations such as medical care where physicians from different specialties need to collaborate with one another to treat a patient in an effective and timely manner (Berhe, 2009, 2010, 2012); this work also defines new UML diagrams based on our functional work (Pavlich-Mariscal, 2005, 2010a, 2010b). Third, from an information perspective, we have defined and developed an XML security framework (De la Rosa Algarín, 2012, 2013a, 2013b) with new XML oriented UML diagrams that integrates RBAC, DAC, and MAC to further extend (Pavlich-Mariscal, 2005, 2010a, 2010b) by allowing the definition of security policies for XML for sharing and exchange of information in a secure manner via XACML. Our combined work promotes security as an integral part of a secure software engineering approach, while tracking software quality assurance in terms of the consistency of the security and non-security requirements.
In this chapter, we will unify the aforementioned three security efforts as shown in Figure 1 to provide a secure software engineering process for functional, collaborative, and information modeling and design. From a functional perspective, our work (Pavlich-Mariscal, 2005, 2010a, 2010b) extended UML to represent RBAC, DAC, and MAC (see upper portion of Figure 2) via the introduction of the Role Slice Diagram, the User Diagram, the Delegation Diagram, and MAC extensions coupled with a Secure Subsystem Diagram (middle right hand side of Figure 2). The Secure Subsystem Diagram denotes the subset of an application’s overall classes and methods that are restricted and require permissions to be in place for authorized users. The Role Slice Diagram denotes RBAC policies, providing the role slice, a stereotyped package that represents the permissions assigned to a role. A role slice uses method-based permissions to allow or deny users to access specific operations, regardless of the object to which it is applied. The Delegation Diagram can be utilized to represent all of the rules of delegation between roles. This diagram provides the delegation slice, a stereotyped package that contains all of the roles that a user can delegate authority to another user, who may also be allowed to further delegate the role. The User Diagram uses stereotyped packages to denote users and stereotyped dependency relations to represent user-role assignments. MAC extensions enhance the previous three diagrams with sensitivity levels (e.g., confidential, secret, top secret) and their ordering relations to indicate classifications of methods, clearances of role slices, and, implicitly, to declare access constraints based in the relation between classifications and clearances. From an enforcement perspective, once defined, the diagrams are utilized to generate aspect-oriented enforcement code in AspectJ (bottom portion of Figure 2) that is able to verify, at runtime, whether the active user has a role with permissions over the protected method and grants or denies access accordingly. The end result is that aspects can effectively modularize access control concerns and enhance traceability from design to code.

The second part of our work (Berhe, 2009, 2010, 2012) (lower middle left of Figure 2) has focused on the extension of RBAC to define collaboration and sharing capabilities across a workflow. Collaborative computing has emerged in many domains, with users interacting with one...
another towards some common goal. For example, in a health care setting, a patient’s many providers (e.g., internist, cardiologist, physical therapist, etc.) need to interact with one another against a common set of data (patient’s medical record). Unlike traditional security that defines separation of duty and mutual exclusion to prohibit what users can do, in a collaborative setting, the key is on defining when and how users collaborate. Thus, we have extended RBAC with a set of UML diagrams for collaboration on duty and adaptive workflow (COD/AWF) that interacts with our functional extensions in the top of Figure 2 (Pavlich-Mariscal, 2005, 2010a, 2010b). This includes the definition of four new UML diagrams: the extended Role Slice Diagram which defines the roles and privileges for each user within each collaboration step; the Team Slice Diagram which defines the team members and their participation in the various collaboration steps; the Workflow Slice Diagram which defines the steps and connections among them for a given team and specific collaboration; and, the Obligation Slice Diagram which defines the required permissions and participations for a particular collaboration. In addition, we provide the mapping of these new UML based collaboration design-time diagrams to actual machine-readable code-based policies for runtime enforcement.

The third part of our work (De la Rosa Algarín, 2012, 2013a, 2013b) (middle right of Figure 2) has emphasized the control of information created
by one application to be shared and/or exchanged with other applications. One dominant approach for information exchange is the eXtensible Markup Language (XML). Defining XML schemas has become an integral part of the application development process to handle exchange form database to server, from server to end user, among different databases, etc. In support of information-based security, we have extended UML and the work of (Pavlich-Mariscal, 2005, 2010a, 2010b) with two new diagrams: the XML Schema Class Diagram which models a defined XML schema as a UML class diagram via the use of a UML Profile; and, the XML Role Slice Diagram, which models the RBAC requirements of a specific application with respect to role and elements of the original XML schema. These two constructs allow us to automatically generate enforcement policies with the eXtensible Access Control Markup Language (XACML) via a mapping process between the XML Role-Slice Diagram and the XACML schema elements. In turn, these enforcement policies can be readily deployed into any security architecture that utilizes the XACML specification’s processing model. With these two new UML diagrams augmented with the policy mapping process, a software engineer can consider and produce security enforcement code that targets information content by modeling the original XML schema (producing the XML Schema Class Diagram), augmenting it with security features with respect to the different roles and permissions (producing the XML Role Slice Diagram), and then automatically creating an enforcement policy with the mapping process (XACML).

To demonstrate the feasibility of the secure software engineering approach of this chapter, we leverage the health care domain that has a need to integrate multiple health information technology systems such as an Electronic Health Record (EHR) to store patient medical data, a personal health record to store patient data controlled by patient, a patient portal which is a practice management system for patients to communicate for appointments, refills, and referrals, and, ancillary systems such imaging, laboratory, pharmacy, etc. The United States Health Insurance Portability and Accountability Act (HIPAA) provides a set of security guidelines in the usage, transmission, and sharing of protected health information (PHI); in e-commerce, there would be a need to protect personally identifiable information (PII) including names, addresses, accounts, credit card numbers, etc. Realizing PHI and PII must be part of a secure software engineering process. Further, the many XML standards (e.g., clinical document architecture (CDA) and the Continuity of Care Record (CCR)) for storage of administrative, patient demographics, and clinical data are used to support information exchange among hospitals, clinics, physician practices, and laboratories. As CDA and CCR XML documents are circulated among various systems and made available to particular users with specific needs (e.g., using a function via an API call), representing a collaboration via a shared application, or requiring information via an XML or database repository). In such a situation, secure software engineering must consider all of the APIs, XML repositories, and databases. This requires API level (functional), document-level (information), and interaction level (collaboration) access control in order to authorize users at different times based on criteria that include, but are not limited to, a user’s role, time and value constraints on data, collaboration for sharing data, delegation of authority as privileges are passed among authorized users, etc.

The remainder of this chapter has six sections. Section 2 provides background on secure software engineering from functional, collaborative, and information perspectives. Section 3 introduces the work of (Pavlich-Mariscal, 2005, 2010a, 2010b) that extends UML with new diagrams to allow security to be modeled as a new separate concern for functional requirements. Section 4 details the work of (Berhe, 2009, 2010, 2012) that extends RBAC with new UML diagrams for collaboration on duty and adaptive workflow
An Integrated Secure Software Engineering Approach

for sharing requirements of an application. Section 5 explores the work of (De la Rosa Algarín, 2012, 2013a, 2013b) for an XML framework of security that extends UML with XML-schema diagrams to handle the information requirements. Each of the Sections 3, 4, and 5 organizes the presentation by discussing the UML extensions for modeling security followed by the transition to the generation of security enforcement code or policies, which includes a case study using a medication management mobile application that has: a patient version to maintain information on medications, OTCs, supplements, etc., and the ability to authorize access to providers; and, a provider version to be able to view and comment on authorized patients and their medications. Using the material in Sections 3, 4, and 5, Section 6 presents a secure software engineering approach for the definition and enforcement of access control (RBAC, DAC, and MAC) for functional, collaborative, and information concerns integrated with a UML design; to provide an additional context for the utility of our work, we utilize a big-data application example that involves a facet of law enforcement, namely, the collection of data on motor vehicle crashes. Section 7 presents future trends in terms of emerging platforms for health information technologies that have the potential to further impact the secure software engineering process. Section 8 concludes the chapter.

2 BACKGROUND

This section provides background on secure software engineering and security for functional, collaborative, and information concerns. In the functional security area, there have been many research efforts that involve UML. SecureUML (Basin et al., 2006) is a modeling language to design security in distributed systems. This language is based in UML, extending its semantics and notation to support RBAC, and certain authorization constraints. The focus of this approach is to utilize UML to specify access control as part of the main design of the application and then automatically generate access control infrastructures based on the design models. The approach defines a meta-model for SecureUML and details a methodology to integrate it into several design modeling languages. UMLSec, described in (Juerjens, 2005) and improved by (Zisman 2007) and (Popp et al., 2003), is an extension to UML that defines several new stereotypes towards formal security verification of elements such as: fair exchange to avoid cheating for any party in a 2-party transaction; secrecy and confidentiality of information (accessible only to the intended people); secure information flow to avoid partial leaking of sensitive information; and, secure communication links like encryption. AuthUML (Alghathbar 2007) models RBAC policies using use cases and Horn clauses to represent the access control information and to check its consistency. The approach of (Pavlich-Mariscal et al., 2010a) includes the definition of several new UML diagrams to represent different access control concerns (RBAC, MAC, and DAC), and a set of features that represent small subsets of an access control model. These features can then be composed to create custom access control policies.

In the collaborative security area, research has occurred in many areas. In terms of access control and collaboration, in (Tolone et al., 2003), a set of eight criteria (complexity, understandability, ease of use, applicability, groups of users, policy specification, policy enforcement, and granularity) critical for a collaborative environment are presented. The eight characteristics and their support are evaluated against seven access control models (Matrix, RBAC, TBAC, TMAC, C-TMAC, SAC, and Context-AW). This work demonstrates that collaboration capabilities are not primary goals in access control models. As given, the seven access control models do not support an integrated model for coordinated, obligated, secure, and team-based collaboration. Another related area is a distributed secure interoperability framework for collaboration environments (Sachpazidis 2008).
This framework presents a multi-system architecture in which different stakeholders with different privileges collaborate with one another towards optimizing monitoring prescription intake. With regard to collaboration, workflow, and security, the work of (Shehab et al., 2005) addressed security services that support inter-organizational collaborative enterprises, which may span multiple organizations. This work presents a framework for mediator-free collaboration. Similarly as in (Sachpazidis, 2008), this work focuses on inter-system collaboration, while our work focuses on early integration of collaboration requirements into the software engineering process. The work of (Kang et al., 2001) concentrates on workflows that are addressed from an access control perspective. This is an important aspect of our effort (Berhe, 2009, 2010, 2012), where workflow is also represented as collaboration steps with access control addressed at each step and for the overall workflow. The work in (Sun et al., 2005) proposes a model that integrates RBAC into workflows. In their approach, permissions, roles, cardinality, ancestors (pre-obligations), and a status value are assigned to activities (collaboration steps).

Finally, in the informational security area, one approach based on RBAC and UML is SecureUML (Lodderstedt et al., 2002) that combines a graphical notation for RBAC with constraints, where policies can be expressed using RBAC permissions with complex security requirements done with a combination of authorization constraints. A later effort (Mouelhi et al., 2008) presents a model-driven security approach for designers to set security requirements along with the system models to automatically generate an access control infrastructure. The approach combines UML with a security modeling language defining a set of modeling transformations; the former produces infrastructures for JavaBeans, and the latter can generate secure infrastructures for web applications. (Basin et al., 2006) utilizes the model-driven architecture paradigm to achieve security for e-government scenarios with inter-collaboration/communication. This is achieved by describing security requirements at a high-level (models), with relevant “security artifacts” being automatically generated for target architectures, removing the otherwise present learning curve in specifying security requirements by domain experts with no technical know-how. In the approach presented in (De la Rosa Algarín et al., 2012, 2013a, 2013b), UML is leveraged to provide a secure information engineering approach for XML schemas and documents. By extending UML with new XML diagrams, an enforcement security policy in XACML can be generated and deployed for access control purposes (not unlike automatic code generation from UML class diagrams). By doing this, the approach scales to scenarios that involve a high volume of XML documents validated against a common schema.

### 3 FUNCTIONAL SECURITY

To better understand the perspective of our security solutions, it is important to review the five essential security dimensions in our previous work (Pavlich-Mariscal, Demurjian, & Michel, 2010a): access control models, visual notation, separation of concerns, traceability, and customizability. The access control model dimension encompasses mandatory access control (MAC) (Bell & LaPadula, 1976), discretionary access control (DAC) (DoD, 1985), and role-based access control (RBAC) (Ferraiolo, et al., 2001). MAC defines the objects that need protection and the subjects who are the individuals or other systems that require access to the information in the objects. Objects have classifications (e.g., Confidential, Secret, Top-Secret, etc.) that indicate the sensitivity level of the information they contain. Subjects have clearances that determine the set of objects to which they can access, based on the relationship between clearances and classifications. DAC specifies access to objects, based on the subjects to which the objects belong. A subject owns a set of objects. Owned objects...
can be accessed and modified discretionally by the subject. In addition, a subject can directly or indirectly pass that permission to another subject, unless it is restricted by a MAC policy, which is delegation (Liebrand et al., 2003). Our approach supports both delegation of authority, the ability to directly pass the permission to another subject, and pass-on delegation authority, the ability to allow to a second subject to further delegate a delegated authority. RBAC has a clear correspondence to the structure of information in organizations. Unlike MAC, which focuses on data, RBAC focuses on users. The essential element in RBAC is the role that represents a set of tasks that a user may perform in an organization, with the premise that the same role could be assigned to different users. While DAC assigns permissions directly to users (subjects), RBAC assigns permissions to roles. A role is assigned to a user, who obtains all of the permissions associated with the role (Ferraiolo, et al., 2001).

The second dimension, visual notation, provides a means to represent access control policies at a higher conceptual level. A visual notation facilitates the comprehension and development of an access control policy over time, since it can convey the essential information to designers. In this regard, widespread notations, such as UML do not provide explicit support for access control models. Separation of concerns (Parnas, 1972), our third dimension, is important not only for access control, but for the evolution of every concern in a software system. The premise is that in order to better understand and develop different concerns, including security, each concern should be adequately modularized. However, security is a concern that tends to be scattered throughout the application (e.g., access control code is usually included at each place where protection is required, such as at the beginning of methods) and tangled with other concerns (e.g., access control is usually tangled with the code that is being protected, which represents different concerns).

Our fourth dimension, traceability, is the ability to link requirements to design artifacts to the code parts that realize each requirement (Sommerville, 2006). In our work, the focus is on traceability from access control models to code. Traceability is directly influenced by separation of concerns. The better the separation of concerns, the better the traceability, since it is easier to identify the design and code artifacts associated to the same requirement. Our fifth and final dimension, customizability, is the ability to combine a set of primitive components to achieve different results (Pavlich-Mariscal, Demurjian, & Michel, 2010a). In our work, customizability is the ability to combine different access control models to satisfy different security requirements. Customizability is also facilitated by an adequate separation of concerns, since the boundaries of each primitive can be better defined, thus facilitating their composition. All of the above dimensions are important elements to address the ability to seamlessly create, understand, and evolve access control policies. The first part of this chapter addresses the five dimensions for functional security through a set of security extensions to UML to model different access control concerns and through a mapping to aspect-oriented security enforcement code. The remainder of this section describes the UML extensions to support security modeling of functional requirements in Section 3.1 followed by a discussion of the automatic generation of code via a mapping from UML in Section 3.2.

### 3.1 Functional Security Extensions to UML

To address the access control modeling, customizability, and visual notation dimensions, our work proposed a set of extensions to UML to model different access control concerns (Pavlich-Mariscal, Demurjian, & Michel, 2010a). This section describes the four extensions in the upper
part of Figure 2, and the secure subsystem, which were conceived to address the five dimensions. To better understand the concepts, assume that designers are required to model a Virtual Chart Application (VCA) for healthcare (Kenny, 2008), which is intended to manage the medical history of a patient (e.g., demographics, vital statistics, medications, appointments, test history, test results, etc.) and is created from multiple sources (e.g., physician office, hospital, imaging center, etc.) to form a combined representation. For simplicity, the example only addresses a subset of VCA. Figure 3 has several diagrams, denoted from a to g, which show all of the proposed UML extensions for access control, namely: Secure Subsystem Diagram (a), Role Slice Diagram (b), User Diagram (c), Delegation Diagram (d), and MAC Extensions (e, f, and g). We examine each UML extension in turn.

The Secure Subsystem Diagram (SSD) denotes all of the resources in the system that requires protection, e.g., a subset of the classes and methods of an application like VCA. Figure 3a shows the secure subsystem for the VCA, represented as a package with the stereotype <<SecureSubsystem>>. The VCA may include several classes with different responsibilities. However, not all of the VCA’s classes and methods require protection, since they may not access sensitive information, or they are classes that are not directly utilized by user interface layers in the application (e.g., private or protected methods). In this example, for simplicity, the Secure Subsystem Diagram contains only the class MedicalRecord with a set of methods to read and write different aspects of the patient’s medical history. Classes and methods not shown in the SSD are assumed to be non-sensitive, thus they do not need protection.

Our second UML extension, the Role Slice Diagram (RSD), provides a means to define permissions by role to the classes and methods that comprise the SSD. Figure 3b is an example RSD for the VCA, where each role is represented by a role slice, an artifact that contains all of the methods and classes from SSD that are authorized for the corresponding role. Visually, a role slice is represented as a package with the stereotype <<RoleSlice>>. Role slices may inherit permissions from parent roles, denoted by the connection with the stereotype <<RoleInheritance>>. Permissions in a role slice are represented as methods with the stereotype <<pos>> or <<neg>>. The <<pos>> stereotype means that the method is a positive permission, i.e., it is explicitly allowed to that role. The <<neg>> stereotype means that the method is a negative permission, i.e., it is explicitly denied to that role.

Figure 3b illustrates BaseRole, Provider, Nurse, Physician, and Patient roles. BaseRole and Provider, denoted with the tag {abstract}, means that these roles cannot be assigned to users, but instead are utilized to contain permissions that are common to two or more roles. The BaseRole represents permissions that are shared by all of the descendant roles, with the Provider role defining shared permissions for its two descendant roles (Physician and Nurse). BaseRole has a collection of get methods to access information in the VCA (from the SSD in Figure 3a) and one set method related to billing. The Provider Role has the positive permission setMedicationHistory which is assigned to both Physician and Nurse through role inheritance. The Physician role is authorized to have access to the entire electronic health record (EHR) of the patient through the VCA via the BaseRole (read access via get methods) and Provider and Physician roles (write access via set methods). The Nurse role inherits all of the positive permissions from BaseRole and Provider, and has an additional positive method (setAppointmentHistory). Negative permissions can be used to override positive permissions in ancestor roles. For instance, while BaseRole is allowed to access method setBillingHistory, Patient is explicitly denied that permission. The negative permission in Patient overrides the positive one at BaseRole. Overall, the permissions of a given role slice is the union of the permissions directly
An Integrated Secure Software Engineering Approach

Figure 3. Example of UML extensions for access control
An Integrated Secure Software Engineering Approach

assigned to that role slice, with the permissions directly assigned to all of its ancestors and inherited down the RSD. For instance, Nurse has access to: setAppointmentHistory, which is directly assigned to Nurse; setMedicationHistory, which is assigned to Provider; and, all of the methods, assigned to BaseRole.

Our third UML extension, the User Diagram, shown in Figure 3c, focuses in denoting the roles assigned to each user in the system, with the role retaining the actual permissions. This approach allows a role’s permissions to change without having to change each user that is assigned that role, easing the administrative process for security definition. A user is denoted as packages with the stereotype <<User>>. A user-role association is denoted as a dependency relation with the stereotype <<RoleAssignment>>. In this example, role slice Patient is assigned to user Alice and role slice Physician is assigned to user Bob. The user diagram also denotes Separation of Duty (SoD) constraints. A separation of duty between two roles means that a user cannot play both roles simultaneously. Pictorially, a SoD constraint is represented as an n-ary relation with the stereotype <<SoD>>. In the example, the roles Patient and Physician have a separation of duty constraint, providing the appropriate context that a patient cannot be his/her own physician.

Our fourth UML extension, shown in Figure 3d for VCA, is the delegation diagram that depicts all of the delegation constraints in an access control policy. The main artifact of this diagram is the delegation slice, a package with the stereotype <<DelegationSlice>> that represents the roles that a user may delegate to others; note that a user can only delegate a role that s/he is assigned to. These permissions can be assigned to specific users through the delegation assignment relation, a dependency relationship with the stereotype <<DelegationAssignment>>. In the example, the user Alice has permissions to delegate role Nurse and Physician. The role slice Physician has the tag {da}, which means that Alice has delegation authority (da) over role Physician, i.e., Alice can delegate the role Physician to another user, but the latter cannot further delegate that role to another individual. The role slice Nurse has the tag {poda}, which means that Alice has pass-on delegation authority (pado) over role Nurse, i.e., Alice can delegate the role Nurse to another user and that user can further delegate that role to other individuals.

Lastly, to realize MAC policies in UML, we also provide several elements to represent classifications, clearances, and sensitivity levels in a policy. Figure 3g is a MAC diagram, which represents sensitivity levels and their order relations. Sensitivity levels are depicted as packages with the stereotype <<SensitivityLevel>>, while order relations are dependencies with the stereotype <<order>>. In the example, four levels are defined: unclassified (u), confidential (c), secret (s), and top-secret (ts). These sensitivity levels can be used to tag methods in the secure subsystem diagram to denote classifications, and in role slices to denote clearances. Figure 3e is a secure subsystem diagram with classifications in its methods. In this example, the cls tag indicates the classification of the method, which is secret for all of them. The am tag indicates the access mode to the method, which can be either read or write. In the example, all of the methods are tagged as write-methods. Figure 3f is a role-slice diagram with clearances. Each role slice has a clr tag that indicates the clearance of the corresponding role. In this example, Patient has confidential clearance, while Physician has secret clearance.

### 3.2 Transition from Functional Security Design to Automatically Generated Code

To properly realize an access control policy in an application, the access control design must be translated into its equivalent in code. In this regard, two of the dimensions discussed in the introduction of Section 3 must be taken into ac-
An Integrated Secure Software Engineering Approach

Figure 4. Basic security enforcement code

```
public void setMedicalHistory() {
    if (Session.isUserAuthorized(user, "setMedicalHistory")) {
        // Original Code of setMedicalHistory
    } else {
        throw new RuntimeException("Access denied to method setMedicalHistory");
    }
}
```

Separation of concerns is important to properly modularize the system, facilitate distribution of tasks among developers, and above all, to improve the maintainability of the system (Parnas, 1972). Moreover, separation of concerns must be preserved from design models into code, in the sense that a generated code element must have a similar separation from other concerns as the corresponding access control diagrams (i.e., secure system, role slice, user, and delegation) are separated from standard UML diagrams (e.g., class, use-case, sequence, activity, etc.). Preserving separation of concerns in this way ensures that access control concerns are separated from other concerns both at the design and code. Therefore, it facilitates traceability of access control model elements to the parts that realize them in code. However, separating security concerns from the rest of the code is difficult, because security tends to be scattered across the code and tangled with other requirements (De-Win, 2004). For instance, one way to protect a Java method according to access control rules would be the code of Figure 4. In this example, method setMedicalHistory is being added in a conditional statement to realize security enforcement code which protects the method according to user permissions. This example is very simple, since the enforcement code is directly added to the method as an if-statement that verifies whether the logged user is authorized to access that method. If so, the code of the method is executed. Otherwise, an exception is raised and the access is denied.

To protect all of the methods in the system, similar code would have to be inserted at every method that requires protection. As a result, the security enforcement code is scattered throughout all of the methods that need access control and it is tangled with the code that is specific for each method. To address this problem, we proposed an approach based on aspect-oriented programming (Kiczales & Irwin, 1997) to enforce security in an application, shown in Figure 5. At the design level, designers create different access control diagrams by selecting each of their features (see Section 3.1 again). All of these diagrams are composed with one another into a custom access control policy. Aspect-oriented code is generated that realizes this policy in the form of aspects and classes that preserves the separation of concerns given at the design level to the realization of custom access control code at the implementation level, thereby achieving traceability. The aspect weaver and the compiler compose these components into the custom access control code for the application (Pavlich-Mariscal, Demurjian, & Michel, 2010b).

The main element of the generated code is the access control aspect show in Figure 6. Instead of manually inserting the enforcement code at each method, an aspect is created that weaves the enforcement code at every method belonging to the secure subsystem. In this example, the pointcut secSubs corresponds to all of the methods in the secure subsystem (only two join points are shown for space reasons). The advice that follows is executed before any method in the secure subsystem is executed. This advice obtains
the information about the method being executed (variable op), the object to which the method is executed (obj), and the arguments of the invocation (args). Then, the advice checks whether the active user is allowed to execute the method op over object obj, with arguments args. If not allowed, it throws a runtime exception and denies access to the method. Otherwise, the execution of the method continues normally.

The mappings to code also include a set of code generation strategies for secure subsystems and the UML extensions. Figure 7 shows a portion of the generated code for the secure subsystem of the VCA, which is mapped to a class with the annotation @SecureSubsystem. Each class in the
An Integrated Secure Software Engineering Approach

Figure 7. Secure subsystem code

```java
@SecureSubsystem
public interface VirtualChartApplicationSecureSubsystem {
    @ReferencedClass(MedicalRecord.class)
    interface MedicalRecord {
        @am(AM.WRITE) @cls(S.class) public void setMedicalHistory(...);
        @am(AM.WRITE) @cls(S.class) public void setAllergyHistory(...);
        ...
    }
}
```

secure subsystem maps to an interface with the annotation @ReferencedClass, which indicates the Java class that is being protected. This interface includes the methods being protected. Each of those methods includes the annotation @am to indicate the access mode (read or write) and the annotation @cls to indicate the classification of the method. Sensitivity levels map to an interface hierarchy, as shown in Figure 8, where the order relation between levels maps directly to interface inheritance. In the example, level C (confidential) is less secure than S (secret), thus interface C extends S. Similarly, role slices and delegation slices are directly mapped into interfaces. For instance, Figures 9 and 10 show portions of the code that realize the Nurse and Patient role slices, respectively.

Each role slice maps to an interface with the annotation @RoleSlice. Clearances are denoted with the annotation @clr. The interface contains a set of inner interfaces with the annotation @ReferencedClass which references the classes that are restricted by the corresponding role slice. The inner interfaces contain a set of methods annotated with @pos or @neg to denote positive or negative permissions, respectively. Interface Patient also has the annotation @SoD to indicate that it has separation of duty with the role slice Physician. Figure 11 is an example of the code generated from delegation slices. Delegation slices map to interfaces with the annotation @DelegationSlice.

Figure 8. Sensitivity levels code

```java
interface TS extends SensitivityLevel { }
interface S extends TS { }
interface C extends S { }
interface U extends C { }
```

Figure 9. Nurse role slice code

```java
@RoleSlice
@clr(S.class)
public interface Nurse extends Provider {
    @ReferencedClass(MedicalRecord.class)
    public interface MedicalRecord {
        @pos public void setMedicationHistory(...);
    }
}
```

Figure 10. Patient role slice code

```java
@RoleSlice
@clr(U.class)
@SoD(Physician.class)
public interface Patient extends BaseRole {
    @ReferencedClass(MedicalRecord.class)
    public interface MedicalRecord {
        @neg public void setBillingHistory(...);
    }
}
```
An Integrated Secure Software Engineering Approach

and contain inner interfaces, each one associated with the role slices allowed for that delegation slice. Inner interfaces are annotated with @da or @poda for delegation authority or pass-on delegation authority, respectively.

The above strategies to generate access control code have several important properties. First, the generated code is clearly isolated into specific modules. The access control aspect effectively modularizes enforcement code. The interfaces generated from access control diagrams (Figures 7-11) also belong to clearly identified modules. In other words, scattering of security code is effectively addressed. Second, the generated code is not mixed with non-access control concerns. The access control aspect contains only enforcement code. This code is not present anywhere else where other concerns are located. The interfaces generated from access control diagrams also contain only access control code and they are cohesive, in the sense that each interface contains only information associated to a specific artifact from the model level. For instance, the interface Nurse only contains the information from role slice Nurse, thereby eliminating tangling. Third, mappings to code are almost 1:1, which means that traceability is significantly improved. For instance, role slice Nurse from the design model is clearly traceable to the interface Nurse at the code level (Figure 9). Similarly, sensitivity levels from the MAC diagram are clearly traceable to the interfaces unclassified, confidential, secret, and top secret (U, C, S, and TS) at the code level. Overall, at the design level, access control modeling, visual notation, and customizability are addressed by our access control extensions to UML. At the code level, the mappings to code preserve separation of concerns and improve traceability.

4 COLLABORATIVE SECURITY

Collaborative computing is emerging a number of settings including: web portals such as Mediawiki and Sharepoint that facilitate interactions of stakeholders that are authoring, creating, and editing a shared content repository; collaborative software development tools such as Tigris and Git that turn the software development process into a truly shared activity; and, the health care setting where medical providers must interact and communicate with one another (and patients and their families) (Berhe et al., 2011; Abraham & Reddy, 2008; Agrawal et al., 2007). In the latter case, as a patient with a chronic condition transfers among different settings of care, there is increasing need to provide access to a patient’s medical record that is stored in multiple locations and in different systems. The Virtual Chart Application (VCA) as introduced in Section 3 must be capable of limiting access while simultaneously promoting effective and timely collaborative treatment. Traditional RBAC approaches utilize separation of duty, mutual exclusion, and cardinality constraints (Ahn & Sandhu, 2000; Han et al., 2007) to limit what user can do. However, for collaborative health care, patient privacy and confidentiality must be protected while promoting and encouraging shared access of information by medical providers so that decisions can be made in a timely manner on the most up-to-date medical record of a patient.

Our prior work in this regard has been a model for obligated collaboration on duty and adaptive workflow (COD/AWF) (Berhe et al., 2009, 2010, 2011) that extends RBAC and UML with capabilities that include: secure collaboration to control access to data at the correct time (e.g., providers access a shared VCA); obligated collaboration

Figure 11. Delegation slice code

```
@DelegationSlice
public interface Delegations {
    @da interface Physician {}
    @poda interface Nurse {}
}
```
which denotes individuals that must participate and how they interact towards a common goal (e.g., the internist, cardiologist, and radiologist who are all involved in treating the same patient); and, team-based collaboration, which defines the collaboration with multiple individuals toward a specific task (e.g., the team of medical providers and their interactions). This part of the chapter addresses collaboration security via extensions of the work in Section 3 and of UML to model collaborative structures and the interactions of individuals playing roles towards a common goal. The remainder of this section describes the UML extensions to support security modeling of collaboration requirements in Section 4.1 followed by a discussion of the automatic generation of annotated code for enforcement via a mapping from UML in Section 4.2.

4.1 Collaboration Security Extensions to UML

This section describes our extensions to UML in support of collaboration of duty and adaptive workflow (COD/AWL). As shown in Figure 12, there are four new UML diagrams: the collaboration workflow diagram in Figure 12a which is in charge of defining the workflow requirements; the extended role slice diagram in Figure 12b which defines the roles and privileges for each collaboration step; the collaboration obligation slice diagram in Figure 12c which defines the...
required permissions and participations for a particular collaboration; and, the collaboration team slice diagram in Figure 12d which is in charge of defining teams. To begin, the collaboration workflow diagram in Figure 12a extends the UML activity diagram and allows the security engineer to focus only on interactions of a set of users playing roles through a set of collaboration steps as denoted by the \texttt{<<CsSlice>>} stereotype to represent each activity. Note that the set of activities in Figure 12a are each of type \texttt{<<CsSlice>>} which includes the steps Triage through TreatmentPlan with a linear flow, which can be forked in the last step to either Admit or Discharge a patient. The entire activity diagram for collaboration workflow is specialized using the \texttt{<<CwSlice>>} stereotype which serves as the second part (along with \texttt{<<CsSlice>>}), which via the type “CW” (collaboration workflow) can be associated with the other three diagrams in Figures 12b, 12c, and 12d.

Next, in Figure 12b, the Extended Role Slice Diagram is shown, which augments the one from Figure 3b to define permissions at a higher level of abstraction. Specifically, \texttt{<<PosRoleSlice>>} and \texttt{<<NegRoleSlice>>} (only positive/negative permissions allowed) can be applied to packages which contain classes which are restricted to only positive (negative) permissions utilizing the \texttt{<<pos>>}(<<neg>>) stereotype. \texttt{<<PosRoleSlice>>} only allows the specification of positive permissions and is used in the root role slice to set the scope of allowed privileges throughout the collaboration, while \texttt{<<NegRoleSlice>>} only allows the specification of negative permissions which is utilized to further restrict privileges in a particular collaboration step (CS). This insures that team members with roles cannot utilize permissions that are not permitted in the corresponding CS. The role slice diagram in Figure 3b defined permissions for the Emergency Room Collaboration (ERC) and has Provider, Physician, and Nurse role slices. The Nurse role slice negates getBillingHistory while the Patient role slice negates all of the methods in the MedicalRecord class. During the collaboration, all of the activated permissions must be a subset of the allowed permissions which is modeled using \texttt{<<RoleInheritance>>}, so that the assigned permissions to the collaboration workflow (\texttt{<<CwSlice>>}) is represented as the root role slice (see Figure 12b), namely the MedicalRecord class.

The Obligation Slice Diagram shown in Figure 12c defines the set of permissions that must be activated and the set of roles that must participate in each collaboration step and in the overall collaboration workflow. These complement RBAC’s separation of duty and cardinality constraints and model the obligation requirement (i.e., who is allowed to perform which method at which time) (Ahn & Sandhu, 2000). The \texttt{<<ObligationSlice>>} stereotype applies to packages that only contain obligated permissions and roles using \texttt{<<obl>>} obligation sets with the \texttt{<<ObligationSubset>>} relationship. For instance, during Triage, a Physician role must be present and the medication history must be read (getMedicationHistory in diagram 1 of Figure 3) before a decision is made on the way to continue with the patient treatment process. In Figure 12c, obligated participation implies that a role must activate at least one of the authorized permissions, which for the ERC team means that a physician is a role that can be obligated to participate. With regards to obligated permission activation, getMedicationHistory and getAllergyHistory must be activated before the collaboration terminates. Permissions from Figure 12b are used to constrain the role slice elements within the obligation slice. Permission activation requirements are modeled as classes along with their obligated permissions that are elements of the obligation slice marked using the \texttt{<<obl>>} stereotype. The root obligation slice represents the set of obligations that must be activated during the collaboration, while each collaboration step must fulfill a subset of the root obligations.
obligation slice. To correctly link the obligation slices with the corresponding collaboration step, we utilize both the name of the obligation slice and its tagged note, which distinguishes between collaboration step slices and collaboration workflow obligation slices.

Finally, the team slice diagram in Figure 12d depicts a separate concern to capture permissions for a team. In the ERC example, each of the collaboration teams contains the specific role slices that are needed (see Figure 12b). Using the subset (<<TeamSubset>>) relationship for the team slice diagram, the root team slice represents the entire collaboration team and all of the CSs subset team members from this root team slice. A team slice is depicted as a UML package with the stereotype <<TeamSlice>>.

This package contains a set of role slices. The <<TeamSlice>> diagram is used to define a team of <<RoleSlice>> members. For example, each collaboration step illustrated in Figure 12a (e.g., Triage, Lab Test, X-Ray Test, EKG Test, etc.) has its own team that participates in this context. To simplify the administration, we utilize the <<TeamSubset>> stereotype to indicate the parent-child relationship between two <<TeamSlice>> specialized packages. For permission activation, team membership allows a role to be authorized to a particular permission in the collaboration. The root team slice contains the set of roles that are allowed to participate throughout the entire collaboration. To flag this setting, we utilize the tagged note which contains the stereotype <<CwSlice>> ERC which indicates that this team slice defines the underlying setting based on which sub-teams are defined for each of the collaboration steps. To indicate that a team slice corresponds to a collaboration step, the tagged note that contains the stereotype <<CsSlice>> with the corresponding collaboration step name. Both stereotypes (<<CwSlice>> and <<CsSlice>>) are used to match the teams with the other three diagrams in Figures 12a, 12b, and 12c.

### 4.2 Transition from Collaborative Security Design to Automatically Generated Policy Code

Once the COD/AWF requirements have been specified using UML as given in Section 4.1 and shown in Figures 12a to 12d, the next logical step is to generate code that can enforce the defined security policy for collaboration. To accomplish this, we utilize the meta-programming capabilities of the Java language to define the COD/AWF and RBAC policies that then are used to enforce their correct authorization. The main reason that we utilize Java is that its meta-programming capabilities provide a simple mechanism to define annotations, which allows additional semantic information to be assigned to any Java entity (e.g., variable, method, class, package, etc.). These annotations can be preprocessed before the compiler runs the code using the Java reflection library. Java expresses semantic meta-data using the character @ to signify an annotation. If the annotations are not read using meta-programming, the semantics of a program does not change. A summary of the annotations is given in Table 1 in order to allow the reader to more easily follow the discussion that maps from the diagrams of Figure 12 to annotated Java code.

First the team slice diagram from Figure 12d is mapped to the code given in Figure 13, using a public interface annotated with @TeamSlice. The interface name determines the collaboration workflow that the team belongs to. The team members are composed out of roles, which are defined within the roles interface. For each collaboration step, only a subset of the team members may be required. To specify the subset relationship between the entire collaboration and a particular collaboration step, we introduce the @TeamSubset annotation that allows only roles from the super set of roles specified in the entire collaboration workflow. This information is parameterized in the @TeamSubset annotation. In Figure 13, the entire collaboration team is composed out of six of
the roles. Throughout the collaboration workflow, only those roles are allowed to participate. For the specific collaboration step named Roundtable, collaboration is limited to the nurse role and the physician role. The remaining four roles are not allowed to participate during the Roundtable collaboration step.

Second, the mapping between the extended role slice diagram from Figure 12b and the Java code in Figure 14 is equivalent with the role slice diagram presented in Section 3.1, except for one additional annotation. To satisfy the subset relationship between the collaboration workflow (root extended role slice) and the collaboration steps (child extended role slices), the @NegRoleSlice annotation is added. An interface with this annotation only allows negative permissions. For instance in Triage collaboration step, all of the methods from the parent extended roles are permitted except for the two methods related to appointment history and billing.

Third, the obligation slice in Figure 12c for the entire collaboration workflow and each collaboration step is mapped in Figure 15 to Java using the @ObligationSlice annotation. Similar as in the extended role slices, the subset relationship between the entire collaboration workflow and each of its collaboration steps is achieved using the @CodecSubset annotation. This dictates that only a subset of the obligations can be re-used. The obligated methods and roles are wrapped as public variables within the collaboration step/workflow interface. For example, during emergency room triage, the physician must check the allergy and medication history of the patient. After triage, if the patient is discharged, the office staff role must update the billing portion of the patient’s medical record.

Finally, mapping of the Collaboration Workflow Slice Diagram in Figure 12a to Figure 16 in Java is accomplished by annotating the collaboration workflow with @CollabWorkflowSlice. Within the collaboration workflow, each col-

<table>
<thead>
<tr>
<th>Table 1. Annotations for policy generation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>@Pos</strong>: The UML &lt;&lt;Pos&gt;&gt; stereotype defined during the design time, which states that the role is authorized to activate the method.</td>
</tr>
<tr>
<td><strong>@Neg</strong>: The UML &lt;&lt;Neg&gt;&gt; stereotype defined during the design time, which states that the role is not authorized to activate the method.</td>
</tr>
<tr>
<td><strong>@PosRoleSlice</strong>: The UML &lt;&lt;PosRoleSlice&gt;&gt; stereotype defined during the design time, which states that this role slice can only have positive permissions.</td>
</tr>
<tr>
<td><strong>@NegRoleSlice</strong>: The UML &lt;&lt;NegRoleSlice&gt;&gt; stereotype defined during the design time, which states that this role slice can only have negative permissions.</td>
</tr>
<tr>
<td><strong>@TeamSlice</strong>: The UML &lt;&lt;TeamSlice&gt;&gt; stereotype defined during the design time, which states that this role slice can only have negative permissions (see @Neg annotation).</td>
</tr>
<tr>
<td><strong>@TeamSubset</strong>: The UML &lt;&lt;TeamSubset&gt;&gt; stereotype defined during the design time which states that a team may only be composed out of a subset of the parent team.</td>
</tr>
<tr>
<td><strong>@Obl</strong>: The UML &lt;&lt;Obl&gt;&gt; stereotype defined during the design time the set of permissions that must be activated and the set of roles that must participate.</td>
</tr>
<tr>
<td><strong>@CollabWorkflowSlice</strong>: The UML &lt;&lt;CollabWorkflowSlice&gt;&gt; stereotype defined during the design time wraps the collaboration workflow specifying the set of collaboration step and their order for a particular collaboration.</td>
</tr>
<tr>
<td><strong>@CollabSlice</strong>: The UML &lt;&lt;CollabSlice&gt;&gt; stereotype defined during the design time, which specifies a collaboration step.</td>
</tr>
<tr>
<td><strong>@NextCollabSlice</strong>: The UML &lt;&lt;NextCollabSlice&gt;&gt; stereotype defined during the design time, which specifies the set of collaboration steps followed by a particular collaboration step.</td>
</tr>
</tbody>
</table>
An Integrated Secure Software Engineering Approach

Collaboration step is defined as a separate interface and parameterizes the set of ensuing collaboration steps using the @NextCollabSlice annotation. The collaboration steps with an empty @NextCollabSlice represents the last collaboration step in the collaboration workflow. In Figure 16, the interface Triage specifies that potential subsequent collaboration steps are limited to Test, Admission, and Discharge. During runtime, this information will be looked up to verify the clinical workflow.

5 INFORMATION SECURITY

The third aspect of our work with respect to software security focuses on the control of information created by an application that in turn is shared and/or exchanged with other applications. Information exchange can be achieved in multiple ways; efforts to structure data in order to provide a common processing model of information has resulted in the creation of formats such as the JavaScript Object Notation (JSON) and the eXtensible Markup Language (XML). The latter has emerged as the de facto standard for information exchange, with its use being leveraged for the creation of document standards to which information has to conform. For example, following the health care domain theme utilized throughout this chapter, standards such as the Continuity of Care Record (CCR) and Health Level 7’s (HL7) Clinical Document Architecture (CDA) have been utilized to represent patient information in various care settings and situations. These standards are realized as XML schemas, which serve as a blueprint for the new document instances and as a validation agent for existing ones. The definition of XML schemas has become a necessary component of the software engineering process, usually with the purpose of handling exchange between the components that comprise an enterprise application.

With the purpose of supporting information-based security, our previous work (De la Rosa Algarín et. al, 2012, 2013a and 2013b) has extended the Unified Modeling Language (UML) with two new diagrams: an XML Schema Class Diagram (XSCD) to model a XML schema as a UML class diagram; and, an XML Role Slice Diagram (XRSID) to model RBAC requirements of a specific application with respect to the original XML schema.

```
@TeamSlice
public interface ERC {
    @ReferencedTeamClass(cod.Roles.class)
    public interface Roles {
        public interface Physician();
        public interface OfficeStaff();
        public interface ERPPhysician();
        public interface Patient();
        public interface Nurse();
    }
}
@TeamSlice
@TeamSubset(name="TeamSlice",value="ERC")
public interface Roundtable {
    @ReferencedTeamClass(cod.Roles.class)
    public interface Roles {
        public interface Physician();
        public interface Nurse();
    }
}

@PosRoleSlice
public interface ERC {
    @ReferencedPermissionClass(cod.Emr.class)
    public interface EMR {
        @pos public String getAllergyHistory();
        @pos public String getMedicationHistory();
        @pos public String getBillingHistory();
    }
}
@NegRoleSlice
public interface Triage extends ERC {
    @ReferencedPermissionClass(cod.Emr.class)
    public interface EMR {
        @neg public String getBillingHistory();
    }
}
```
An Integrated Secure Software Engineering Approach

The definition of these two UML artifacts permits us to generate enforcement policies in the eXtensible Access Control Markup Language (XACML), which provides a common data and processing model for policy enforcement. This third part of this chapter addresses information security via extensions of the work in Section 3.1 and of UML to model the information structure (via XML schemas) at a level that allows abstract security policy definitions in UML from which XACML policies can be generated. The remainder of this section describes the UML extensions to support security modeling of information requirements in Section 5.1 followed by a discussion of the automatic generation of annotated code for enforcement in XACML via a mapping from UML in Section 5.2.

5.1 Information Security Extensions to UML

XML schemas (and instances) exhibit a hierarchical structure as part of their specifications. Elements (nodes) can have sub-elements (children), and these can be of different types, such as xs:element, xs:complexType, etc. One approach to modeling with XML schemas is to represent them as UML diagrams (Bernauer et al., 2003; Vela et al., 2003). In our work (De la Rosa Algarín et al., 2012), we extend UML with two new diagrams: the XML Schema Class Diagram (XSCD) to model the architecture, structure characteristics, element constraints, and relations of an XML schema; and, the XML Role Slice Diagram (XRSD) to model the role security requirements with respect to information, which contrasts to the method level focus of the role-slice diagram in Figures 3b and 12b. To begin, the XSCD that represents a given XML schema for an application will have a set of XSCDs (one corresponding to each XML schema) to support the definition of information security. Recall that XML schemas are characterized by a hierarchical structure with data type constraints. To handle the hierarchical nature of XML schemas, we represent each xs:complexType in the schema as a UML class with their respective UML stereotype. If an xs:element is a descendant of another schema concept, then this relation is represented as an equivalent class, denoted by a subclass relation in the class diagram. This holds true for xs:sequence, xs:simpleType, etc. XML schema extensions (xs:extension) are represented as associations between classes. Data-type cardinality requirements (minOccurs, maxOccurs) and constraints are represented with a generic <<constraint>> stereotype assigned to...
the attribute. The xs:element type is respectively represented with a <<type>> stereotype to determine its type. Figure 17a shows the XSCD for the HL7’s CDA “section” elements (upper left). Note that the hierarchical nature of the XML schema segments (upper left of Figure 17a) is maintained with class-subclass relations in the XSCD (lower left of Figure 17a), and the element constraints are set inside each respective element class.

After the XML schema utilized by the application is modeled and represented as a UML class diagram (with the XSCD), the next step is to model the security requirements of the application that are related to the information to be used. To model these requirements, the XML Role Slice Diagram (XRSD) can be employed to represent the way that each role targets information, which complements the approach from the RSDs in Sections 3.1 and 4.1 that are method based. We note that permissions on XML documents are read, no read, write, and no write. To represent these, we set stereotypes with the combination of permissions with respect to a role (4 total): <<read/write>>, <<read/nowrite>>, <<noread/write>>, and <<noread/nowrite>>, as shown for the XRSD in Figure 17b. Pictorially, the XRSD is an UML package via an <<XRSD>> stereotype, and class diagrams are members that represent each of the elements of the XML schema (form the XSCD) that need to be secured. These diagram segments, which are organized hierarchically depending on their position with respect to the XML schema.
tree, contain stereotypes that represent the operations permitted by the role being described. As an example, consider Figure 17b using the Virtual Chart Application (VCA) and two sample XRSDs for Physician and Nurse, which describes the permissions they have with respect to the “section” schema elements. In Figure 17b, Physician has read/write access to most of the section element (top of Figure 17a) except for the past medical history, which is nowrite (i.e., in medical record, information is never overwritten, but instead is added over time). Nurse has read access to the majority of the section element, except for vital signs, which they are allowed to record (write).

5.2 Transition from Information Security Design to Automatically Generated Policy Code

Following a similar path for functional and collaboration security (Sections 3.2 and 4.2), this section explores the generation from an UML-based information security design of an XACML instance enforcement policy. The end result is a policy that allows XML instances to be customized and delivered to users based on their role permissions, which has the benefit that the security privileges defined at a schema level do not impact the XML instances of an application. This means that when privileges and permissions evolve, updating the enforcement policy would just require a regeneration of the policy itself (unlike embedded security methods, which require the update of every existing instance), effectively providing separation of security concerns. Like XML schemas, the new UML diagrams XSCD and XRSD are leveraged as blueprints of the access-control policy for reading and writing permissions in a granular level. Each XRSD effectively describes the permission of a role, and the set of XRSDs for a given application is the set of roles and permissions for said application. To map an XRSD into an XACML policy, we utilize the policies’ language structure and processing model. XACML policies consist of a PolicySet, a Policy, and a Rule. An XACML PolicySet is utilized to make the authorization decision via a set of rules in order to allow for access control decisions. A PolicySet can contain multiple Policy structures, and each Policy contains the access control rules. As a result, the Policy structure acts as the smallest entity that can be presented to the security system for evaluation. To create an XACML Policy structure per each XRSD, Table 2 contains the general mapping equivalences and rules.

Given the XSCDs (Figure 17a) and their XRSDs (Figure 17b) for an application, it is possible to provide an automated process to generate the target XACML policy using the mapping rules (Table2). The first step is creating a template XACML Policy, but with four high level rules. The four rules are divided between the four possible permissions (one for the readable elements, one for the non-readable elements, one for the writable elements, and a fourth one for the non-writable elements). The resources of each of these rules and its single action (mapped from the permission) are associated with the tied permission stereotype of the XRSD. For example, in the Nurse XRSD in Figure 17b, all of the elements that have <<read/*/>> would be mapped as resources to the rule with the mapped action of “read” (and effect Permit since it’s an allowable permission). Those elements with a <</* nowrite>> stereotype in turn would be added to the rule with the mapped action of “noread” and effect Deny. Naturally, all of the subjects of these rules are equal (the role), creating one policy with several rules for a given role. After the mapping processes between rules and XRSD elements/stereotypes are completed, sanitation can be performed. This sanitation verifies that all of the four rules contain at least one resource (element). If one of them does not, then the rule can be dropped. After this step is completed, the XACML Policy is finalized, shown in Figure 18, as a XACML rule for the Physician role. Note that the example shown has been condensed to
An Integrated Secure Software Engineering Approach

Table 2. XACML mapping equivalences and rules

| Policy and Rule Descriptors and Structure: | Policy’s PolicyId attribute value is the XRSD’s Role value concatenated to AccessControlPolicy. |
| Rule’s RuleId attribute value is the XRSD’s Role value concatenated to the XRSD’s higher order element, also concatenated to “ProductRule”. If the permission stereotypes are not all positive or all negative (that is, at least once occurrence of read/write or noread/write), the permission name is also concatenated to the RuleId. |
| Rule’s Description value is the XRSD’s Role concatenated to “Access Control Policy Rule”. |
| Rule Target’s Subject: | Only one XACML Subject and SubjectMatch per Rule. |
| SubjectMatch’s MatchId uses the function “string-equal” to evaluate the user’s role as modeled in the XRSD. |
| AttributeValue of the Subject is a string, and the value is the XRSD’s Role. |
| SubjectAttributeDesignator’s AttributeId is the role attribute. |
| While more than one Rule per Policy might exist, the Subject is equal in both cases. This means that the role to be considered for policy evaluation is the same for operations that are allowed or denied. |
| Rule Target’s Actions: | One XACML Action per operation permitted exists. If the stereotypes of an XRSD are all-positive or all-negative, then both permissions are set as two actions under the same rule. Otherwise, one rule per permission is created (each one with an individual action). |
| ActionMatch’s MatchId uses the function “string-equal”. |
| ActionAttributeDesignator’s AttributeId value is action-write or action-read. |
| ActionMatch’s AttributeValue is the permission, read, write, noread, nowrite, depending on the stereotypes of the XRSD. |

the subject, resource, and action elements of the XACML schema. This segment represents one of the multiple rules (more specifically, the rule of not writing the Past Medical History element) that make the Physician policy.

6 A SECURE SOFTWARE ENGINEERING PROCESS

This section describes a secure software engineering process that spans across the functional, collaboration, and information security concerns for an application. As shown in Figure 19, our process provides a means to define a security policy using security diagrams and features to fully illustrate the creation of a custom security model, the composition of security features, and the security design for functional, collaboration, and information concerns. In Figure 19, the steps are numbered from 1 to 6, and will be referenced when each concept is explained. To illustrate the secure software process in Figure 19, we leverage a big data application from the law enforcement/traffic domain where data collected from traffic crash reports that contain confidential information and access must be limited to authorized users and their respective roles. In effect, a traffic crash report system (CRS) big data application can be defined that would use hand-held and other devices to collect information on accidents (e.g., cars involved, people involved, location, specifics of actual accident, etc.) and would have as a backend a big data repository that can be queried by different stakeholders. This example is based on an actual crash report system in Connecticut that has data from over 20 years that has been a joint effort by faculty in the Civil & Environmental Engineering and Computer Science & Engineering faculty under the supervision of the State of Connecticut Department of Transportation. CRS serves as a means for researchers to collaboratively...
Figure 18. The physician XRSD mapped to XACML

```
<Rule RuleId=""...:names:tc:xacml:2.0:example:ruleId:PhysicianNoWriteRule" Effect="Deny">
  <Description>Physician Access Control Policy Rule</Description>
  <Target>
    <Subject>
      <SubjectMatch MatchId=""...:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">Physician</AttributeValue>
      </SubjectMatch>
    </Subject>
    <Resource>
      <ResourceMatch MatchId=""...:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">hl7:cda:schema:caption:Past Medical History</AttributeValue>
      </ResourceMatch>
    </Resource>
    <Action>
      <ActionMatch MatchId=""...:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">nowrite</AttributeValue>
      </ActionMatch>
    </Action>
  </Target>
</Rule>
```
analyze the data for future crash prevention and other operational purposes. The example presented in this section has been excerpted from the Model Minimum Uniform Crash Criteria Guide (MMUCC) that defines an XML standard for data to be collected on traffic crashes to be stored in CRS.

To begin the process of Figure 19, software engineers create a requirements specification that serves as input to the (1) main security design. Specifically, we assume that a class diagram for CRS has been developed as the output of step 1. This is shown in Figure 20, where a limited UML class diagram for CRS has: the CRS class that can write and read a crash history and share crash data; subclasses for tracking crashes on roads that are federal (FederalCRS), state (StateCRS), on local (LocalCRS); and, the InsuranceReporting class that allows insurance reporting information on the crash to be created and shared. Next, as part

---

**Figure 19. Our secure software engineering process**

(1) Main Security Design of the Application

(2a,b) Initial Functional Security and Collaboration Design

(2a,b.1) Define Functional Security and Collaboration Use Cases

(2a,b.2) Define Secure SubSystem + Collaboration Capable Subsystem

(2c) Initial Information Security Design

(2c.1) Define XML Schema Class Diagram

(2c.2) Define Information Security Requirements

(3a) Functional Security Design

Define Security Features [DONE]

Group Users into Roles [DONE]

NEEDS MAC

Select MAC Features [NOT DONE]

Separation of Duty, Delegation Authority [DONE]

Security Refinement Process [DONE]

[NOT DONE]

(3b) Collaboration Security Design

Create Collaboration Workflow Name [DONE]

Create Collaboration Step/Workflow [NOT DONE]

Security Refinement Process [DONE]

Collaboration Team [DONE]

[NOT DONE]

Collaboration Obligation [DONE]

[NOT DONE]

(3c) Information Security Design

Define set of Roles with Information Access [NOT DONE]

Determine Permissions of Roles to Information [DONE]

Create XML Role Slice Diagrams for each Role [NOT DONE]

Security Refinement Process [DONE]

[NOT DONE]

(4) Refinement of Functional, COD/AWF and Information Security Design

(5) Combine Three Facets and Transition into Final Design

(6) Mapping to Enforcement Code and XACML Policies

Generated Functional, Collaborative & Information Secure System

---

355
of an iterative process, designers create an initial security design of the application (steps 2a, b, and c). For functional and collaboration design, there is a need to define use cases (steps 2a and b.1) as well as to define the secure subsystem diagram for the functional and collaboration concerns. The class diagram in Figure 20 will be utilized to define those methods that have the potential to be accessible to different users based on role. In this example, all of the classes and methods of Figure 20 are sensitive, thus they are all included in the Secure Subsystem Diagram in Figure 21a.

In addition to the class-based design, steps 2a, b, and c in Figure 19 will also include a description of the process and stakeholders involved in CRS at a high level coupled with a more detailed view of a scenario of usage to help guide later steps in the secure software engineering process. At a high level, each state has its own police or shared police with other towns. Accidents that occur in a municipality usually fall under the jurisdiction of the local office. Data collected will be entered in a central and shared regional, state, and federal crash report system that usually allows read and write access to officers at the same level (e.g., state officer to state crash report system). Accidents that happen on a state owned or interstate highway usually fall under the jurisdiction of the state police. Therefore, if a local officer arrives first and enters the crash data report, the local officer is obligated to share and hand over the data to the state officers and state troopers. This achieves the sharing or reports through system integration of the local office’s data with the state officer’s system. Whenever there is a major accident and the local/state officers request federal help or whenever an accident happens between multiple states, the jurisdiction goes to the federal officers. In such use cases, data is often first collected by the local or state officers, and later shared with federal officers with write access. Researchers usually have read access to de-identified portions of the databases systems at the local, state, and federal level to query and analyze the collected big data.

At a more detailed level, a scenario could involve the following. On a Monday morning during rush hour, John Smith accidentally drives into Bob Doe’s car and both cars have broken front tires and rear lights on the interstate highway I-84. Local officer Jackie Kerr while on her way to work, sees the accident and immediately rushes to the location, checks Bob’s and John’s health state, before starting the insurance and personal data collection, as well as information on the way that the accident happened. She enters the information into a hand-held traffic crash report system (CRS). A few moments later, state officer Neal Drake arrives and discusses the accident with Officer Jackie Kerr. Since the accident happened on an interstate highway, Officer Kerr is obligated to share the collected data via the local traffic crash report system so that the state officer can also access. Officer Neal Drake takes over the accident location and queries John Smith’s accident history. The results show that Mr. Smith has a history of convicted car accidents in five states over the past
Figure 21. Various diagrams for the CRS access control policy
three years. Due to the multi-state car accident history, John Smith’s collected accident information will be automatically shared with the federal crash report system read by Officer Ramona Tyler. In parallel, a researcher, Dr. Ervin Walsh, has access to the entire automatically updated de-identified CRS big data repository to query and analyze the data towards crash prevention. In order to make a decision on whether to charge John Smith or not in a legal court, Officers Kerr, Drake, and Tyler have a roundtable discussion about the crash. Using the high-level and detailed view of CRS as input, the functional (step 3a), collaboration (step 3b) and information (step 3c) security design can proceed in parallel.

To begin, Box 3a in Figure 19 is for the functional security design and involves a number of steps. First, from the specification of the secure subsystem, designers select the features they want to incorporate into the access control model. This is reflected in the security diagrams that will be used to specify the policy. In the next step, designers can then start grouping users into roles that denote their common privileges in the system. The result is a role slice diagram that contains only the role slices. Designers can also assign positive and negative permissions to each role slice to realize role-based features of the policy. Using the role slice diagram as input, designers can define separation of duty constraints in the user diagram, and also define delegation slices and delegation authority privileges in the delegation diagram. If the policy also requires MAC features, they can be incorporated into the diagrams as MAC extensions. The resulting policy can be further refined until designers decide it is ready to transition into code.

To illustrate Box 3a, in Figure 21b a role-slice diagram (RSD) of the CRS access control policy is defined. Role slices for LocalOfficer, StateOfficer, and FederalOfficer denote the permissions for each type of officer, which are, essentially, the ability to read, write, and share the crash history data registered in CRS, and also write insurance reporting information in the system with denied read access due to privacy reasons. A DoTWorker would be someone that works for the department of transportation agency of the state. A Passenger can read his/her own crash data in his/her insurance reporting record. A Researcher can read all of the de-identified information in CRS with the intent to allow ad-hoc queries to be composed against the big-data CRS repository in order to look for trends in accidents and their impact on traffic patterns. Also in this step, a User Diagram that denotes the Separation of Duty constraints among all of the roles is defined in Figure 21c. No person with a role can assume any other role in the system. For instance a Local Officer cannot impersonate any other type of officer, passenger, or researcher. A Passenger cannot assume any type of officer role or researcher role.

Next, Box 3b in Figure 19 is for the collaboration security design where the workflow, steps, team, and delegation are defined and potentially refined via done/not done decision states. The first decision that must be made is with regards to which collaboration workflow is going to be designed. The collaboration workflow sets the overall context under which the collaboration occurs along with the specified teams, obligations, and authorized permissions. Given the collaboration workflow context, the second step involves the definition of all of the collaboration steps required in this workflow. This process requires different domain experts and stakeholders to bring in their expertise to define the set of collaboration steps along with their relationship into a collaboration workflow. The specification of the collaboration steps must occur before the security, teams, and obligation slice configuration, since the collaboration step happens in the context of a collaboration workflow, serving as the binding element that is utilized in all of the four new UML diagrams (extended role slice, a collaboration team, a collaboration workflow, and a collaboration obligation) with the convention that matching names imply matching policies for a particular step. As part of the security
An Integrated Secure Software Engineering Approach

refinement process, collaboration step names are used in the other slices, while collaboration team and collaboration obligation have an input to an activity that parameterizes it, which expects a set of collaboration step names. After the role slices and the collaboration workflow is specified, the next step is the creation of obligation and team slice diagrams. The obligation slice diagram requires the permission set and the role set as an input. The team slice diagram requires the role set as an input.

Continuing with the CRS example, from a collaborative perspective in Box 3b, the order in which the officers work together follows a process that can be captured with the four diagrams shown in Figure 22. Specifically, the collaboration workflow diagram in Figure 22a defines the workflow of the high-level detail of CRS stakeholders and their interactions, the extended role slice diagram in Figure 22b defines the roles and privileges for each collaboration step that build off of the role slice diagram of Figure 21b, the collaboration obligation slice diagram in Figure 22c defines the participation and permissions of CRS stakeholders, and, the collaboration team slice diagram in Figure 22d defines the CRS teams. When designing a system that is used in the traffic domain, obligated, coordinated, team-based and secure collaboration capabilities are an integral part from the early phases of the software engineering process.

Figure 22. Four UML Diagrams for collaboration of duty and adaptive workflow for CRS

(a) Collaboration Workflow Slice Diagram.

(b) Extended Role Slice Diagram.

(c) Obligation Slice Diagram.

(d) Team Slice Diagram.
• **Team-Based Collaboration:** Local, state, and federal officers compose a team that work together to share and use the collected crash report data sets.

• **Coordinated Collaboration:** The order in which data is shared goes usually either from local to state to federal officers or reverse.

• **Obligated Collaboration:** If a major accident happens in an interstate highway, the local or state officers must inform the federal officers and share the collected data to determine future jurisdiction.

• **Secure Collaboration:** Depending on the context of the accident, read, write, or no access privileges are given to the three types of officers.

Collaboration can be effectively utilized in order to capture the interactions among the various stakeholders involved in the big data law enforcement/traffic domain application.

Finally, Box 3c in Figure 19 is for the information security design. By considering the roles that will interact with the different information structures of the application, the software engineer will select the role’s permission with respect to information elements. These permissions, which are read and write (and their explicit opposites: no read and no write), are graphically described with respect to each role by the creation of the XML Role Slice Diagram (XRSDs). For example, if an application has 10 different roles, a total of 10 XRSDs will be created, each one representing the role’s permissions with respect to the XML schema the application utilizes. Once these XRSDs are defined, if may also to perform a refinement process once all of the role slices and their potential interactions are defined. This feedback process permits software engineers to polish the roles and their information security permissions before an enforcement policy is created.

Picking up the CRS example again, the roles that are involved in collecting, reviewing, and analyzing the data include the officers at the local, state, and federal level as well as researchers, which are shown in Figures 21b and 22b, and can be expanded and refined as in Figure 23 for permissions and roles. Specifically, in Figure 23, the left-hand side indicates the roles, permissions, and entities (objects), the middle portion represents a PoliceOffice PSD, and the right portion represents a Researcher PSD (see Section 5.1). Box 3c can also be utilized to define the XRSDs. In support of this, from an information security perspective, the different roles that act upon the data might have different permissions (which could range from non-destructive operations, such as reading, to destructive operations, such as inserting or deleting) enforced in the information itself (see Figures 20 and 21). These permissions might be augmented with other security features, such as security levels from the MAC model. In these cases, collaboration might be a part of the information level security as well, where a role can be delegated to players in different layers of the operation, e.g., a state officer grating temporary access to a local officer to complete a task. This capability of information-level security can be granular as well. As shown in Figure 24, there are two XRSDs. Consider the role DoTWorker who works for the state transportation department and as a result may be part of the data acquisition process involving a traffic accident, particularly if it has occurred within a construction site. The DoT worker can have read/write permissions to the alcohol component of the MMCUU instances (MMCUU is an XML schema standard utilized for traffic accident information), but no write permission to the personal information of the parties involved. On the other hand, a PoliceOfficer would have read and write permissions to both these components of the CSR.

Over time, the processes of (steps 3a, b, c) in Figure 19 will eventually stabilize and lead to a consideration of the functional, collaboration, and information concerns that can be reconciled with one another in (step 4), and then combinable into
An Integrated Secure Software Engineering Approach

Figure 23. RBAC with basic operations applied to CRS

![Diagram showing RBAC with basic operations applied to CRS](image)

Figure 24. XML Role Slice Diagrams for Example Roles in the Law Enforcement Case Study

![XML Role Slice Diagrams](image)

a single design instance (step 5). The combined design (step 5) satisfies the current iteration requirements, but it may be changed later to adapt to new requirements. A mapping procedure (step 6) translates the combined design (step 5) into security enforcement policies and code as a set of (step 6) to realize the enforcement mechanism for the application. Designers can further iterate over the design and repeat the whole process until the software is ready for deployment (last box on Figure 19).

7 FUTURE TRENDS

Future trends and research directions in secure software engineering are being influenced by the emergence of new standards, architectures, frameworks, and tools. The health care domain is full of approaches that strive to provide health information exchange (HIE) in order to support a more complete realization of patient data collected from multiple sources in real time to meet care needs. These new architectures and frameworks must be capable of achieving functional, collab-
orative, and information security concerns across a wide range of settings and able to work for any health information technology system. These new approaches to software architectures, development, communication, and information exchange present a varied field of use cases, maintenance, and development. Towards this objective, Section 7.1 explores security for HIE that involves emerging frameworks and architectures, while Section 7.2 discusses ongoing work in secure software engineering that spans different venues, certification agencies, and international organizations.

### 7.1 Security for Health Information Exchange

The Harvard University Sustainable Medical Applications, Reusable Technologies (SMART) Platform aims to provide a uniform, abstracted, well-defined and reusable infrastructure for application developers that want their applications to utilize patient and medical data of all types. At the specification level, SMART provides data owners and data source maintainers with a method to develop abstract containers that essentially define a data model for each data source. The reusable moniker of SMART comes from the fact that an application developed for a container that wraps around a data source, say A, can work in the same manner on another SMART container that wraps another data source, say B. The abstraction of data sources, A or B, into a container layer makes software development targets easier to maintain. For example, a SMART Container for an electronic health record would provide a common interface allowing the backend to integrate to different vendor products while providing the same frontend to applications. In such an approach, it will be necessary to resolve not only data access issues but also address security concerns where the different patient data sources may have alternate security approaches. From a secure software engineering perspective, the SMART application developer will need to have some guarantee that the composition (union) of containers is secured as well as the individual containers for any of the authorized users. This means that the application layer needs to have a level of security assurance when considering isolated or interconnected (by information exchange) applications. In addition, the container (data source) layer needs security assurance when an isolated app obtains information from multiple sources or when a group of applications work in group (as a meta-application, or application of applications) obtaining information from one or more sources.

The second abstraction approach, Open mHealth, is an open source architecture with modular components for health applications to be built from. These components, which come in the form of data visualization units (DVU), data processing units (DPU), and a cache unit (CU), create a hybrid of a model-view-controller paradigm for applications. The idea behind the architecture is to provide different end-point developers (e.g., visualization developers, data processors, data storage developers, etc.) a modular component to work on. In turn, all of these components work together using the architecture specification, effectively providing an abstraction from the different aspects of health application development. As an example of these components, a DPU might use insulin dosages and glucose readings with timestamps in conjunction with other patient information and analyze these units to provide some level of clinical decision support or other purpose. In contrast, a DVU can provide the means to display the results of a DPU analysis, and do so for different roles (e.g., a patient, a health provider, a researcher, etc.). Open mHealth provides the specifications on the way in which the different modular units should be developed, but localized security (as well as globalized) is not discussed. Any secure software engineering methodology for applications developed as one of the different modules in Open mHealth must take into consideration that the original specification must be kept, regardless of the security require-
ments set. For example, a developer of a DPU can develop their own application and define their own security requirements, but the intercommunication aspect of the specification must not be broken. While another component might be denied access based on the security policy definitions, the communication component must be kept intact. This unavoidable constraint is necessary to maintain the original goal of the architecture itself. The two examples demonstrate that the health care domain presents secure software engineering challenges masked in existing abstraction solutions and architectures. Not only is it necessary to keep in mind the functional, collaborative, and information security aspects of an application, it is also necessary to keep in mind the way that some specifications provide constraints along with the complex challenges to the engineering process itself. Scenarios like meta-application security, modular intercommunicating component security, etc., must be considered in regards to the secure software engineering process.

Lastly, there are many ongoing efforts in the United States regarding health information exchange (HIE) for facilitating patient care by bringing together all of the patient data from multiple sources and for promoting data analysis against de-identified repositories. The DIRECT project provides a secure, scalable, and standards-based approach for HIE in support of making patient data available to providers, patients, clinics, hospitals, etc. Using this approach, information can be routed to the correct individuals at the required time with assurances in terms of authentication and a guarantee that the information has been routed to trusted recipients. Another effort is the Nationwide Health Information Network, which provides the policies, services, procedures, and standards that are necessary for secure information exchange of health care data. A third effort, Open Health Tools, is seeking to take a broader look at the integration of personal health care, healthcare delivery, and population health, in order to provide a solution that can respond to a wide range of needs including: regularly scheduled patient care, emergency room treatment, disaster response, public health response, etc. Secure software engineering will play a significant role all of these efforts, in the development of the infrastructure that is necessary to link patient data from multiple repositories.

7.2 Focused Trends in Secure Software Engineering

The International Security Engineering Council (ISSECO) is an association with the main focus of software security. By focusing on the production of secure software, their goal is to provide and establish a secure information environment for all domains, users, etc. ISSECO’s mission statement dictates that in order to create secure software, special knowledge about the software development lifecycle is a requirement. ISSECO notes that due to the development of the Internet, secure software engineering’s importance has increased with respect to providing a desired level of software quality. With the ever-increasing amount of applications, all types of developers must take into consideration software security in order to maintain a level of trust from the customers. In support of this, ISSECO has standardized education and certification of skills for secure software development. These skills cover everything in the process of secure software production from the start to the end of the development lifecycle, including requirements engineering, trust modeling, threat modeling, secure coding, testing, and security response to the final code protection for deployed applications.

The European Network and Information Security Agency (ENISA) has several software engineering initiatives, including the monitoring of EU and international initiatives that aim to solve the issue of secure software engineering. ENISA organizes workshops with the purpose of bringing together different initiatives to foster collaboration and promote research and work efforts. Other goals of ENISA are the development
of secure applications by providing security guidelines for the development of mobile (smartphone) applications. The third main objective of ENISA is the assessment of web standards for the next generation of technologies, including those drafted by the W3C and other organizations. In 2011, ENISA co-organized the Round Table session for Global Secure Initiatives – Beyond Awareness, at the OWASP APPSEC Europe 2011 Conference bringing together researchers and practitioners from the field of secure software engineering to discuss what could be done to ensure that valuable guidance and tools created for secure software engineering.

The Software Engineering Institute (SEI) at Carnegie Mellon has proposed the Security Quality Requirements Engineering (SQUARE) process. SQUARE consists of 9 steps to assist organizations and agencies in building the security and privacy aspects of software in the earliest stages of the engineering process. The benefit of using SQUARE is that organizations can predict schedules and costs of a more secure and consistent software component to address the current poor practice of security requirements defects that can cost from 10 to 200 times the amount to correct during the implementation process versus the cost during requirements specification. Another methodology, called software security measurement and analysis, aims to develop a risk-based approach for measuring and monitoring the security characteristics of software systems across the different steps of the application’s lifecycle. This measurement and analysis approach has taken the form of the SEI Integrated Measurement and Analysis Framework that integrates collected performance data of modular software components, providing a consolidated view of the overall software performance. Also, there is the SEI Mission Risk Diagnostic process that is able to analyze a defined set of risk factors, providing decision makers with a benchmark of a software or system’s latest state. Lastly, SEI also has a software assurance curriculum with a focus to identify a core of knowledge necessary to develop a Master of Software Assurance degree (for educational institutions) and promote early specialization in software assurance and related areas.

8 CONCLUSION

The design, development, and deployment of software applications must fully embrace the definition and realization of security requirements in the earliest stages of the software design and development process. While software engineering at the design level is dominated by the utilization of the Unified Modeling Language (UML), this approach has a decided lack of consideration of security, which is a paramount concern in today’s society. There exists a broad variety of security models that should be supported, including: role-based access control (RBAC) to allow privilege definitions to be geared towards a user’s responsibilities and duties; discretionary access control (DAC) to allow a user to delegate authority in certain situations; and, mandatory access control (MAC) to strictly control access to information via security levels. However, their inclusion as part of the software engineering processed is often relegated as an afterthought, with little consideration given to security concerns. This chapter has addressed the need for secure software engineering of functional, collaborative, and information concerns via an extension of UML with RBAC, DAC, and MAC, allowing for applications to be defined to address security during early and all stages of the software development process. Towards this objective, this chapter has explored our extensions of UML for RBAC, DAC, and MAC, supporting the definition of functional, collaborative, and information concerns via an extension of UML with RBAC, DAC, and MAC, allowing for applications to be defined to address security during early and all stages of the software development process. Towards this objective, this chapter has explored our extensions of UML for RBAC, DAC, and MAC, supporting the definition of functional, collaborative, and information concerns via an extension of UML with RBAC, DAC, and MAC, allowing for applications to be defined to address security during early and all stages of the software development process. Towards this objective, this chapter has explored our extensions of UML for RBAC, DAC, and MAC, supporting the definition of functional, collaborative, and information concerns via an extension of UML with RBAC, DAC, and MAC, allowing for applications to be defined to address security during early and all stages of the software development process.
health care domain, where information must be collaboratively shared and exchanged in a secure manner. To place the work into its proper context, in Section 6, we detailed the secure software engineering process for the functional, collaborative, and information concerns, and we illustrated this process with a second example related to big data for law enforcement and the traffic domain. Future trends for secure software engineering in terms of emerging platforms and efforts that also span different venues, certification agencies, and international organizations, were explored in Section 7. Overall, we believe this chapter establishes a baseline for the inclusion of access control models in a secure software engineering process.

REFERENCES


Zisman, A. (2007). A static verification framework for secure peer-to-peer applications. In Internet and Web Applications and Services (pp. 8–8). IEEE. doi:10.1109/ICIW.2007.11
KEY TERMS AND DEFINITIONS

Aspect-Oriented Programming (AOP): A programming paradigm which incorporates an additional modular unit, the aspect, which provides additional code that is automatically woven at specific points in the rest of the program by the compiler.

Continuity of Care Record (CCR): A document standard for health information typically used for Personal Health Records (PHR) with the intended purpose of information exchange. It provides a universal structure to the patient’s information that can be utilized by different personal health records, applications and systems.

Discretionary Access Control (DAC): An access control model in which subjects have specific rights to access resources and can discretionarily give access privileges to other subjects to its own resources.

eXtensible Markup Language (XML): A structured language utilized for information exchange, standards and information validation via the use of schemas. Its extensibility allows developers and experts to design and implement common standards for the use across systems and domains.

eXtensible Access Control Markup Language (XACML): A security policy language designed from XML. Its specifications allow for a uniform policy language that can be enforced in heterogeneous systems. XACML policies can be enforced at a systems level, software level or information level, depending on the policies’ targets and rules.

Health Insurance Portability and Accountability Act (HIPAA): HIPAA provides a US standard to protect the privacy of personal health information, including PHI.

Mandatory Access Control (MAC): An access control model in which objects have classifications that restrict their access. Only subjects with the adequate clearances can access those objects.

Protected Health Information (PHI): Under HIPAA, in clinical care and clinical research, PHI to date refers to a set of sensitive 18 data elements that must be protected or removed for deidentification purposes.

Role-Based Access Control (RBAC): An access control model in which permissions are assigned to roles, which in turn are assigned to users, who get all of the permissions of the assigned roles.