Adaptive Security for Software Systems

Mohamed Abdelrazek, John Grundy and Amani Ibrahim

Abstract

With continuously changing operational and business needs, system security is one of the key system capabilities that need to be updated as well. Most security engineering efforts focus on engineering security requirements of software systems at design time and existing adaptive security engineering efforts require complex design-time preparation. In this chapter we discuss the needs for adaptive software security, and key efforts in this area. We then introduce a new run-time adaptive security engineering approach, which enables adapting software security capabilities at runtime based on new security objectives, risks/threats, requirements as well as newly reported vulnerabilities. We categorize the source of adaptation in terms of manual adaptation (managed by end users), and automated adaption (automatically triggered by the platform). The new platform makes use of new ideas we built for vulnerability analysis, security engineering using aspect-oriented programming (AOP), and model-driven engineering techniques.

1. Introduction

Enterprise security objectives, reported risks and threats, and vulnerabilities are the main sources of software security requirements. During the software development lifecycle, software vendors iteratively refine these high-level security needs into software security requirements and mechanisms to be used. Many software security engineering efforts [2] have been developed to help software vendors in capturing, modeling, refining, and engineering these security requirements into their software systems at design time or ultimately at deployment time – making use of the SOA architecture. These design time security engineering efforts are very important not only in engineering users' security requirements into software, but also in engineering secure systems – i.e. taking into consideration secure software development best practices in architecting, designing, coding and testing the underling software.

However, security has never been an one-time process. Enterprise security objectives, risks and threats (e.g. execute arbitrary scripts and breach confidential data, elevate malicious user privileges, take the system down), and vulnerabilities change over time due to new business goals, changes in software operational IT environment (e.g. new deployment or operational environment – Cloud/SOA/etc), and the continuously changing threat landscape. This usually requires changing software security capabilities to meet these new requirements. In addition, mitigating new reported vulnerabilities is usually done manually, and sometimes by modifying application source code and deploying new patches. These modifications are usually translated into new software change requests sent to software vendors to effect these new requirements. However, this usually takes a long time to fix [1]. As shown in Figure 1, time greatly lags between vulnerability detection and patching. This means that a software service remains vulnerable to security breaches exploiting such vulnerabilities. The possibility of vulnerability exploitation increases dramatically in cloud computing, given the public accessibility of the cloud services and the sharing of services with multiple tenants. Thus, in such deployment models there is an increasing need for an online, automated vulnerability patching approach that can stop such vulnerabilities once reported.

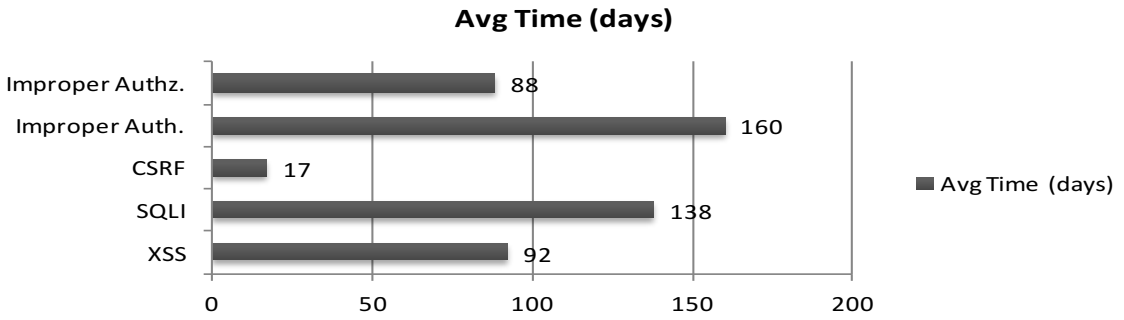

Figure 1. Average time to fix security vulnerabilities (in days) [1]

In this chapter, we introduce a new adaptive security engineering approach meant to address the following key challenges:

1) What are the key security aspects we should capture?
2) How can we model such security requirements taking into consideration that these requirements will change overtime?
3) How can we detect new security vulnerabilities in a running system?
4) How can we automate the realization/enforcement of the new security requirements and virtual patching of reported vulnerabilities at runtime?

Our work is based on externalizing the security engineering practices so that we can update software security capabilities at runtime, on the fly, reflecting new requirements, new reported vulnerabilities, or automated adaptation actions. All the security requirements modeling and refinement activities are done externally, and easy to change at runtime.

2. Motivation

Consider SwinSoft, an imaginary software company building a large web-based Enterprise Resources Planning (ERP) system, called "Galactic". Galactic provides customer management, order management, and employee management modules. Please see Figure 9 for a detailed description model of Galactic including features, architecture, classes, and deployment details. SwinSoft targets different markets in different countries for Galactic. However, such markets, domains and likely customers have different regulations and information security standards that must be satisfied. Galactic must integrate with diverse customers' existing security solutions and other application security. Moreover, SwinSoft has found that customers' security requirements that Galactic must meet may change dramatically over time.

A customer, Swinburne University now wants to purchase a new ERP solution in order to improve its internal enterprise management processes. Swinburne has special security requirements because it is ISO27001 certified. Its enterprise security architects conduct periodic risk assessment that may require reconfiguring the deployed applications' security to block newly discovered threats. Swinburne also wants to have its ERP system security flexible enough as it is planning to integrate its new ERP system with its partners. This implies that the Galactic application's security will change over time after its deployment.

At the same time, another potential SwinSoft customer, SwinMarket, a big brand supermarket chain, has decided to purchase Galactic. SwinMarket also has a need for highly customizable security options on different system features that Galactic must satisfy. SwinMarket expects security solutions deployed in its application operational environment to change over time. Galactic must be able to be updated quickly to adjust to these as well as any emergent security threats. A delay in patching newly discovered vulnerabilities means a loss of money.

An analysis of this scenario identifies many challenges including: security requirements differ from one

customer to another; each customer's security requirements may change over time based on current operational environment security and business objectives; Galactic system security must be integrated with customers' deployed security controls in order to achieve coherent security operational environment; and new security vulnerabilities may be discovered in the Galactic system at any time. Using traditional security engineering techniques would require SwinSoft to conduct a lot of system maintenance iterations to deliver system patches that block vulnerabilities and adapt the system to every new customer needs.

A better security engineering approach that addresses these challenges should: enable each customer to specify and enforce their security requirements based on their current security needs; security should be applied to any arbitrary system component/entity; no predefined/hardcoded secure points or capabilities, usually built at design time; security specification should be supported at different levels of abstraction based on software customers' experience, scale and engineers' capabilities. Integration of security controls with system entities should be supported at different levels of abstraction, from the system as one unit to a specific system method. The security engineering approach should ease the integration with third-party security controls. System and security specifications should be reconfigurable at runtime.

3. Security Engineering State-of-the-Art

Existing security engineering efforts focus on capturing and enforcing security requirements at design time, security retrofitting (maintenance), and adaptive security engineering. On the other hand, most industrial efforts focus on delivering security platforms to help software developers in implementing their security requirements using readymade standard security algorithms and mechanisms. Some of the key limitations we found in these efforts include: (i) these efforts focus mainly on design-time security engineering – i.e. how to capture and enforce security requirements during software development phase; (ii) limited support to dynamic and adaptive security and require design-time preparation. Benjamin et al [2] introduce a detailed survey of the existing security engineering efforts but did not highlight limitations of these approaches. We discuss key efforts in these areas.

3.1 Design-time Security Engineering

Software security engineering aims to develop secure systems that remain dependable in the face of attacks [3]. Security engineering activities include: identifying security objectives that systems should satisfy; identifying security risks that threaten system operation; elicitation of security requirements that should be enforced on the system to achieve the expected security level; developing security architectures and designs that deliver the security requirements and integrates with the operational environment; and developing, deploying and enforcing the developed or purchased security controls. Below, we summarize the key efforts in the security engineering area.

3.1.1 Early-stage Security Engineering

The early-stage security engineering approaches focus mainly on security requirements engineering including security requirements elicitation, capturing, modeling, analysing, and validation at design time from the specified security objectives or security risks. Below we discuss some of the key existing security requirements engineering efforts.

Knowledge Acquisition in automated Specification (KAoS) [4] is a goal-oriented requirements engineering approach. KAoS uses formal methods for models analysis [5]. KAoS was extended to capture security requirements [6] in terms of obstacles to stakeholders' goals. Obstacles are defined in terms of conditions that when satisfied will prevent certain goals from being achieved. This is helpful in understanding the system goals in details but it results in coupling security goals with system goals.

Secure i* [7, 8] introduces a methodology based on the i* (agent-oriented requirements modeling)

framework to address the security and privacy requirements. The secure i* focuses on identifying security requirements through analysing relationships between users, attackers, and agents of both parties. This analysis process has seven steps organized in three phases of security analysis as follows: (i) attacker analysis focuses on identifying potential system abusers and malicious intents; (ii) dependency vulnerability analysis helps in detecting vulnerabilities according to the organizational relationships among stakeholders; (iii) countermeasure analysis focus on addressing and mitigating the vulnerabilities and threats identified in previous steps.

Secure TROPOS [9-11] is an extension of the TROPOS requirements engineering approach that is based on the goal-oriented requirements engineering paradigm. TROPOS was initially developed for agent-oriented security engineering (AOSE). TROPOS introduces a set of models to capture the system actors (actors' model) and their corresponding goals (goal model: hard goals represent the actor functional requirements and soft-goals represent the actor non-functional requirements). These goals are iteratively decomposed into sub-goals until these sub-goals are refined into tasks, plans, and resources. Secure TROPOS is used to capture security requirements during the software requirements analysis. Secure TROPOS was appended with new notations. These included: (i) security constraints: restriction related to certain security issue like: privacy, integrity...etc.; (ii) security dependency: this adds constraints for the dependencies that may exist between actors to achieve their own goals and defines what each one expects from the other about the security of supplied or required goals; and (iii) security entities: are extensions of the TROPOS notations of entities like goals, tasks, and resources as follows: secure goal: means that the actor has some soft-goal related to security (no details on how to achieve) this goal will be achieved through a secure task; secure task: is a task that represents a particular way of satisfying a secure goal; secure resource: is an informational entity that's related to the security of the system; and secure capability: means the capability of an actor to achieve a secure goal.

Misuse cases [12, 13] capture use cases that the system should allow side by side with the use cases that the system should not allow which may harm the system or the stakeholders operations or security. The misuse cases focus on the interactions between the system and malicious users. This helps in developing the system expecting security threats and drives the development of security use cases.

3.1.2 Later-stage Security Engineering

Efforts in this area focus on how to map security requirements (identified in the previous stage) on system design entities at design time and how to help in generating secure and security code specified. Below we summarize the key efforts in this area organized according to the approach used or the underlying software system architecture and technology used.

UMLsec [14-16] is one of the first model-driven security engineering efforts. UMLsec extends UML specification with a UML profile that provides stereotypes to be used in annotating system design elements with security intentions and requirements. UMLsec provides a comprehensive UML profile but it was developed mainly for use during the design phase. Moreover, UMLsec contains stereotypes for predefined security requirements (such as secrecy, secure dependency, critical, fair-exchange, no up-flow, no down-flow, guarded entity) to help in security analysis and security generation. UMLsec is supported with a formalized security analysis mechanism that takes the system models with the specified security annotations and performs model checking. UMLsec [17] has recently got a simplified extension to help in secure code generation.

SecureUML [18] provides UML-based language for modeling role-based access control (RBAC) policies and authorization constraints of the model-driven engineering approaches. This approach is still tightly coupled with system design models. SecureUML defines a set of vocabulary that represents RBAC concepts such as roles, role permissions and user-assigned roles.

Satoh et al. [19] provides end-to-end security through the adoption of model-driven security using the UML2.0 service profile. Security analysts add security intents (representing security patterns) as stereotypes for the UML service model. Then, this is used to guide the generation of the security policies. It also works on securing service composition using pattern-based by introducing rules to define the relationships among services using patterns. Shiroma et al [20] introduce a security engineering approach merging model driven security engineering with patterns-based security. The proposed approach works on system class diagrams as input along with the required security patterns. It uses model transformation techniques (mainly ATL - atlas transformation language) to update the system class diagrams with the suitable security patterns applied. This process can be repeated many times during the modeling phase. One point to be noticed is that the developers need to be aware of the order of security patterns to be applied (i.e. authentication then authorization, then…)

Delessy et al. [21] introduce a theoretical framework to align security patterns with modeling of SOA systems. The approach is based on a security patterns map divided into two groups: (i) abstraction patterns that deliver security for SOA without any implementation dependencies; and (ii) realization patterns that deliver security solutions for web services' implementation. It appends meta-models for the security patterns on the abstract and concrete levels of models. Thus, architects become able to develop their SOA models (platform independent) including security patterns attribute. Then generate the concrete models (platform dependent web services) including the realization security patterns. Similar work introduced by [22] to use security patterns in capturing security requirements and enforcement using patterns.

Hafner et al. [23] introduce the concept of security-as-a-service (SeAAS) where a set of key security controls are grouped and delivered as a service to be used by different web-based applications and services. It is based on outsourcing security tasks to be done by the SeAAS component. Security services are registered with SeAAS and then it becomes available for consumers and customers to access whenever needed. A key problem of the SeAAS is that it introduces a single point of failure and a bottleneck in the network. Moreover, it did not provide any interface where third-party security controls can implement to support integration with the SeAAS component. The SECTET project [24] focuses on the business-to- business collaborations (such as workflows) where security needs to be incorporated between both parties. The solution was to model security requirements (mainly RBAC policies) at high-level and merged with the business requirements using SECTET-PL [25]. These modeled security requirements are then used to automate the generation of implementation and configuration of the realization security services using WS- Security as the target applications are assumed to be SOA-oriented.

We have also determined different industrial security platforms that have been developed to help software engineers realizing security requirements through a set of provided security functions and mechanisms that the software engineers can select from. Microsoft has introduced more advanced extensible security model - Windows Identity Foundation (WIF) [26] to enable service providers delivering applications with extensible security. It requires service providers to use and implement certain interfaces in system implementation. The Java Spring framework has a security framework – Acegi [27]. It implements a set of security controls for identity management, authentication, and authorization. However, these platforms require developers' involvement in writing integration code between their applications and such security platforms. The resultant software systems are tightly coupled with these platforms' capabilities and mechanisms. Moreover, using different third-party security controls requires updating system source code to add necessary integration code.

3.2 Security Retrofitting

Although a lot of security engineering approaches and techniques do exist as we discussed in the last section, the efforts introduced in the area of security re-engineering and retrofitting are relatively limited. This comes, based on our understanding, from the assumption that security should not be considered as an

afterthought and should be considered from the early system development phases. Thus, research and industry efforts focus mainly on how to help software and security engineers in capturing and documenting security in system design artifacts and how to enforce using model-driven engineering approaches. Security maintenance is implicitly supported throughout updating design time system or security models. In the real world, system delivery plans are dominated by developing business features that should be delivered. This leads to systems that miss customers expected or required security capabilities. These existing legacy systems lack models (either system or security or both) that could be used to conduct the reengineering process. The maintenance or reengineering of such systems is hardly supported by existing security (re)engineering approaches.

Research efforts in the security retrofitting area focus on how to update software systems in order to extend their security capabilities or mitigate security issues. Abdulkarim et al [28] discussed the limitations and drawbacks of applying the security retrofitting techniques including cost and time problems, technicality problems, issues related to the software architecture and design security flaws. Hafiz et al. [29, 30] propose a security on demand approach, which is based on a developed catalog of security-oriented program transformations to extend or retrofit system security with new security patterns that have been proved to be effective and efficient in mitigating specific system security vulnerabilities. These program transformations include adding policy enforcement point, single access point, authentication enforcer, perimeter filter, decorated filter and more. A key problem with this approach is that it depends on predefined transformations that are hard to extend especially by software engineers.

Ganapathy et al. [31, 32] propose an approach to retrofit legacy systems with authorization security policies. They used concept analysis techniques (locating system entities using certain signatures) to find fingerprints of security-sensitive operations performed by system under analysis. Fingerprints are defined in terms of data structures (such as window, client, input, Event, Font) that we would like to secure their access and the set of APIs that represent the security sensitive operations. The results represent a set of candidate joinpoints where we can operate the well-known “reference monitor” authorization mechanism.

Padraig et al. [33] present a practical tool to inject security features that defend against low-level software attacks into system binaries. The authors focus on cases where the system source code is not available to system customers. The proposed approach focuses on handling buffer overflow related attacks for both memory heap and stack.

Welch et al. [34] introduce a security reengineering approach based on java reflection concept. Their security reengineering approach is based on introducing three meta-objects that are responsible for authentication, authorization, and communication confidentiality. These meta-objects are weaved with the system objects using java reflection. However, this approach focuses only on adding predefined types of security attributes and do not address modifying systems to block reported security vulnerabilities.

3.3 Adaptive Application Security

Several research efforts target to enable systems to adapt their security capabilities at runtime. Elkhodary et al. [35] survey adaptive security systems. Extensible Security Infrastructure [36] is a framework that enables systems to support adaptive authorization enforcement through updating in memory authorization policy objects with new low level C code policies. It requires developing wrappers for every system resource that catch calls to such resource and check authorization policies. Strata Security API [37] where systems are hosted on a strata virtual machine which enables interception of system execution at instruction level based on user security policies. The framework does not support securing distributed systems and it focuses on low level policies specified in C code.

The SERENITY project [38-40] enables provisioning of appropriate security and dependability mechanisms for Ambient Intelligence (AI) systems at runtime. The SERENITY framework supports:

definition of security requirements in order to enable a requirements-driven selection of appropriate security mechanisms within integration schemes at run-time; provide mechanisms for monitoring security at run-time and dynamically react to threats, breaches of security, or context changes; and integrating security solutions, monitoring, and reaction mechanisms in a common framework. SERENITY attributes are specified on system components at design time. At runtime, the framework links serenity-aware systems to the appropriate security and dependability patterns. *SERENITY does not support dynamic or runtime adaptation for new unanticipated security requirements neither adding security to system entities that was not secured before and become critical points*.

Morin et al. [41] propose a security-driven and model-based dynamic adaptation approach to adapt applications' enforced access control policies in accordance to changes in application context – i.e. applying context-aware access control policies. Engineers define security policies that take into consideration context information. Whenever the system context changes, the proposed approach updates the system architecture to enforce the suitable security policies. *The key limitation of this work is that it focuses mainly on access control policies and requires design time preparation of the software.*

Mouelhi et al. [41] introduce a model-driven security engineering approach to specify and enforce system access control policies at design time based on AOP-static weaving. These adaptive approaches require design time preparation (to manually write integration code or to use specific platform or architecture). *They also support only limited security objectives, such as access control. Unanticipated security requirements are not supported. No validation that the target system (after adaptation) correctly enforces security as specified.*

Eric et al [42] introduce a more comprehensive survey of efforts in the area of self-protecting software systems. They have also outlined the key research gaps in the existing techniques. This includes: (i) lack of comprehensive self-protecting systems either from the monitoring, planning, execution perspective, or from the software stack perspective – i.e. host, network and software; (ii) lack of an integrated solution that supports both design-time and runtime security, (iii) support of more security adaptation patterns. Our approach focus is the first problem, which is to extend a given software system with necessary security monitors (using user defined metrics and properties), security analysis (using formalized vulnerability signatures), planning (using models for manual adaptation and rules for automated adaptation), and execution (using aspect-oriented programming). Furthermore, we generate a set of integration test cases to verify that the specified adaptations (realized by security controls' integration with the software system) are functioning as expected. The big picture of our approach is available in [43]. In this chapter we focus mainly on how adaptation can be specified (manually/automatically) and how such adaptations can be realized.

4. Run-time Security Adaptation

We identified two potential types of security adaptation: *manual adaptation*: usually triggered manually by security engineers/administrators based on change in security goals, security threats and risks; and *automated adaptation*: triggered automatically based on specified adaptation rules fired when a certain metric exceeds a user-defined threshold, a property is violated, or a new vulnerability was reported.

Our approach, outlined in Figure 2, is based on externalizing the software security capabilities from the software so that we can easily change such security capabilities without the need to change the software itself. At the same time being able to integrate (inject) such new capabilities within the software at any arbitrary system entity. This is abstracted to end users by a set of Domain-Specific Visual Languages (DSVLs) at different levels of abstraction to help them describe their security needs, software details, and mapping security to system entities at the right level of abstraction for different stakeholders. This helps in speeding up the software change time to ad-hoc security needs. The security

vulnerability analysis is based on formalized signatures that describe bad code smells we need to look for in a given system. The same idea is used in the security monitoring component. All these inputs (requirements for adaptation) are realized/executed using the same execution component (MDSE@R, Security Engineering at Runtime).

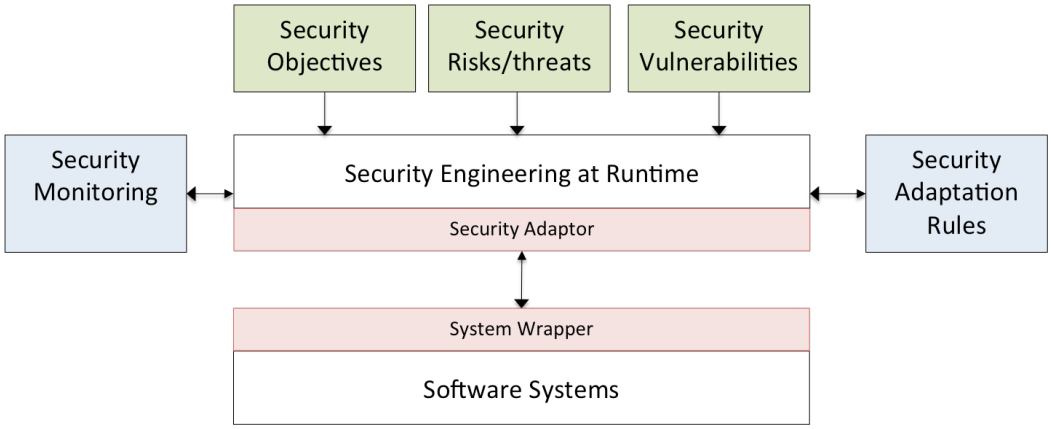

Figure 2. Block diagram of our adaptive security approach

4.1 Supporting Manual Adaptation using MDSE@R

The MDSE@R (model-driven security engineering at runtime) approach [44, 45] targets externalizing all security engineering activities so we can define and change system security at any time, while being able to integrate these new security capabilities on the system at runtime. MDSE@R is based on two key concepts: (i) Model-driven Engineering (MDE), using DSVL models at different levels of abstraction to describe system and security details; and (ii) Aspect-oriented Programming, that enables dynamic runtime weaving of interceptors and system code based on configuration files that specify the required security point-cuts in the system. Figure 3 shows an overview of how to apply MDSE@R in engineering security for a given system at runtime, as discussed here.

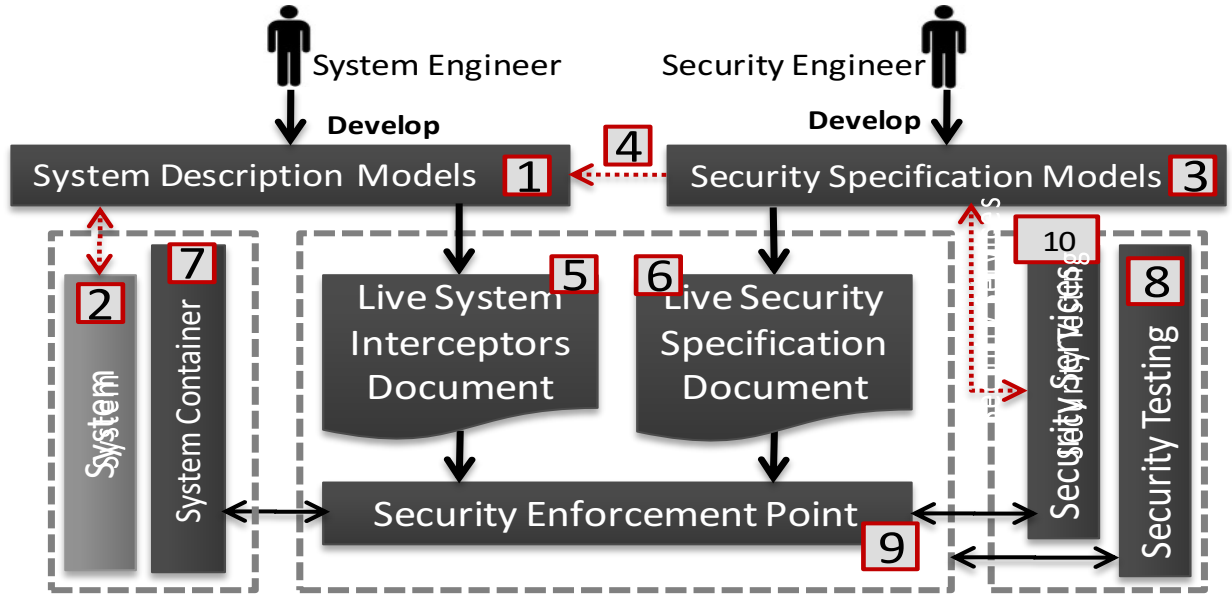

Figure 3. Security engineering at runtime

Build System Description Model (SDM): A detailed system description model (Figure 3-1, see Figure 9 for an example) made up of a set of models delivered by the system provider. This describes various details of the target software application. Our system description models include: system features (using use case diagrams), system architecture (using component diagrams), system classes' model (using class diagrams), system behavior model (using sequence diagrams), system deployment (using deployment diagrams), and system context (using component diagrams). We have selected these models as they cover all system perspectives that may be required in order to specify system security. The use of many of these sub-models is optional. It depends on how many of the system details the system provider exposes to their customers and how many details customers' security engineers will need in enforcing the required security on the target system. Security engineers may be interested in specifying security on system entities (using system components and/or classes models), on system status (using system behavior model), on system hosts (using system deployment model), or external system interactions (using system context model). Moreover, system customers can specify security on coarse-grained level (using system component model), or on fine- grained level (using system class models). The system description models can be synchronized with the running system instance using models@runtime synchronization techniques [25, 26], or manually by the system provider. Some of such system description information can be reverse-engineered, if not available, from the target system (Figure 3-2).

Build Security Specification Model (SSM): A set of models developed and managed by security engineers (Figure 3-3) to specify the security needs that must be satisfied in the target system. They include a set of sub-models that capture the details required during the security engineering process including: security goals and objectives, security risks and threats, security requirements, security architecture for the operational environment, and security controls to be enforced. These models deliver different levels of abstractions and enable separation of concerns between customer stakeholders including business owners, security analysts, security architects and implementers. The key mandatory model in the security specification models set is a security controls model. This is required in generating interceptors and security aspect code.

System-Security Models Weaving: A many-to-many mapping between the system description models and security specification models is developed by the customer security engineers (Figure 3-4). One or more security concepts (security objective, security requirement and/or security control) is mapped to one or more system model entities (system-level, feature-level, component-level, class-level and/or method-level entities). Mapping a security concept on a higher level system entity implies a delegation to the underlying levels. Whenever a security specification is mapped to a system feature, this implies that the same security specification is mapped on the feature related components, classes, and methods.

The few steps discussed so far helps in addressing the planning phase in security adaption. New security requirements (objectives, risks, etc.) can easily be reflected on the security specification model described above. The next steps related to enforcing (executing) the specified security, and are automated by MDSE@R without any involvement from the security or system engineers. Whenever a mapping is defined or updated between a security specification model and a system description model, the underlying MDSE@R framework propagates such changes as follows:

Update Live System Interceptors' Document (Figure 3-5) – this maintains a list of pointcuts where security controls should be weaved/integrated with the target software application entry points. This document is updated based on the modeled security specifications and the corresponding system entities where security should be applied. *Update a Live Security Specification Document* (Figure 3-6) - this maintains a list of security controls to be applied at every pointcut defined in the system interceptors'

document. *Update the System Container* (Figure 3-7) - this is responsible for injecting interceptors defined in the system interceptors' document into the target system at runtime using dynamic weaving AOP. Any call to a method, with a matching in the interceptors' document, will be intercepted and delegated to a central security enforcement point. *Test Current System Security* (Figure 3-8) – this validates that the target system is currently enforcing the specified security levels. The security-testing component makes sure that the intended security is correctly integrated with the target application at run-time. MDSE@R generates and fires a set of security integration test cases. This is done before MDSE@R gives confirmation to security engineers that required security is now enforced. *Security Enforcement Point* (Figure 3-9) – this acts as a bridge between the target system (system container) and the security controls that deliver the required security. The security enforcement point uses the live security specification document to determine, and initiate, security control to be enforced on a given, intercepted, request. S*ecurity Services* (Figure 3-10) are the application security controls (deployed in the system operational environment) that are integrated with the security enforcement point. This enables the security enforcement point to communicate with these services via APIs implemented by each service.

Thus, MDSE@R covers manually adaptation scenarios. A given set of security objectives and requirements are reflected on the security specification model, and MDSE@R will make sure to automatically inject (or may be leave out) these security requirements as needed. For legacy systems, this might seem infeasible, but we have used static aspect oriented to modify system binaries and add calls to our security enforcement point.

4.2 Automated Adaptation using Vulnerability Analysis, and Mitigation

Another key trigger for security adaption is the discovery of a new vulnerability in the software. In our approach [46-48], we assume that this requires automated adaptation of the enforced security to (virtually) patch the reported security until the software vendor develops a real patch. In this section we discus how we can do the vulnerability analysis, and then using a set of rules to come up with necessary adaptation actions to block such vulnerability. Figure 4 summarizes the interactions between the vulnerability analysis component, security mitigation component, and the software. Our vulnerability analysis approach depends on a formalized vulnerability definition schema that covers many concepts of software security weaknesses (flaws) such as vulnerability signature - what are the key things in the software when found, it means that the system suffers from such vulnerability, and mitigation actions – what adaptation we need to apply to patch the vulnerability.

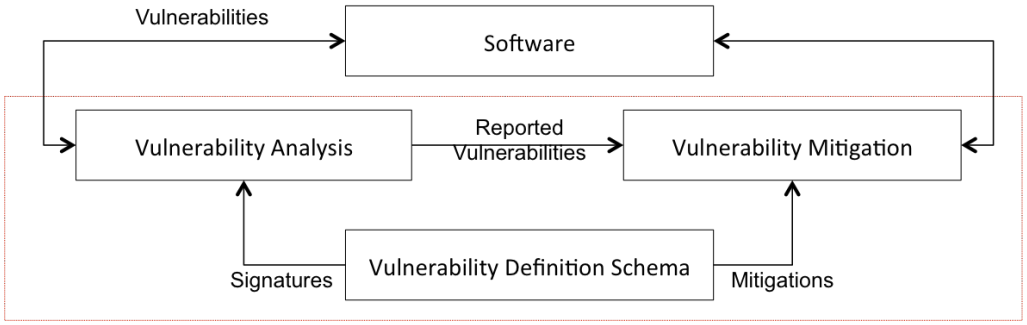

Figure 4. Automated vulnerability analysis and mitigation

Formalizing vulnerability signatures helps automating the vulnerability analysis process. Ideally, a formal vulnerability signature should be specified on an abstract level far from the source code and programming language details, enabling locating of possible vulnerability instances in different programs written in different programming languages. We use Object Constraint Language (OCL) as a well-known,

extensible, and formal language to specify semantic rather than syntactical signatures of security weaknesses. To support specifying and validating OCL-based vulnerabilities' signatures, we have developed a system-description meta-model, shown in Figure 5. This model is inspired from our analysis of the nature of the existing security vulnerabilities. It captures the main entities in any object-oriented program and relationships between them including components, classes, instances, inputs, input sources, output, output targets, methods, method bodies, statements e.g. if-else statements, loops, new objects, etc. Each entity has a set of attributes such as method name, accessibility, variable name, variable type, method call name. This model helps conducting semantic analysis of the specified vulnerability signatures. Table 1 shows examples of vulnerability signatures specified in OCL and using our system description model.

Table 1. Example vulnerability signatures

Vuln.	Vulnerability Signature
SQLI	Context Method Inv SQLICheck: self.Statements->exists(S | S.StatementType = 'MethodInvocation' and S.MethodName = 'ExecuteSQL' and S.Parameters.exists(P | self.IsTainted(P.ParameterName) = true)
XSS	Context Method Inv SQLICheck: self.Exists(S | S.StatementType = 'Assignment' and S.RightPart.Contains(InputSource) and S.LeftPart.Contains(OutputTarget))
Authn. Bypass	Context Method Inv SQLICheck: self.IsPublic == true and self->Exists(S | S.StatementType = 'MethodInvocation' and S.IsAuthenitcationFn == true and S.Parent == IFElseStmt and S.Parent.Condition.Contains(InputSource))
Improper Authz.	Context Method Inv SQLICheck: self.IsPublic == true and self.Contains(S| S.Exists(X| X.StatementType = 'InputSource' and X.IsSanitized = false or X.IsAuthorized == False)

SQLI: any method statement *"S'* of type "*MethodInvocation*" where the callee function is *"ExecuteQuery"* and one of the *parameters* passed to it, is assigned to *"identifier"* coming from one of the input sources. Taint analysis "*IsTainted*" can be defined as an OCL function that adds every variable assigned to a user input parameter to a suspected list.

XSS Signature: any method statement *"S"* of type assignment statement where left part is of type "*output target*" e.g. text, label, grid, etc. and right part uses input from the input sources or tainted identifier as just discussed.

Authn. Bypass: any public method that has statement "*S*" of type "*MethodInvocation*" where the callee method is marked as Authentication function while this method call can be skipped using user input as part of the bypassing condition.

Improper Authz.: any public method that has statement "S" that uses input data X without being sanitized, authorized.

OCL-based Vulnerability Analyzer

Given that vulnerability signatures become now formally specified using OCL, the static vulnerability analysis component simply traverses the given program looking for code snippets with matches to the given vulnerabilities' signatures.

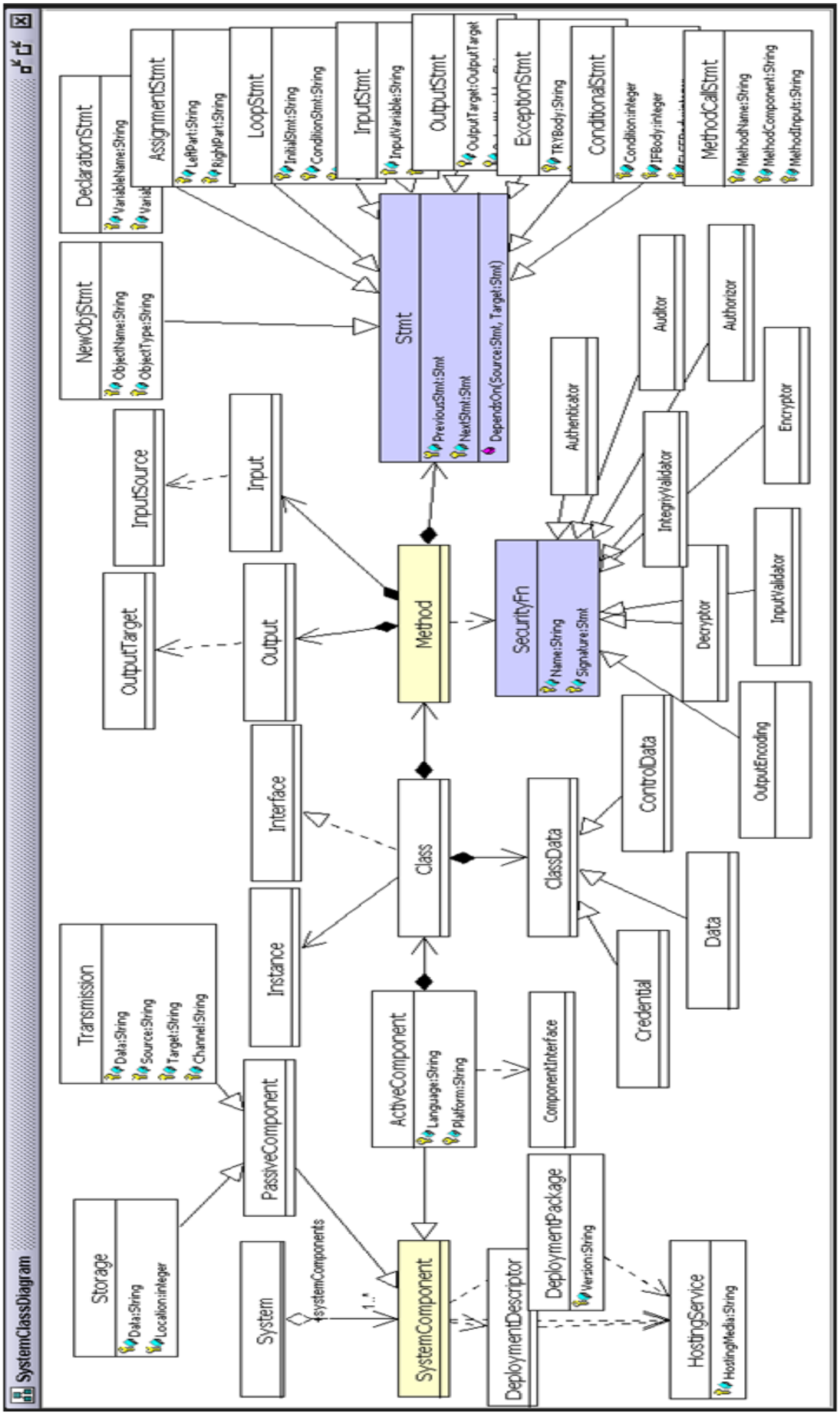

Figure 5. Software description meta-model

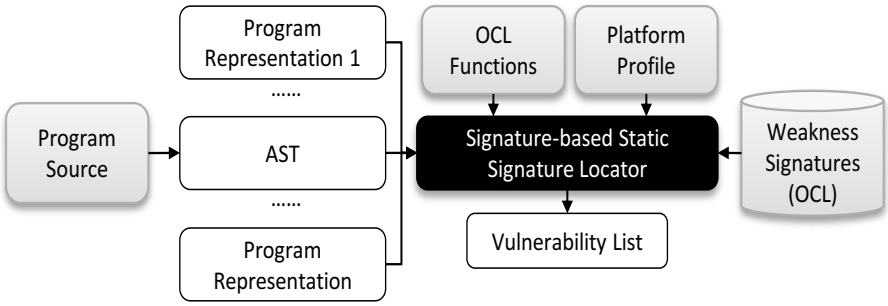

Figure 6. OCL-based vulnerability analysis

The architecture of our formal and scalable static vulnerability analysis component, as shown in Figure 6, is based on our formalized vulnerability signature concept.

Program Source Code: we should have source code or binaries (dlls, exes - de-compilation is used to reverse engineer source code) of the application to be analyzed.

Abstract Program Representation: Source code is transformed into an abstract syntax tree (AST) representation. This abstracts language-specific source code details away from specific language constructs. Extracting source code AST requires using different language parsers (currently support C++, VB.Net and C#). Then, we perform more abstract transforming from AST to system description model that conforms to the model.

OCL Functions: represent a library of predefined functions that can be used in specifying vulnerability signatures and in identifying matches to these signatures. This includes control flow, data flow, string patterns, program taint analysis, etc.

Signature Locator: This is the main component in our vulnerability analysis tool. It receives the abstract service/application model and outputs the list of discovered vulnerabilities in the given system along with their locations in code. At analysis time, it loads the platform (C#, VB, PHP) profile based on the details of the program under analysis. Then, it loads the existing weaknesses defined in the weaknesses' signatures database, based on the target program platform/language. The signature locator transforms these signatures into C# methods that check different program entities based on the specified vulnerability signature. We use Application Vulnerability Description Language (AVDL) to represent the identified vulnerabilities in XML format to support interoperability with existing vulnerability databases such as National Vulnerabilities Database (NVD).

Vulnerability Mitigation

Discovered application/service security vulnerabilities can be mitigated in different approaches including: modifying application source code to block the identified problems (patches); however, this solution will be hard to approach in public accessible software systems – e.g. cloud systems - as it may take long time to deliver patched version. A quick solution is to use Web application firewall (WAF) to filter requests/responses that exploit such vulnerabilities; however, WAF has many limitations including it does not help in output validation, cryptography storage, and mitigating improper authorization.

Table 2. Example vulnerability mitigation rules/actions

Vul.	Security Control	Entity Level
SQLI	Input sanitization	Method level
XSS	Input encoding	Component level
Authn. Bypass	WAF	Component level
Improper Authz.	Authorization	Method Level

Our approach supports integrating different security controls including identity management, authentication controls, authorization controls, input validation, output encoding, WAF, cryptography controls, etc. In our approach, each vulnerability mitigation action specifies a security control type/family to be used in mitigating the related vulnerability, its required configurations, and application/service entity where the security control will be integrated with (hosting service – webserver or operating system, components, classes, and methods). Thus, a reported SQLI vulnerability in a method (M) that belongs to component (C) can be mitigated by adding input sanitization control (Z) on component (C) that removes SQL keyword from every single request to the method (M). In Table 2, we show examples of mitigation actions for some of the known security vulnerabilities. These actions should be specified in XML and included as a part of the formalized vulnerability definition.

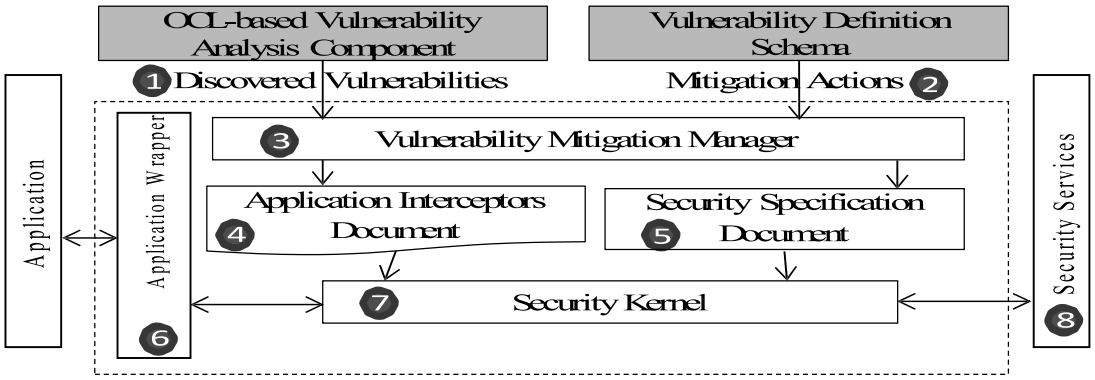

Figure 7. Vulnerability mitigation component

Vulnerability Mitigation Component

The analysis component outputs a list of the discovered vulnerabilities in the software system (Figure 7-1). Each entry in this list has a service/application vulnerable entity (method, class, or component) along with the list of discovered vulnerabilities in this entity. Given this list of vulnerabilities, the security vulnerability mitigation manager queries the vulnerability definition schema database (Figure 7-2) to retrieve the appropriate actions to be taken in order to mitigate each of such reported vulnerabilities. Examples of the retrieved actions are shown in Table 2. Using these two lists (vulnerable software entities and mitigation actions), the vulnerability mitigation manager (Figure 7-3) decides the patching level (component level, class level, or method level) using e.g. HttpModules, object interceptor using dependency injection, or method level interception using dynamic weaving AOP respectively. The rest of the steps to enforce the right security control at the right place are as described in the MDSE@R section.

5 USAGE EXAMPLE

To demonstrate the capabilities of our new MDSE@R security engineering approach we revisit our example discussed in section 2, the ERP system “Galactic” developed by SwinSoft and procured by Swinburne and SwinMarket. The two customers using the Galactic ERP system have their own distinct security requirements to be enforced on each of their Galactic ERP application instances. We illustrate this security engineering scenario using screen dumps from our prototype tool.

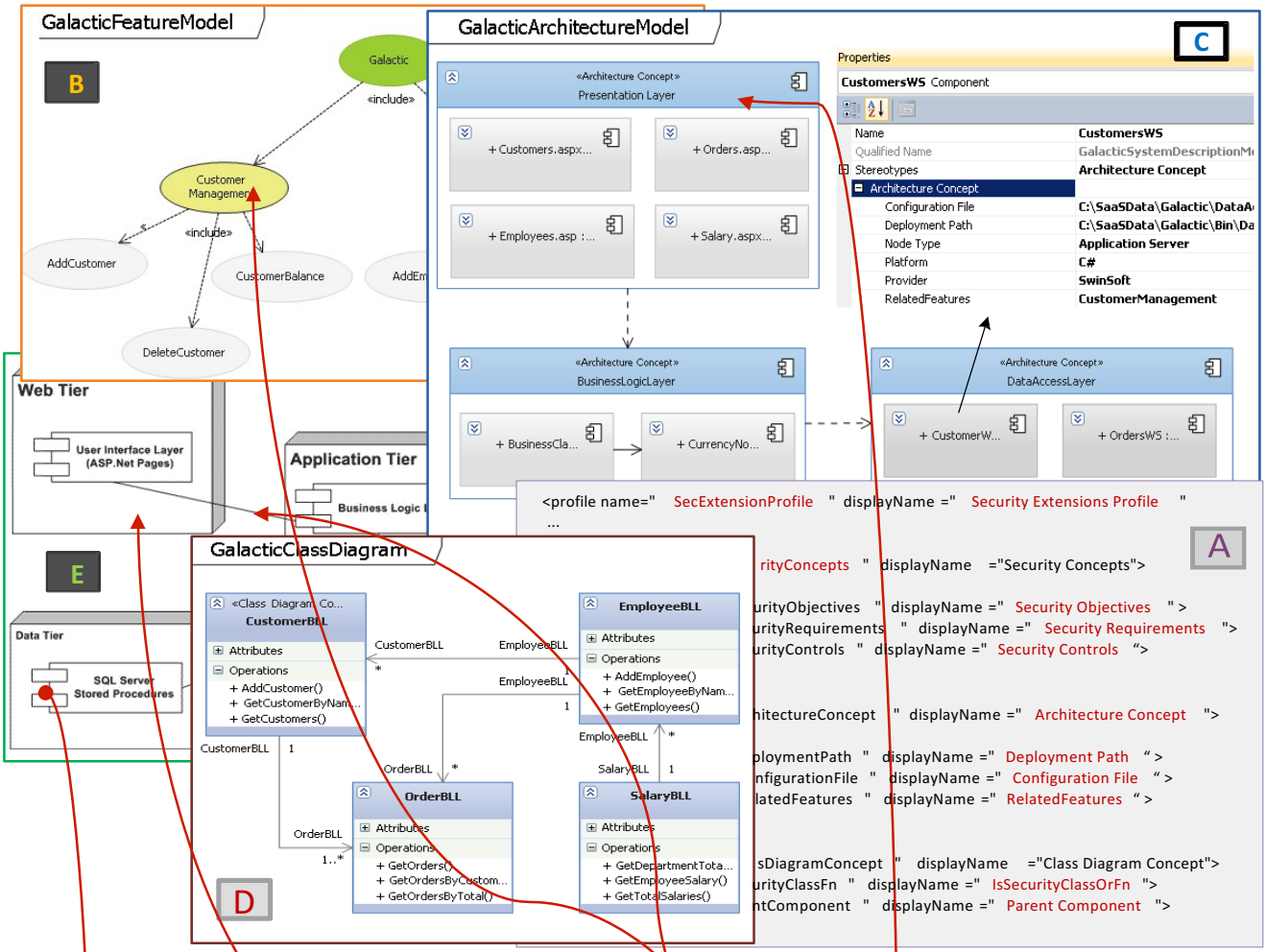

Figure 8. Examples of Galactic software definition model

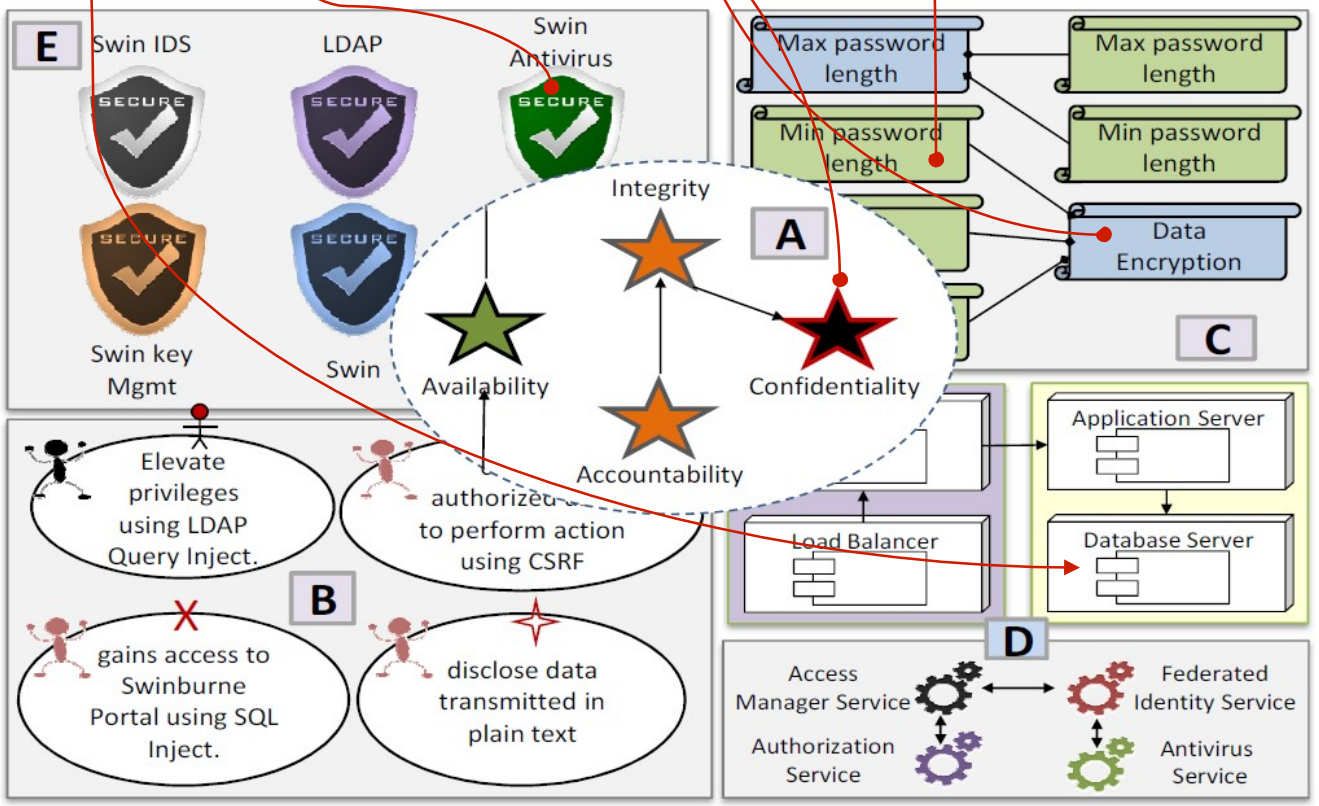

Figure 9. Examples of Swinburne security specification model

Task 1- Model Galactic System Description - one time task

This task is done during or after the system is developed. SwinSoft decides the level of application details to provide to its customers in the Galactic system model. Figure 8 shows that SwinSoft provides its customers with description of system features including customer, employee and order management features (Figure 8-b), system architecture including presentation, business logic layer and data access layer (Figure 8-c), system classes including CustomerBLL, OrderBLL, EmployeeBLL (Figure 8-d), and system deployment including web tier, application tier, and data tier (Figure 8-e). SwinSoft uses the provided UML profile (Figure 8-a) to specify the dependences and relations between system features and components, and components and their classes. Figure 8-a shows the UML profile we built to extend UML with security properties (what security controls/requirements/objectives) are mapped to a given system entity; and to store the tracability information between different system artifacts – e.g. system features to realization components, components to classes, etc.

Task2 –Model Swinburne Security Needs

This step is conducted by Swinburne and SwinMarket security engineers during their repetitive security management process. In our scenario, Swinburne security engineers document Swinburne security objectives that must be satisfied by Galactic system. This is done using a high-level security objectives model (Figure 9-a). This model can be revisited at any time to incorporate changing Swinburne security objectives. Security engineers refine these security objectives in terms of security requirements that must be enforced on the Galactic system, developing a security requirements model. This model keeps track of the detailed security requirements and their link back to the high level security objectives (Figure 9-b). This example shows user authentication requirements to be enforced on the Galactic application and its hosting server.

Swinburne security engineers next develop a detailed security architecture including other existing IT systems. This security architecture (Figure 9-c) identifies the different security zones (Big Boxes) that cover Swinburne network and the allocation of IT systems, including Galactic, as either one unit or in terms of system components according to the Galactic deployment model. The security architecture also shows the security services, security mechanisms and standards that should be deployed. Swinburne security engineers finally specify the security controls (i.e. the real implementations) for the security services modeled in the security architecture model (Figure 9-d). This includes SwinIPS Host Intrusion Prevention System, LDAP access control and SwinAntivirus. These are used to realize the security requirements and security architecture as previously specified. Each security specification model maintains traceability information to parent models' entities. In Figure 9-d, we specify that LDAP "realizes" the *AuthenticateUser* requirement. Whenever MDSE@R finds a system entity with a mapped security requirement *AuthenticateUser* it adds LDAP as its realization control i.e. an LDAP authentication check is run before the entity is used e.g. before a method or web service is called or module loaded.

Task 3 – System - Security Weaving

After developing the system SDMs – done by SwinSoft, and the security SSMs – done by Swinburne security engineers, the Swinburne security engineers map security attributes (in terms of objectives, requirements and controls) to Galactic system specification details (in terms of features, components, classes). This is achieved by drag and drop of security attributes to system features in our toolset. Any system feature, structure or behaviour can dynamically and at runtime reflect different levels of security based on the currently mapped security attributes on it.

Figure 9-e shows a sample of the security objectives, requirements and controls mapped to customerBLL class. In this example the security engineer has specified that the *AuthenticateUser* security requirement

should be enforced on the CustomerBLL class (1). Such a requirement is achieved using LDAP control (3). Moreover, they have specified Forms-based authentication on the GetCustomers method (2). This means that a request to a method in the CustomerBLL class will be authenticated by the caller's Windows identity, but a request to the GetCustomers method will be authenticated with a Forms-based identity. MDSE@R uses security attributes mapped to system entities to generate the full set of methods' call interceptors and entities' required security controls, as shown in Figure 13.

Task 4 - Galactic security testing

Once security has been specified and interceptors and configurations are generated, MDSE@R makes sure that the system is correctly enforcing security as specified. MDSE@R generates and fires a set of required security integration test cases. Our test case generator uses the system interceptors and security specification documents to generate a set of test cases for each method listed in the interception document. The generated test case contains a set of security assertions (one for each security property specified on a given system entry). During the firing phase, the security enforcement point is instrumented with logging transactions to reflect the calling method, called security control, and the returned values. Security engineers should check the security test cases execution log, as shown in Figure 10, to make sure that no errors introduced during the security integration with Galactic entities. Figure 11 shows a sample run of Galactic after weaving Forms-based authentication control when calling GetCustomers method.

Test Case Name	Message
AuthenticationTesting	Authenitcation Control 'Forms-based authentication' is plugged-in
AuthorizationTesting	Authorization Control is not plugged-in

Figure 10. Sample test cases firing log

Figure 11. Testing Galactic with injected form-based authentication

SwinMarket security engineers go through the same process as Swinburne did when specifying their security requirements. However, SwinMarket specifies their requirements, context, security controls, and IT applications. This results in quite different generated security enforcement controls.

Both Swinburne and SwinMarket security engineers can modify the security specifications while their Galactic applications are in use. MDSE@R framework updates interceptors in the target systems and enforces changes to the security specification for each system as required. For example, the Swinburne Galactic security model can be updated with a Shibboleth single sign-on security authentication component. The updated interceptors and security specification are applied to the running Galactic deployment, which then enforces this authentication protocol instead of the Forms approach as above.

Task5 - Galactic Continuous Vulnerability Analysis and Mitigation

We have applied the vulnerability analysis tool on Galactic ERP system (and many other applications), and using the mitigation actions, summarized in Table 2. Table 3 shows the number of reported vulnerability instances grouped by vulnerability type. We applied the vulnerability analysis incrementally – i.e. SQLI analysis, then XSS, and so on. For each of these reported vulnerabilities, we have checked that the proper security control(s) was integrated successfully as specified in the actions table, Table 2, and that the reported vulnerability is no longer exploitable.

Table 3. Number of reported vulnerabilities

SQLI			XSS			Authn. Bypass			Improper Authz.		
TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
2	0	0	3	1	1	4	0	0	2	1	0
(TP: True Positives, FP: False Positives, FN: False Negative)											

6 DISCUSSION

Our approach is based on promoting security engineering from design time to runtime by externalizing security engineering activities including capturing objectives, requirements controls, and realization from the target system implementation. This permits both security to be enforced and critical points to secure to evolve at runtime (supporting adaptive security at runtime). Using a common security interface helps integrating different security controls without a need to develop new system-security control connectors. Moreover, a key benefit reaped from MDSE@R approach is to the support model-based security management. Enterprise-wide security requirements, architecture and controls are maintained and enforced through a centralized security specification model instead of low level scattered configurations and code that lack consistency and are difficult to modify. Thus any update to the enterprise security model will be reflected on all IT systems that use our security- engineering platform. This is another key issue in environments where multiple applications must enforce the same security requirements. Having one place to manage security reduces the probability of errors, delays, and inconsistencies. Moreover, automating the propagation of security changes to underlying systems simplifies the enterprise security management process.

One may argue that MDSE@R may lead to a more vulnerable system as we did not consider security engineering during design time. Our argument is that at design time we need to think more about building secure systems. However, given that we continue to discover a lot of vulnerabilities in systems even those with design time security consideration, we have supported our approach with both continuous vulnerability analysis and mitigation. The vulnerability analysis component is based on formal vulnerability definition schema that includes vulnerability signature and mitigation actions. Using abstract representation instead of source code helps to generalize/abstract our analysis from programming language and platform details. It also helps to make the approach more scalable for larger applications.

Aspect-oriented programming (AOP) is always suspected as a source of potential security attacks [49] given that a malicious user might be able to plug vulnerable aspect code that can alter the innovation parameter, redirect the request or discard it completely. Moreover, using AOP to integrate security aspects as a cross cutting concern is also questionable given that these security aspects could lead to inconsistent update of system properties. However, the authors did not stop using AOP to develop their permission model, they have suggested a set of recommendations when using AOP such as dealing woven code, define appropriate language extension, and analyse weaver components for potential flaws. To avoid such issues, we disable the write permission on the interceptor document and security handlers. Thus only our platform will have write access to these documents.

Security adaptation of existing software systems: the security engineering of existing services (extending system security capabilities) has three possible scenarios: (i) systems that already have their SDMs, we can use MDSE@R directly to specify and enforce security at runtime; (ii) systems without SDMs, we reverse engineer parts of system models (specifically the class diagram) using MDSE@R. Then we can use MDSE@R to engineer required system security. Finally, systems with built-in security, in this case we can use MDSE@R to add new security capabilities only. MDSE cannot help modifying or disabling existing security. We have built another tool (re-aspects) to leave out existing built-in security methods and partial code using modified AOP techniques.

Security and performance trade-off: The selection of the level of details to apply security on depends on the criticality of the system. In some situations like web applications, we may intercept calls to the presentation layer only (webserver) while considering the other layers secured by default (not publicly accessible). In other cases, such as integration with a certain web service or using third party component, we may need to have security enforced at the method level (for certain methods only). Security and performance trade-off is another dilemma to consider. The more security validations and checks the more resources required. MDSE@R enables adding security only whenever needed. Thus, when we believe that the system operational environment we can reduce the security controls required which improves system performance and vice-versa. So the trade-off between performance and security is now at the hand of system/security admins.

Hybrid vulnerability analysis: From our experience in developing signatures of the OWASP Top10 vulnerabilities (most frequently reported vulnerabilities) we determined that: **(i)** the accuracy of our vulnerability analysis depends heavily on the accuracy of the specified vulnerability signatures; **(ii)** it is better to use dynamic analysis tools with certain vulnerabilities, such as Cross site reference forgery (CSRF), because these vulnerabilities can be handled by the web server. This means static analysis may result in high FP, if used; **(iii)** some vulnerabilities can be easily identified and located by static analysis such as SQLI and XSS vulnerabilities; **(iv)** some vulnerabilities such as DOM-based SQL and XSS vulnerabilities need a collaborating static and dynamic analysis to locate them. We believe that combining static and dynamic analysis is needed to increase the precision and recall rates. Static analysis approaches usually result in high false positives as they work on source code level – i.e. the vulnerability may be addressed on the component or the application level. Employing dynamic vulnerability analysis can solve this problem. However, dynamic vulnerability analysis approaches cannot help locating specific code snippets where vulnerabilities exist. Moreover, they do not help testing code coverage by generating all possible test cases.

Virtual patching trade-off: From our experiments in the mitigation actions and security controls integrations, we found that although the use of web application firewalls is a straightforward solution, it is not always feasible to use WAF to block all discovered vulnerabilities. The selection of the entity level to apply security controls on (application, component, method, etc.) impacts the application performance – i.e. instead of securing only vulnerable methods, we intercept and secure (add more calls) the whole component requests. A key point that worth mentioning is that the administration of security controls should be managed by the service/cloud provider admins. We focus on integrating controls within vulnerable entities. Our vulnerability mitigation component works online without a need for manual integration with the applications/services under its management. The overhead added by the mitigation action can be easily saved if the service developers worked out a new service patch. In this case, the vulnerability analysis component will not report such vulnerability. Thus, the mitigation component will not inject security controls.

Pros & cons: The key benefits of our adaptation approach are: (i) we support both manual security adaptation and rule-based adaptation. Most of the existing efforts either focus on engineering systems to support adaptive-ness with either intensive development required, or limiting the approach to specific security properties – e.g. access control; (ii) Our approach also takes into consideration different sources of adaptation including: new security requirements, current system status (using security monitors), and/or reported security vulnerabilities. Most of the existing efforts consider only one source: either new security requirements or monitored system status but not reported vulnerabilities; and (iii) we adopt security

externalization and MDE techniques, which make it easier to change system security capabilities whenever needed and at system, component, an method levels based on user experience and needs. The security model itself can be shared between different systems. Thus, an enterprise security model can be easily managed.

7 Chapter Summary

In this chapter we discussed our adaptive security engineering approach, which enable adapting software security capabilities at runtime based on the new security objectives, risks/threats, requirements as well as the newly reported vulnerabilities. We categorize the source of adaptation in terms of manual adaptation (managed by end users), and automated adaption (automatically triggered by the platform). The platform makes use of the formal vulnerability definition schema, the formal signature-based security analysis, externalization of security engineering using aspect-oriented programming (AOP), and model-driven engineering techniques.

Appendix A – Platform Implementation

The architecture of our approach is aggregate of two key components: the security engineering at runtime (MDSE@R) and the security vulnerability analysis. Both of them are end-user oriented – i.e. both depend on end user specifications in terms of security objectives, requirements, controls, properties, vulnerabilities and mitigation action. Both components are discussed below in more details.

MDSE@R: Model-driven Security Engineering At Runtime

The architecture of the MDSE@R platform is shown in Figure 12. It consists of a system description modelling tool (1), a security specification modelling tool (2), a repository for the system and security models (3), a library of registered security controls and extensible security patterns that can be used by security engineers in enforcing their security needs (4), a system container that manages system execution and intercepts requests and function calls for system entry points at runtime (5), and a security test case generator (6) that is used to test the integration of configured application with required security controls.

The System description modeller (1) was developed as an extension of Microsoft VS 2010 modeller with an UML profile to enable system engineers modelling their systems' details with different perspectives including system features, components, deployment, and classes. The UML profile defines stereotypes and attributes to maintain the track back and foreword relations between entities from different models. Moreover, a set of security attributes to maintain the security concepts (objectives, requirements and controls) mapped to system entities. The minimum level of details expected from the system provider is the system deployment model. MDSE@R uses this model to reverse engineer system classes using .Net Reflections.

The *security specification modeller tool* (2) is a Visual Studio 2010 plug-in. It enables application customers, represented by their security engineers, to specify the security attributes and capabilities that must be enforced on the system and/or its operational environment. The security modeller delivers a set of security DSVLs. The security-objectives DSVL captures customer's security objectives and the relationships between them. Each objective has a criticality level and the defence strategy to be followed: preventive, detective or recovery. The Security requirements DSVL captures customer's security requirements and relationships between requirements including composition and referencing relations. The Security Architecture DSVL captures security architectures and designs of the customer operational environment in terms of security zones and security level for each zone; security objectives, requirements and controls to be enforced in each layer; components and systems to be hosted in each layer; security services, mechanisms and standards to be deployed in each layer or referenced from other layers. The

security controls DSVL captures details of security controls that are registered and deployed in the customer environment and relationships between these and the security requirements they cover. The system models, security models, interception documents, and security specification documents are maintained under one repository (3). We use Visual Studio T4 Templates and code generation language to generate these documents from the software and security specification models and mapping between both sets of models. T4 templates are a mixture of text blocks and control logic that can generate a text file. The control logic is written as program code in C#.

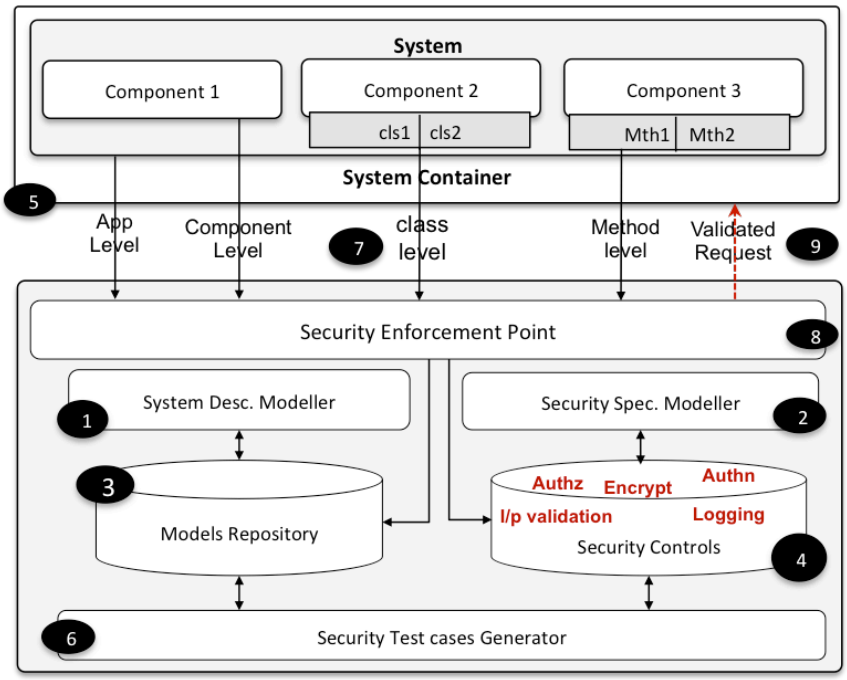

Figure 12. MDSE@R architecture

The *Security Controls Database* is a library of available and registered security patterns and controls. It can be extended by the system providers or by a third party security provider. Security controls implement certain APIs as defined by the security enforcement point in order to be able to integrate with target security control systems. Having a single enforcement point with a predefined interface for each security control category enables security providers to integrate with systems without having to redevelop adopters for every system. We adopted the OWASP Enterprise Security API (ESAPI) library as a part of MDSE@R security controls database.

To support run-time security enforcement, MDSE@R uses a combined interceptor and AOP approach. Whenever a client or application component makes request to any system component method, this request is intercepted by the system container. The system container supports wrapping of both new developments and existing systems. For new development, Swinsoft system engineers should use the Unity application block delivered by Microsoft PnP team to intercept calls to registered classes. This is a .NET-based implementation of the dependency injection design pattern. It supports dynamic runtime injection of interception points on methods, attributes and class constructors. For existing systems we adopted Yiihaw AOP for C#, where we can modify application binaries (dll and exe files) to add security aspects at any arbitrary method (in our implementation we add a call to our security enforcement point).

The *Security Test Case Generator* (6) uses the NUnit testing framework to partially automate security controls and system integration testing. We developed a test case generator library that generates a set of security test cases for authentication, authorization, input validation, and cryptography for every enforcement point defined in the interceptors' document. MDSE@R uses NUnit library to fire the generated test cases and notifies security engineers via test case execution result logs. At runtime, whenever a request for a system resource is received (7), the system container checks for the requested

method in the live interceptors' document. If a matching found, the system delegates this request with the given parameters to the default interception handler - security enforcement point (8).

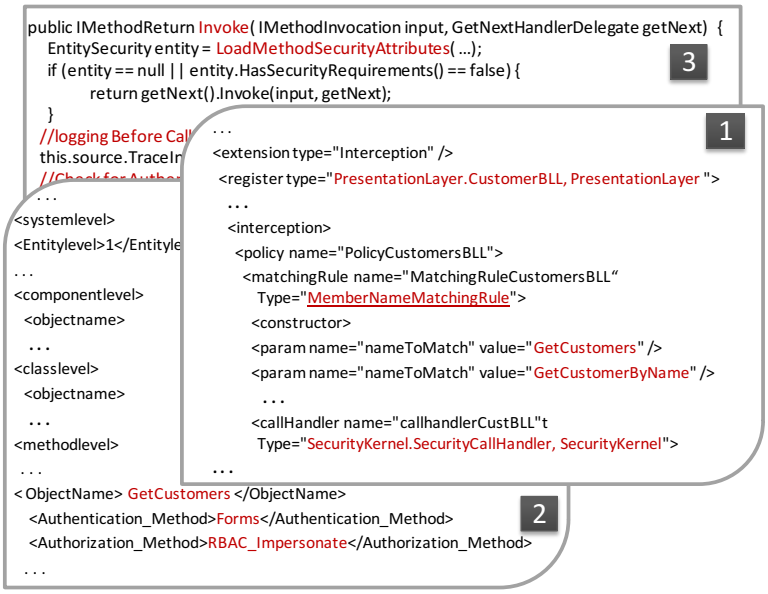

Figure 13. Examples of MDSE@R weaved system interceptors and security specification files

The Security Enforcement Point (9) is a class library that we developed to act as the default interception handler and the mediator between the system and the security controls. Whenever a request for a target application operation is received, it checks the system security specification document to enforce the particular system security controls required. It then invokes such security controls through APIs published in the security control database (4). The security enforcement point validates a request via the appropriate security control(s) configured and specified, e.g. imposes authentication, authorization, encryption or decryption of message contents. The validated request is then propagated to the method for execution (10).

Both system and security modelling tools are based on VS 2010 Modelling SDK that enables developing DSVLs integrated with VS IDE. To develop each DSVL, we developed a meta-model for the DSL domain and specified the corresponding shapes that visualize each domain model concept. Then we specified the mapping between the domain concepts' attributes and the shape compartments. Finally we developed code generation templates that generate the system live interceptors' document and the security specification document from the system and security models. Our modelling tools use a repository to maintain models developed either by the system engineers or by the security engineers. It also maintains the system live interceptors' document and security specification document. Examples of these documents are shown in Figure 13. Examples of MDSE@R weaved system interceptors and security specification files. This example shows a sample of the Galactic interceptors' document generated from the specified security-system mapping. It informs the system container to intercept GetCustomers and GetCustomerByName methods (1); a sample of Swinburne security specification file defining the security controls to be enforced on every intercepted point (2); and a sample of the security enforcement point API that injects the necessary security control calls before and after application code is run (3).

Vulnerability Analysis and Mitigation

We developed a GUI, as shown in Figure 14, to assist security experts in capturing vulnerability signatures' in OCL. This provides vulnerability signature editing, validity checking, and testing these signatures' specifications on simple target applications. We use an existing OCL parser to parse and validate signatures against our system description meta- model. Once validated, the vulnerability signature is stored in our weakness signatures database. To parse the given program source code and generate a

system abstract model, we use ***NReFactory*** .NET parser Library [27], which parses source code and generates its corresponding AST (it supports VB.Net and C#. We are currently working on parsers for PhP and Java). Applications without source code - i.e. only binaries are available – are decompiled using ***ILSPY***. This is currently supported for C# and VB.NET. We developed a class library to transform the generated AST into a more abstract (summarized) representation that conforms to our system description model. Our signature locator has an OCL translator that translates a given OCL signature into a corresponding C# class with a signature matching method that checks the passed in system entity looking for matches to specified signatures. The OCL functions library maintains a set of functions that extend the system description meta- model entities capabilities and can be used during the vulnerability analysis phase. This includes control- flow analysis (CFA), data-flow analysis (DFA), and tainted-data analysis. These functions can be extended with further analysis functions based on future vulnerability analysis needs. The OCL to C# transformer performs a transformation for these functions as well as new OCL signatures once defined. Program slicing and taint analysis techniques (core techniques in program and security analysis area) can be easily captured in OCL. Platforms' profiles are specified in XML documents that contain information about specific platforms' details. It is used to set the context of the signature locator according to the software.

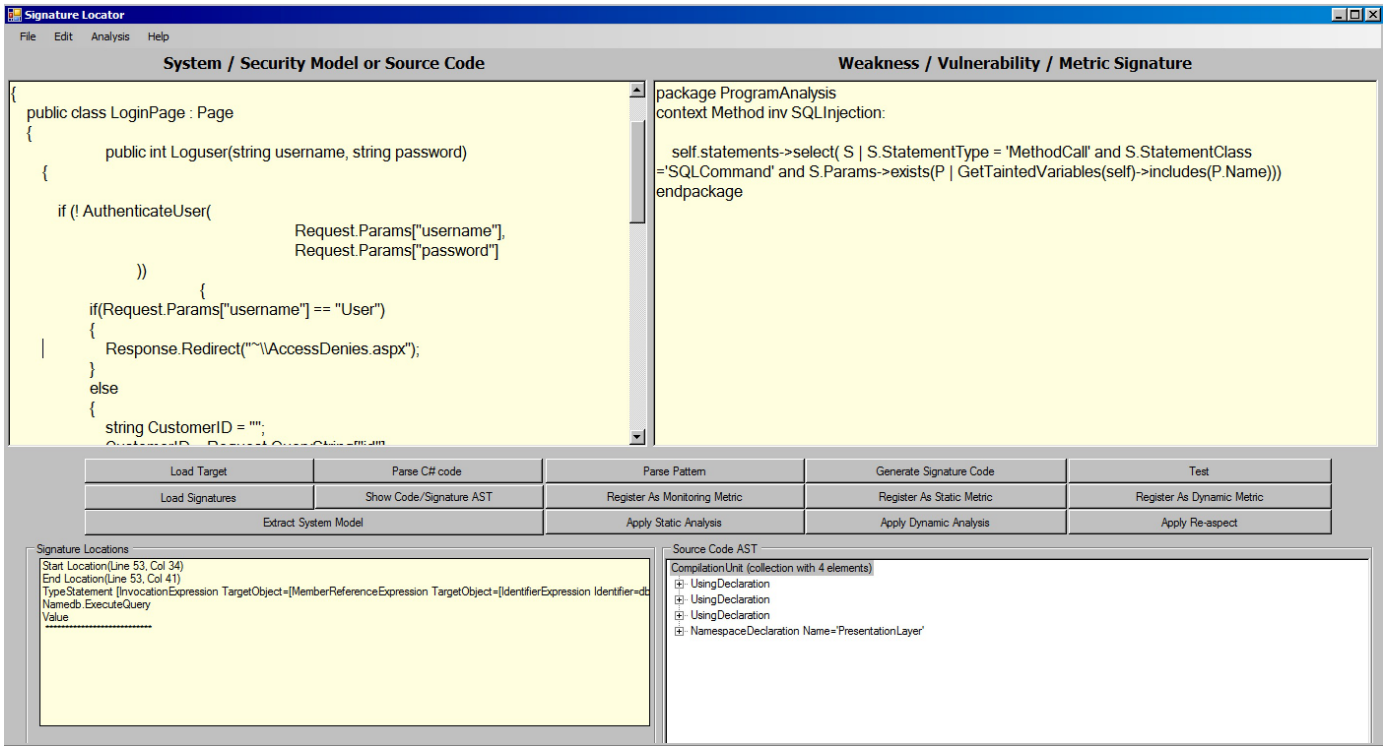

Figure 14. Snapshot of the vulnerability analysis tool

References

[1] R. Barnett, "WAF Virtual Patching Challenge: Securing WebGoat with ModSecurity," 2009.

[2] B. Fabian, S. Gürses, M. Heisel, T. Santen, and H. Schmidt, "A comparison of security requirements engineering methods," *Requirements Engineering,* vol. 15, pp. 7-40, 2010.

[3] R. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*: John Wiley and Sons, 2001.

[4] A. Dardenne, A. v. Lamsweerde, and S. Fickas, "Goal-directed requirements acquisition," presented at the Selected Papers of the Sixth International Workshop on Software Specification and Design, 1993.

[5] H. S. F. Al-Subaie and T. S. E. Maibaum, "Evaluating the Effectiveness of a Goal-Oriented Requirements Engineering Method," 2006.

[6] A. Lamsweerde, S. Brohez, and e. al, "System Goals to Intruder Anti-Goals: Attack Generation and Resolution for Security Requirements Engineering," in *Proc. of the RE'03 Workshop on Requirements for High Assurance Systems*, Monterey, 2003, pp. 49-56.

[7] L. Liu, E. Yu, and J. Mylopoulos, "Secure ¡* : Engineering Secure Software Systems through Social Analysis," *International Journal of Software and Informatics,* vol. Vol.3, pp. 89-120, 2009.

[8] L. Liu, E. Yu, and J. Mylopoulos, "Security and Privacy Requirements Analysis within a Social Setting," presented at the Requirements Engineering, 2003.

[9] H. Mouratidis, and P. Giorgini, "Secure Tropos: A security-oriented Extension of the Tropos Methodology," *International Journal of Software Engineering and knowledge Engineering,* 2007.

[10] H. Mouratidis and J. Jurjens, "From goal-driven security requirements engineering to secure design," *International Journal of Intelligent Systems,* vol. 25, pp. 813-840, 2010.

[11] R. Matulevičius, N. Mayer, H. Mouratidis, E. Dubois, P. Heymans, and N. Genon, "Adapting Secure Tropos for Security Risk Management in the Early Phases of Information Systems Development," in *Proc. of the 20th international conference on Advanced Information Systems Engineering*, 2008, pp. 541-555.

[12] G. Sindre, and A. Opdahl, "Eliciting security requirements with misuse cases," *Requir. Eng.,* vol. 10, pp. 34-44, 2005.

[13] D. G. Firesmith, "Security Use Cases," *JOURNAL OF OBJECT TECHNOLOGY,* vol. Vol. 2, No. 3, pp. pp. 53-64, 2003.

[14] J. Jürjens, "Towards Development of Secure Systems Using UMLsec," in *Fundamental Approaches to Software Engineering*. vol. 2029, ed: Springer Berlin Heidelberg, 2001, pp. 187-200.

[15] J. Jurjens, J. Schreck, and Y. Yu, "Automated Analysis of Permission-Based Security using UMLsec," in *Proc. of 11th international conference on Fundamental approaches to software engineering* 2008, pp. pp. 292 - 295.

[16] J. Jürjens, "UMLsec: Extending UML for Secure Systems Development," presented at the Proc. of the 5th International Conference on The Unified Modeling Language, 2002.

[17] L. Montrieux, J. Jurjens, C. B. Haley, Y. Yu, P.-Y. Schobbens, and H. Toussaint, "Tool support for code generation from a UMLsec property," presented at the Proc. of The 2010 IEEE/ACM international conference on Automated software engineering, Antwerp, Belgium, 2010.

[18] T. Lodderstedt, D. Basin, and J. Doser, "SecureUML: A UML-Based Modeling Language for Model-Driven Security," in *Proc. of The 5th International Conference on The Unified Modeling Language*, Dresden, Germany, 2002, pp. 426-441.

[19] F. Satoh, Y. Nakamura, N. K. Mukhi, M. Tatsubori, and K. Ono, "Methodology and Tools for End-to-End SOA Security Configurations," in *Services - Part I, 2008. IEEE Congress on*, 2008, pp. 307-314.

[20] Y. Shiroma, H. Washizaki, Y. Fukazawa, and A. Kubo, "Model-Driven Security Patterns Application Based on Dependences among Patterns," in *Proc. of The International Conference on Availability, Reliability, and Security.* , Krakow 2010, pp. 555-559.

[21] N. A. Delessy and E. B. Fernandez, "A Pattern-Driven Security Process for SOA Applications," in *Proc. of The Third International Conference on Availability, Reliability and Security*, 2008, pp. 416-421.

[22] M. Schnjakin, M. Menzel, and C. Meinel, "A pattern-driven security advisor for service-oriented architectures," presented at the Proc. of 2009 ACM workshop on Secure web services, Chicago, Illinois, USA, 2009.

[23] M. Hafner, M. Memon, and R. Breu, "SeAAS - A Reference Architecture for Security Services in SOA " *Journal of Universal Computer Science,* vol. vol. 15, pp. 2916-2936, 2009.

[24] M. Alam, "Model Driven Security Engineering for the Realization of Dynamic Security Requirements in Collaborative Systems," in *Models in Software Engineering*. vol. 4364, T. Kühne, Ed., ed: Springer Berlin / Heidelberg, 2007, pp. 278-287.

[25] M. Alam, R. Breu, and M. Hafner, "Modeling permissions in a (U/X)ML world," in *Proc. of The First International Conference on Availability, Reliability and Security*, 2006, p. 8 pp.

[26] V. Bertocci, *Programming Windows Identity Foundation*: Microsoft Press, 2010.

[27] L. Peng and Y. Zhao-lin, "Analysis and extension of authentication and authorization of Acegi security framework on spring," *Computer Engineering and Design,* 2007.

[28] L. A. Abdulkarim and Z. Lukszo, "Information security implementation difficulties in critical infrastructures: Smart metering case," in *Proc. of The International Conference on Networking, Sensing and Control*, 2010, pp. 715-720.

[29] M. Hafiz and R. E. Johnson, "Improving perimeter security with security-oriented program transformations," in *ICSE Workshop on Software Engineering for Secure Systems*, 2009, pp. 61-67.

[30] M. Hafiz and R. E. Johnson, "Security-oriented program transformations," presented at the Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies, Oak Ridge, Tennessee, 2009.

[31] V. Ganapathy, D. King, T. Jaeger, and S. Jha, "Mining Security-Sensitive Operations in Legacy Code Using Concept Analysis," presented at the Proc. of the 29th international conference on Software Engineering,

2007.
[32] P. O'Sullivan, K. Anand, A. Kothan, M. Smithson, R. Barua, and A. D. Keromytis, "Retrofitting Security in COTS Software with Binary Rewriting," presented at the Proc. of the 26th IFIP International Information Security Conference (SEC), Lucerne, Switzerland., 2011.
[33] V. Ganapathy, T. Jaeger, and S. Jha, "Retrofitting legacy code for authorization policy enforcement," in *2006 IEEE Symposium on Security and Privacy*, 2006, pp. 15 pp.-229.
[34] I. S. WELCH and R. J. STROUD, "Re-engineering Security as a Crosscutting Concern," *The Computer Journal,* vol. 46, pp. PP. 578-589, 2003.
[35] A. Elkhodary and J. Whittle, "A Survey of Approaches to Adaptive Application Security," in *Int. Workshop on Software Engineering for Adaptive and Self-Managing Systems*, 2007, pp. 1-16.
[36] B. Hashii, S. Malabarba, R. Pandey, and e. al, "Supporting reconfigurable security policies for mobile programs," in *Proc. of the 9th international World Wide Web conference on Computer networks*, Amsterdam, The Netherlands, 2000, pp. 77-93.
[37] K. Scott, N. Kumar, S. Velusamy, and e. al, "Retargetable and reconfigurable software dynamic translation," presented at the Proceedings of the international symposium on Code generation and optimization, San Francisco, California, 2003.
[38] F. Sanchez-Cid, and A. Mana, "SERENITY Pattern-Based Software Development Life-Cycle," in *19th International Workshop on Database and Expert Systems Application*, 2008, pp. 305-309.
[39] F. Sanchez-Cid and A. Mana, "Patterns for Automated Management of Security and Dependability Solutions," presented at the Proc. of the 18th International Conference on Database and Expert Systems Applications, 2007.
[40] A. Benameur, S. Fenet, A. Saidane, and S. K. Sinha, "A Pattern-Based General Security Framework: An eBusiness Case Study," in *Proc. of The 11th IEEE International Conference on High Performance Computing and Communications*, 2009, pp. 339-346.
[41] B. Morin, T. Mouelhi, and F. Fleurey, "Security-driven model-based dynamic adaptation," presented at the Proc. of the IEEE/ACM International Conference on Automated software engineering, Antwerp, Belgium, 2010.
[42] E. Yuan, N. Esfahani, and S. Malek, "A systematic survey of self-protecting software systems," *ACM Transactions on Autonomous and Adaptive Systems (TAAS),* vol. 8, p. 17, 2014.
[43] M. Almorsy, A. Ibrahim, and J. Grundy, "Adaptive Security Management in SaaS Applications," in *Security, Privacy and Trust in Cloud Systems*, S. Nepal and M. Pathan, Eds., ed: Springer Berlin Heidelberg, 2014, pp. 73-102.
[44] M. Almorsy, J. Grundy, and A. S. Ibrahim, "MDSE@R: Model-Driven Security Engineering at Runtime," presented at the Proc. of the 4th International Symposium on Cyberspace Safety and Security Melbourne, Australia, 2012.
[45] M. Almorsy and J. Grundy, "SecDSVL: A Domain-Specific Visual Language To Support Enterprise Security Modelling," presented at the 2014 Australian Conference on Software Engineering Sydney, 2014.
[46] M. Almorsy, J. Grundy, and A. S. Ibrahim, "Supporting automated vulnerability analysis using formalized vulnerability signatures," presented at the Proc. of 27th IEEE/ACM International Conference on Automated Software Engineering, Essen, Germany, 2012.
[47] M. Almorsy, J. Grundy, and A. S. Ibrahim, "Automated Software Architecture Security Risk Analysis Using Formalized Signatures," in *Proc. of The 36th International Conference of Software Engineering*, San Francisco, 2013, pp. 300-309.
[48] M. Almorsy, J. Grundy, and A. Ibrahim, "VAM-aaS: Online Cloud Services Security Vulnerability Analysis and Mitigation-as-a-Service," in *Web Information Systems Engineering - WISE 2012*, X. S. Wang, I. Cruz, A. Delis, and G. Huang, Eds., ed: Springer Berlin Heidelberg, 2012, pp. 411-425.
[49] B. D. Win, F. Piessens, and W. Joosen, "How secure is AOP and what can we do about it?," presented at the Proceedings of the 2006 international workshop on Software engineering for secure systems, Shanghai, China, 2006.