Extending the Agile Development Process to Develop Acceptably Secure Software

Lotfi ben Othmane, Pelin Angin, Harold Weffers, and Bharat Bhargava, *Fellow, IEEE*

Abstract—The agile software development approach makes developing secure software challenging. Existing approaches for extending the agile development process, which enables incremental and iterative software development, fall short of providing a method for efficiently ensuring the security of the software increments produced at the end of each iteration. This article (a) proposes a method for security reassurance of software increments and demonstrates it through a simple case study, (b) integrates security engineering activities into the agile software development process and uses the security reassurance method to ensure producing acceptably secure—by the business owner—software increments at the end of each iteration, and (c) discusses the compliance of the proposed method with the agile values and its ability to produce secure software increments.

Index Terms—Agile software development, secure software, security assurance cases

1 Introduction

Developing *secure software* that continue to function correctly under malicious (intended) attacks [1] requires integrating security engineering activities and verification and validation gates into the development process. The *security engineering activities* capture and refine protection requirements and ensure their integration into the software through purposeful security design [2], while the verification and validation gates ensure traceability [3] of analysis, design, coding, and testing artifacts; which helps addressing the weakest link (i.e., least protected point) problem [4] by ensuring completeness of the protection mechanisms.

The sequential software development approach suits the integration of the security engineering activities, which are commonly used in sequence, and the use of verification and validation gates between the development stages: analysis, design, coding, and testing. In contrast, the iterative and incremental nature [5] of the agile software development (ASD) [6] approach enables developing software in regular intervals, i.e., iterations, producing the software in increments. The iterative and incremental nature [5] of the Agile Software Development (ASD) [6] approach limits its ability to accommodate the security engineering activities and the use of verification and validation gates. The development process it employs does not fit the sequential use of the security engineering activities and the set of verification and validation gates.

There are several challenges that limit the use of ASD for developing secure software such as, lack of complete view of the system, absence of security engineering activities in the development process, lack of detailed documentation, lack of security awareness of the customers, and conflict of interests between security professionals and developers [7], [8], [9], [10]. Several solutions have been proposed for extending the ASD process to produce secure software, e.g., [11], [12], [13], but they either fall short of ensuring the security of the increments produced in each iteration, or require performing a long list of security verification and validation tasks, (e.g., OWASP verification requirements [14]) in each iteration—which implies that all security requirements must be implemented in the first development iteration.

This paper aims to address the question: Is it possible to extend the agile development process to produce acceptably secure software in each iteration? However, to answer this question we need to answer the secondary question: Is it possible to efficiently ensure the security of software increments produced by development iterations? We answer the secondary question through proposing a method for security reassurance of software increments. We address the issue through using security assurance case technique [15], which supports security reassurance of software increments thanks to its tree-like structure. We answer the main question through extending the agile development process with security engineering activities while preserving the ASD values.

The paper is organized as follows. First, we provide an overview of the agile software development approach, secure software development, and security assurance cases (Section 2). Then, we discuss related work (Section 3), describe our method for security reassurance of software increments (Section 4), propose a process for developing acceptably secure software using the ASD approach (Section 5), and illustrate the use of the proposed development process using a case study, a simple HR Web Portal (Section 6). Next, we discuss the proposed development approach (Section 7) and conclude the paper (Section 8).

- *L. ben Othmane is with Lero-The Irish Software Engineering Research Center, Ireland.*
- *P. Angin and B. Bhargava are with the Department of Computer Science, Purdue University, West Lafayette, IN 47907.*
- *H. Weffers is with the Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, North Braband, The Netherland.*

Manuscript received 4 Nov. 2013; accepted 31 Dec. 2013; published online xx xx xxxx.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-2013-11-0321.
Digital Object Identifier no. 10.1109/TDSC.2014.2298011

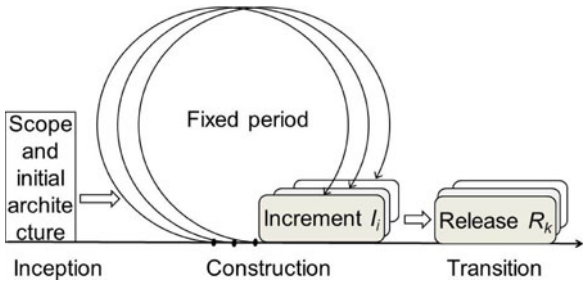

Fig. 1. The agile software development process.

2 Background

This section gives an overview of the ASD approach, secure software development and security assurance cases.

2.1 Agile Software Development Approach

The ASD approach, as specified in the manifesto [6], has four values: individuals and interactions, working software, customer collaboration, and responding to change. The approach is implemented by several methods including: Scrum [16], extreme programming (XP) [17], and agile modeling (AM) [18]. It enables producing potentially shippable working software at regular intervals [18] (i.e., iterations), which enables providing customers high value features (customer-valued product functionalities) in a short time. It applies a greedy-like approach with incomplete information for selecting functionalities to develop.[1] The approach relies on the use of patterns, principles, and best practices for developing "good" software. The approach accommodates several classes of software, such as Web applications [19].

The agile approach has several advantages. First, it reduces the chance of project failure because it enables early detection of gaps between business expectations and developers understanding. Second, it enables discovery of customer needs rather than customer wishes since customers can see demos of the product while being developed and adapt the requirements based on their needs. Third, it enables early discovery of technical barriers since the developers experiment their ideas and use the results to adapt the system architecture and work plan.

The ASD methods share a similar process, shown in Fig. 1. The process supports developing software in an iterative and incremental fashion. It has three phases (cf. [20]): inception, construction, and transition. The *inception phase* is for defining the scope of the project and model of the initial architecture. The *construction phase* is for developing the software in a set of iterations. For each iteration, the business owner and development team determine the goal of the iteration and select a set of user stories to achieve the goal. Then, they elicit the requirements for the stories, update the design to address the requirements, and code a software increment that addresses the requirements and is potentially shippable. They demonstrate the user stories and review the team efficiency at the end of the iteration.

1. The greedy approach (the term "greedy algorithm" is used in the literature) makes the choice that looks best at the moment with the hope to have an optimal solution.

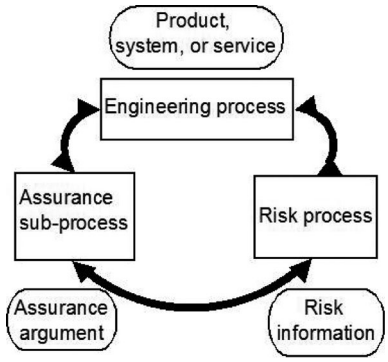

Fig. 2. CMM security engineering process [21].

The *transition phase* is for integration testing and for hardening the increment to make it ready as a release[2] for use in a production environment.

2.2 Secure Software Development

Secure software are developed using processes that integrate security activities for capturing and refining protection requirements and for ensuring their integration into the software through purposeful security design [2]. A known reference model of engineering secure software is the System Security Engineering Capability Maturity model (SSE-CMM) [21]. Fig. 2 shows the Capability Maturity model (CMM) security engineering process (SSE-CMM). The process has three sub processes: risk process, engineering process, and assurance process. The risk process enables identifying threats to and vulnerabilities of a given system along with their associated impacts and likelihood of occurrence [22]; that is, their risks. The engineering process supports determining and implementing solutions to the threats. The security assurance process ensures that the security features (high-level security requirements that express protection capabilities of the software to mitigate the threats [23]), practices, procedures, and architecture of software accurately mediate and enforce the security policy [2]. *Security policies* state the required protection of the information objects [24]; they are rules for sharing, accessing, and using information, hardware, and software.

2.3 Security Assurance Cases

Security assurance enables developing coherent objective arguments—which could be reviewed—to support claiming that a software product mitigates its security risks. A *security assurance case* [15], a semi-formal approach for security assurance, is a collection of security-related claims, arguments, and evidences where a *claim*, i.e., a security goal, is a high-level security requirement, an *argument* is a justification that a set of (objective) evidences justify that the related claim is satisfied, and an *evidence* is a result of a verification through, for example, security testing, source code security review, mathematical proofs, checking use of secure coding standards, qualification of the developers in terms of training on developing secure software (cf.[25]), etc.

2. A software product could have many releases.

Ensuring that software mitigates a security risk requires:

- ensuring that collected evidences indeed support the related claim. For example, a source code static analysis of a Web application cannot justify that the communication between a client and a Web server is secure–it only supports the claim.
- having evidences that sufficiently justify the claims.[3] For example, a verification of compliance with standards for writing secure code [27] helps avoiding source code vulnerabilities, such as buffer overflow [28] but does not justify a claim that the code is free from source code vulnerabilities.
- evaluating and addressing conflicts and dependencies between both the claims and the evidences.

A security assurance case has a tree structure, where the root is the main claim, intermediate nodes are either sub-claims or arguments, and the leaves are the evidences. A common way to represent assurance cases is to use the goals-structuring notation (GSN) [29]. *Goals-Structuring Notation (GSN)*[4] is a graphical argumentation notation that represents each element of the assurance case and the relationships between these elements.

The main steps of creating a security assurance case (cf. [31], [32]), in sequence, are:

1. *Identify the claims*—decompose the claim "the software is secure" into sub-claims such that satisfying the sub-claims induce satisfying the claim. The sub-claims (which in turn become claims) could be iteratively subdivided, until getting verifiable sub-claims.
2. *Establish the context*—specify additional information for claims, such as definitions, reference to documents, explanations, and assumptions.
3. *Identify the strategies*—provide information on how a claim is decomposed into sub-claims. The strategy could be explicitly described in the assurance case or be implicit if no strategy is specified.
4. *Identify evidences*—collect the result of using the security evaluation techniques, such as security testing and security review of source code to evaluate the security countermeasures [23] used to eliminate or reduce the risk of the threats to the software and achieve the related security goals.
5. *Specify the arguments*—show implicitly or explicitly that an evidence supports a claim. For example, the results of a security analysis tool may report that the software has a set of code security vulnerabilities (e.g., buffer overflow). The argument describes that the "errors" are false positives and the claim is satisfied.

The two main advantages of security assurance cases over checklists are (a) richness of argumentation and (b) completeness of decomposition. Security assurance cases provide richness of argumentation because they record the evidences, arguments, assumptions, and contexts that justify why the evaluator[5] believes that a claim is satisfied. For instance, it records all results generated by a code security analysis tool, lists false positives and the arguments for ignoring them. In contrast, checklists require to assert for each claim[6] if it is satisfied, but does not justify how the evidence supports the claim.

Security assurance cases have a tree-like structure, which helps ensuring completeness of decomposition of claims because the assurance method simplifies identifying (known) sub-claims of a claim and helps identifying the dependency relationship between the claims. In contrast, checklists[7] do not help to easily identify all the divisions of claims. For instance, a security verification checklist (e.g., [14]) may include the requirement "verify that cryptographic module failures are logged" but it does not help to easily identify the need for the claim "Unauthorized access to the logs is prevented"—which may cause a security flaw. In contrast, the security assurance case technique helps to identify the need easily.

3 Related Work

We describe in this section three approaches for extending the agile development process to develop secure software and a security assurance approach that is close to our work. There are several other approaches, in the literature, that integrate the security engineering activities into the agile development process, e.g., [33]; however, they do not ensure the security of the developed increments.

OWASP approach. The Open Web Application Security Project (OWASP) [11] proposes developing evil user stories (hacker abilities to compromise the system), and expressing them in a conversational style. An example for an evil story is: As a hacker, I can send bad data in URLs, so I access data for which I'm not authorized. The stories address authentication, session management, access control, input validation, output encoding/escaping, cryptography, error handling and logging, data protection, communication security, and HTTP security features.

The main issues of the approach are: (1) it requires mitigating all applicable evil stories in every iteration, which implicitly requires focusing on the security user stories in the first iterations of the software and (2) it requires verification of a list of security requirements (e.g., [14]) for the full software in each increment of the software. In contrast, the approach that we propose requires that the product owner (A project stakeholder that has a vision about the system to build and represents the users of the future system.) chooses for each iteration the security user stories to implement and requires reassessing the security of only a set of security claims. (Section 4 discusses the security reassessment process.)

Microsoft approach. Sullivan [12], [34] proposed integrating security-related activities in the development life-cycle considering their required frequency of completion. He divides the activities into three groups based on their required completion frequency. The groups are: one-time activities, cycle security-related activities, and bucket

3. Note that security is a system property [26].

4. Goals-Structuring Notations (GSNs) could be traced back to McDermid's work [30], but without the graphical notations.

5. In general, an evaluator is a specialized professional who evaluates whether the software complies with the requirements and how it does so.

6. Checklists use the term security requirement to mean a claim.

7. The elements of checklists are commonly identified based on experience.

security-related activities. The one-time activities group includes activities that are performed only once in each project, during the preparation phase. The cycle security-related activities group includes activities that are performed in each iteration of the project. The bucket security-related activities group includes one verification task, one design review task, and one response planning task. Verification tasks include, for example, attack surface analysis and fuzz testing; design review tasks include, for example, review of code that uses cryptographic operations; and response planning tasks include, for example, update of security response contacts.

The full set of bucket security-related activities are performed in a cycle of a sequence of a set of iterations—e.g., a cycle of six iterations. This reduces the set of increments that are ready for releasing to the increments produced by the last iteration of each cycle. In contrast, the approach that we propose does not use bucket security-related activities.

Risk-driven secure software development. Vähä-Sipilä [13], [35] proposes a risk-based approach for developing secure software. The method is based on managing the security risks and implementing security solutions.

Managing security risks is reducing the risks of security threats, through implementing security controls, to a level acceptable by the business owner. The security threats include the threats related to the functionalities and the architecture of the software (e.g., as a user, I do not want my data to be used by anybody, except for processing my transaction), threats related to specific user stories, and threats related to code vulnerability and data validation. The risks of threats are reduced using security mechanisms, which are associated with implementation and operation costs. The approach conditions the completeness of user stories to mitigating the related threats.

The main issue with the approach is that it focuses on the threats to user stories individually and does not consider the impact of code changes resulting from implementing new user stories on the validity of the preexisting security claims. In contrast, our approach maintains a global state of the security assurance of the software in the form of a security assurance case of the software. The developers reassess the security assurance of the software at the end of each iteration.

A similar approach is proposed by Ge et al. to develop secure Web applications using the Feature-Driven Development (FDD) method [36]. The authors integrate efficiently the risk assessment activities into the development process. However, their process does not integrate security assessment activities, which limits the ability to claim that the produced increments are secure.

Security assurance-driven software development. Vivas et al. [15] proposed a method for security assurance-driven software development that focuses on evolving the security assurance cases of software throughout the development life-cycle phases: analysis, design, development, and test. They proposed an approach for decomposing the claims of assurance cases and refining them as the development team members identify the use cases of the software, design the components of the software and implement them as code classes.

The approach that we propose relates security assurance cases elements to the software development artifacts as in [15]. However, the approach enables reassurance of the software as it evolves in increments—that is, produced iteratively in successive iterations.

4 Security Reassurance of Software Increments

Evolving software, through adding user stories, requires reassessing the security assurance of the software. The reason is: the changes to the software components could invalidate the evidences and claims of the security assurance case. For example, evidences collected using a code security analysis tool become invalid if new code is added to the software and the claim "access control to files is enforced" becomes invalid if the files become available in several locations.

The safe approach for assessing the security assurance of a new increment of software is to discard the security assurance case of the previous increment and perform a new security assessment for the new increment. However, performing a new security assessment for each increment is not practical because it is time consuming and has high cost. For instance, it is not possible to perform the 121 security verification requirements required by OWASP [14] for a Web application that has a new increment every week.

This section describes our method for developing security reassurance cases of software increments. First, we analyze the security verification requirements of OWASP [14]. Then, we analyze the relationship between security assurance case elements and software components, which helps to implement the ideas identified in Section 4.1 and devise a method of security reassurance of software increments that we describe in Section 4.3.

4.1 Analysis of Security Verification Requirements of OWASP

The Open Web Application Security Project developed a standard for evaluating the security of Web applications [14]. We classified the 121 security verification requirements of the standard (which are grouped in 14 security verification areas) based on classes of security assessment techniques, locality (the assessment concerns specific components or all components of the software), and automation (the assessment is automatic or manual). Table 1 shows the statistics that we obtained. Based on the analysis, we observe the following:

- Assessment automation: We use assessment techniques to collect evidences. These techniques are either manual (i.e., performed by humans) or automatic (i.e., using tools and scripts). We observe that most of the security verification requirements, 79 percent (96 out of 121), require manual assessment.
- Evidence locality: Some evidences apply to only parts of the software (e.g., components or classes) and others apply to the whole software. For instance, verifying an authentication mechanism requires verifying only the web pages and business logics

TABLE 1
Analysis of the Verification Areas of OWASP [14]

Security verification areas	Assessment technique class	Locality (Local/ Global)	Automation (Automatic/ Manual)
Security architecture	RA	4/2	1/5
Authentication	RA/TFU	12/3	3/12
Session management	RA/TFU	5/8	4/9
Access control	RA/TFU	13/2	6/9
Input validation	RA/TFU	8/1	4/5
Output encoding/ escaping	RA	9/1	1/9
Cryptography	RA	9/1	0/10
Error handling and logging	RA/TFU	9/3	2/10
Data protection	RA	6/0	0/6
Communication security	RA/TFU	7/2	1/8
HTTP security	RA	6/1	3/4
Security configuration	RA	0/4	0/4
Malicious code search	RA	2/0	0/2
Internal security	RA	0/3	0/3
		90/31	25/96

Notes:
1) We use the following abbreviation: RA for review and analysis and TFU for testing of security features and their use.
2) The numbers correspond to the count of the security verification requirements of the class.

that implement the security feature; there is no need to assess all the software. The analysis shows that most of the verification requirements, 74 percent (90 out of 121), are local.

The results of this analysis suggests that automation of security verification requirements may not have big impact on simplifying the security assurance activities as suggested by [7].[8] However, it does suggest that we could reduce the risk assessment activities by considering the locality of the software changes.

4.2 Relationship between Security Assurance Case Elements and Software Components

Fig. 3 shows the conceptual model of security assurance cases for agile software development. In this model, the project owner specifies the user stories and the security policies. A *user story* describes a functionality valuable to the user of the software [37]. A *security policy* states the required protection of the information objects [24].

A security claim, i.e., a goal, specifies a capability of the system to protect, prevent, detect, or deter a set of threats. For example, the security claim "prevent unauthorized users from accessing and using the application" specifies that the software prevents the threat "unauthorized access and use of the application." A claim could be decomposed into sub-claims using a decomposition strategy and described using a context annotation.

Threats are mitigated by security countermeasures. A *countermeasure* is an action, device, procedure, or technique that counters a threat by eliminating or preventing it, by minimizing the harm it can cause, or by discovering and reporting it so that corrective action can be taken [23]. Countermeasures are related to user stories, e.g., a countermeasure is implemented by a set of user stories. They are also related to arguments, e.g., secure coding could contribute to the argument of the claim "minimum source code vulnerabilities."

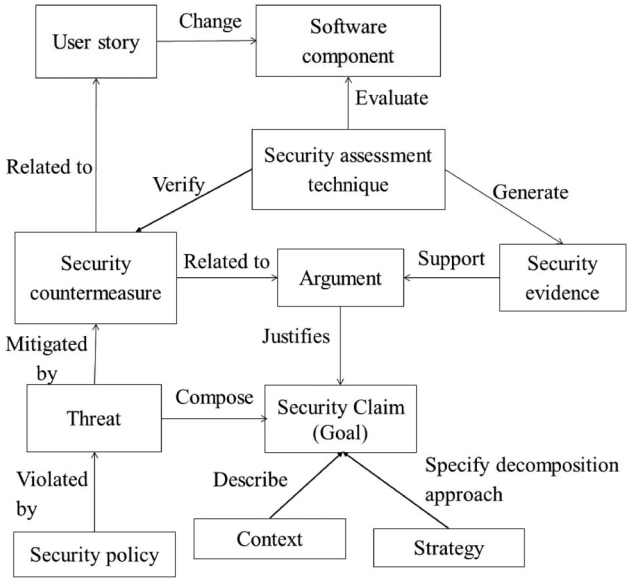

Fig. 3. Model of the relationship between security assurance case concepts and software development artifacts.

Implementing a user story may require adding new components,[9] removing components, and/or updating existing components—i.e., changing their structure or behavior. These operations could invalidate the elements of the security assurance case of the software, e.g., a claim becomes false. The invalidation of claims and evidences depends on the application because changing a component may or may not invalidate related evidences. *Invalidate* means "does not support" for evidences and "is not true" for claims. The invalidation types are:

I1. Changes to a context could invalidate related claims. We formulate this invalidation using the function: $I_{cl}(C_x) = \{C_l^i\}$.
I2. Changes to a component could invalidate related evidences. We formulate this invalidation using the function: $I_{ev}(C_o) = \{E_v^i\}$.
I3. Invalidation of evidences invalidates related claims. We formulate this invalidation using the function: $I_{cl}(E_v) = \{C_l^i\}$.
I4. Claims could have relationships, such as dependency and conflict. Changes to a claim could invalidate a related claim. We formulate this relationship using the function: $T_{cl}(C_l) = \{C_l^a\}$.

We use symbols C_x for context, C_o for component, E_v for evidence, C_l for claim, C_l^i for invalidated claim, E_v^i for invalidated evidence, and C_l^a for potentially affected claim.

8. The reason is: the percentage of security verification requirements that could be automated is low.

9. A *software component* is a unit of composition (and is also subject to composition) with specified interfaces and explicit context dependencies and can be deployed independently [38].

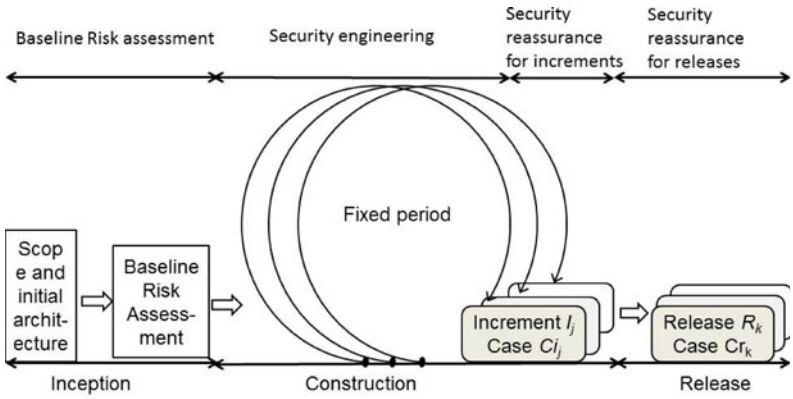

Fig. 4. Process for agile development of secure software.

We also use $I_{cl}(C_x)$ to denote a function that provides the claims associated with the context provided as a parameter, $I_{ev}(C_o)$ to denote a function that provides the evidences associated with the component provided as a parameter, $I_{cl}(E_v)$ to denote a function that provides the claims associated with the evidence provided as a parameter, and $T_{cl}(C_l)$ to denote a function that provides the claims related to the claim provided as a parameter.

Note also that invalidation of claims makes evidences associated to the claims useless. Also, invalidation of claims and evidences makes associated arguments useless. (We do not discuss invalidation of arguments because the process is straightforward.)

4.3 Methodology of Security Reassurance of Software Increments

The method of security reassurance of software increments needs to maintain the security assurance cases. For instance, each assurance case shares a set of artifacts with the security assurance case of the previous iteration. The security assurance case of a new increment requires only new evidences for invalidated claims or new claims.

An efficient method for security reassurance of software increments minimizes the reassurance task, which requires minimizing the number and size of claims to evaluate. We exploit the decomposition of software into a set of components and we maintain the security assurance case based on changes to the components—We exploit the locality property identified in Section 4.1.

A software increment could affect the security assurance case in several ways:

- *Case 1. Context change.* The set of claims that need evaluation due to context change of a claim includes the claim and related claims.
- *Case 2. Component update.* Component update could invalidate evidences and claims associated with the component. It could potentially affect claims associated with components related to the modified component.
- *Case 3. New component.* The set of claims that need evaluation due to adding a new component includes claims related to the new component and claims that could be affected by updating other components connected to the new component—e.g., they have a dependency with the new component.
- *Case 4. New claim.* The set of claims that need evaluation due to adding a new claim includes the new claim, sub-claims, and parent claims—i.e., related claims.

We note that the reassurance method is recursive: implementing a new claim may require implementing new components or updating existing components which affects the security assurance case as specified in Case 2 and Case 3. Also, in the worst case, an iteration invalidates all claims of the security assurance case, which requires reevaluation of these claims and in the best case it preserves all the claims of the previous iteration and evaluates the claims associated with the new components integrated to the increment.

5 Extending the Agile Software Development Process with Security Engineering Activities

The proposed approach for extending the agile development process is based on the following: (1) redesign the use of the security engineering activities (risk assessment, security engineering, and security assurance) to accommodate the iterative nature of the agile software development process (see Section 2.1); and (2) use of our method of security reassurance to reassure the security of software increments.

We extend the agile development process by integrating the security sub-processes: risk assessment, security engineering, and security assurance (see Section 2.2) to the agile development process, and ensuring production of acceptably secure software at each development iteration. Recall that *acceptably secure* software is a software increment that demonstrates a set of security goals selected by the product owner for the iteration [39].

Fig. 4 depicts the new process, called *agile secure software development process*. The description of the phases of the process follows.

5.1 Extending the Inception Phase

We extend the inception phase with three activities: threat modeling, risk estimation, and security goals identification, which are performed after scoping the project, sketching the architecture, and identifying the main user stories. The focus in this phase is on assessing the security risks to the software and not on finding solutions that we may never use [40].

The extension that we propose does not require the use of a specific threat modeling or risk assessment method. However, a possible approach to do so follows. A team composed of developers, the product owner, and a security expert could, for example, meet in a workshop to identify the threats relevant to the software [40] and the possible violations of key security protection mechanisms, e.g., authorization and data validation. (They could also use the misuse case method [41] for threat modeling.) Then, the developers and the security expert provide their perceptions of the likelihood of occurrence of the threats (the "chance" that an attacker exploits a weakness or vulnerability and attacks the software) and the product owner provides his/her perception of the severity of the impact of the threats—level of the damage of a threat when successfully triggered. The likelihood and severity estimates are combined for each threat to obtain its risk level [42].

The team members compose a set of security goals that address the threats. The security goals take the form of claims for the assurance cases and are added as security user stories to the project backlog.

5.2 Extending the Construction Phase

We extend the construction phase by adding a set of activities to enable producing "acceptably secure" software increments. At the beginning of each iteration, the product owner defines "acceptably secure" software for the iteration; that is, the security claims that the increment should satisfy. (The product owner takes responsibility for choosing the threats to mitigate and for accepting the impacts of the remaining threats in each iteration or release.)

There are two types of user stories that the developers and product owner could select from: functional user stories and security user stories. The team members should consider both the results of the risk assessment and the urgency of the customer needs in selecting the security user stories. The description of how to develop functional and security user stories follows.

Developing functional user stories. In addition to the regular software activities required in the development of a user story (see Section 2.1), the development team members perform a risk assessment for the selected user stories and develop security user stories to mitigate the threats they identify. The team members need also to apply secure coding and data validation techniques [43] to avoid source code vulnerabilities. For example, they need to carefully use memory allocation and exception handling.

Developing security user stories. The steps are:

1. Elicit the security requirements—use a threat-based security requirements eliciting process, such as Sindre and Opdahl method [44], to derive the security requirements related to the chosen security goal.
2. Develop security test scenarios—develop attack scenarios to test if the software satisfies its security requirements. For example, they design experiments for bypassing access data policy through performing a SQL injection attack.
3. Design a security feature for the user story—design a security feature that implements the security requirements and that could be integrated to the software architecture. The design transforms the security requirements expressed as constraints to a software behavior.
4. Develop the security feature—split the security feature into a set of user stories and implement them.
5. Test the security feature—implement and execute the security test scenarios to evaluate the compliance of the increment with the security requirements. The team concludes that the software implements the security user story if the results of the tests are positive.

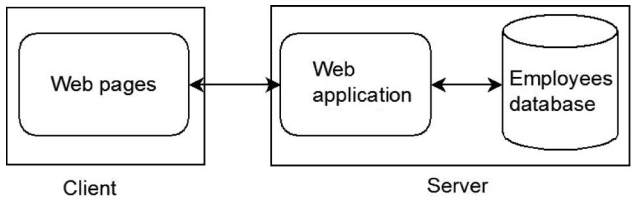

Fig. 5. Architecture of the HR Web portal application.

Security assurance. The developing team members define the claim "acceptably secure" for the iteration and use the method that we described in Section 4.3 to ensure the security of new increment.

We note that we record the evidences collected from the test scenarios of a security feature in the assurance case only when the security feature is fully developed because the assurance case notation does not provide a simple way to record partial satisfaction of a claim without ambiguity—in our opinion.

5.3 Extending the Transition Phase

In this phase, the team members perform the security assurance tasks that require extensive time and cost. They could also perform automated security tests and analysis of all the software with the aim to identify inconsistencies and increase the confidence in the security of the software. The team may seek the help of external reviewers at this phase.

6 Case Study: HR Web Portal

This section illustrates the use of the agile secure software development process including the security reassurance method that we propose using a simple HR Web portal. The portal enables viewing and updating employees' personal information and holidays.

6.1 Inception Phase

At the project inception, the product owner and the developers sketch the architecture of the intended software as depicted by Fig. 5 and identify the initial user stories as provided in Table 2. Then, they choose to develop the application using Java and run it on an Apache Web server [45].

The product owner sets two security policies: (1) each employee can view and update his/her own personnel information; and (2) HR officers can view and update all employee records. Next, the team members analyze the threats and assess the risks to the software. Then, they select four main security goals, which we describe in Table 3.[10]

10. The use of dependencies column is helpful in the security assurance task.

TABLE 2
List of User Stories

Id	User story
U1	As an HR officer, I want to create/view/update employee records.
U2	As an employee, I want to view and update my personal information.
U3	As an HR officer, I want to view and update employees' holidays.

TABLE 3
Relationships between the Security Claims

Id	Security claim	Dependencies
C01	Prevent anonymous access to the software	Supports C02
C02	Prevent unauthorized access and modification of personal information	Depends on C01
C03	Inputs are validated	
C04	Reduced source code vulnerabilities	

TABLE 4
Relationships between Presentation and Business Layer Components

Presentation Layer	Action	Business layer class	Method
Login	Input	Authentication	Authenticate
CreateEmployee	Input	Employee	setEmployee
ViewEmployee	View	Employee, Holidays	getEmployee, getHolidays
UpdateEmploye	View, Input	Employee	getEmployee, setEmployee
ManageHoliday	View, Input	Employee, Holidays	getEmployee, getHolidays, SetHoliday
ManageResources	Input, Search, View	Resource	setResource, findResource, getResource
AssignAccessControl	Input, View	AccessControl	setAccessControl, getAccessControl

(We do not describe the threat modeling and risk assessment process that helped to choose the security goals because this work does not contribute to these subjects and the choice of the methods does not have impact on the process that we propose.)

6.2 Construction Phase

The team planned three development iterations to implement the user stories, which end with a release. To ease the understanding of the evolution of the software we show the relationship between the components of the presentation and business layers of the application in Table 4 and we summarize the impacts of the user stories on the components of the software in Table 5. Table 5 includes the functional user stories of Table 2 and two security user stories: (U4) for authenticating users and (U5) for controlling use of the application.[11]

Now, we describe the development iterations and show the evolution of the security assurance case of the application.

11. These tables are not part of the proposed method.

TABLE 5
Impact of User Stories on the Components of the Architecture

User story Id	Affected components
U1	CreateEmployee (A), ViewEmployee (A), UpdateEmployee (A), Employee (A)
U2	
U3	CreateEmployee (U), ViewEmployee (U), UpdateEmployee (U), Login (U), ManageHoliday (A), Holidays (A)
U4	Login (A), Authenticate (A)
U5	Login(U), Resource (A), ManageResources (A), AccessControl (A), AssignAccessControl (A)

We use the code "A" for adding a new component and "U" for updating an existing component.

TABLE 6
List of Evidences

Id	Description
E01.1	Results of deployment architecture review
E01.2	Successful tests with the authentication mechanism
E03.1.1	Successful tests for invalid input to CreateEmployee
E03.1.2	Code review results for CreateEmployee
E03.2.1	Successful tests for invalid input to ViewEmployee
E03.2.2	Code review results for ViewEmployee
E03.3.1	Successful tests for invalid input to UpdateEmployee
E03.3.2	Code review results for UpdateEmployee
E03.4.1	Successful tests for invalid input to Login
E03.4.2	Code review results for Login
E04.1.1	Code security analysis results for Employee
E04.2.1	Code security analysis results for CreateEmployee
E04.3.1	Code security analysis results for ViewEmployee
E04.4.1	Code security analysis results for UpdateEmployee
E04.5.1	Code security analysis results for Login
E04.6.1	Code security analysis results for Authentication

Iteration 1. This iteration involves the development of the user story $U1$. The components of the increment are the `CreateEmployee`, `ViewEmployee`, and `UpdateEmployee` Web forms and the `Employee` class.

The team members define the security claim "the system is acceptably secure" for the iteration as satisfying the claims: $C01$, $C03$ and $C04$. They decide to host the application on a server that only HR officers could authenticate to using their network login.

Fig. 6 shows the security assurance case of this iteration. Claim $C01$ is satisfied by the evidence collected from a review of the software deployment architecture. Claim $C03$ is satisfied because all input for the `CreateEmployee`, `ViewEmployee`, and `UpdateEmployee` Web forms are validated, hence the sub-claims: $C03.1$, $C03.2$, and $C03.3$ are satisfied. This argument is supported by two forms of evidences: results of code review for the Web forms: `CreateEmployee`, `ViewEmployee` and `UpdateEmployee`; and security testing of the three Web forms with invalid

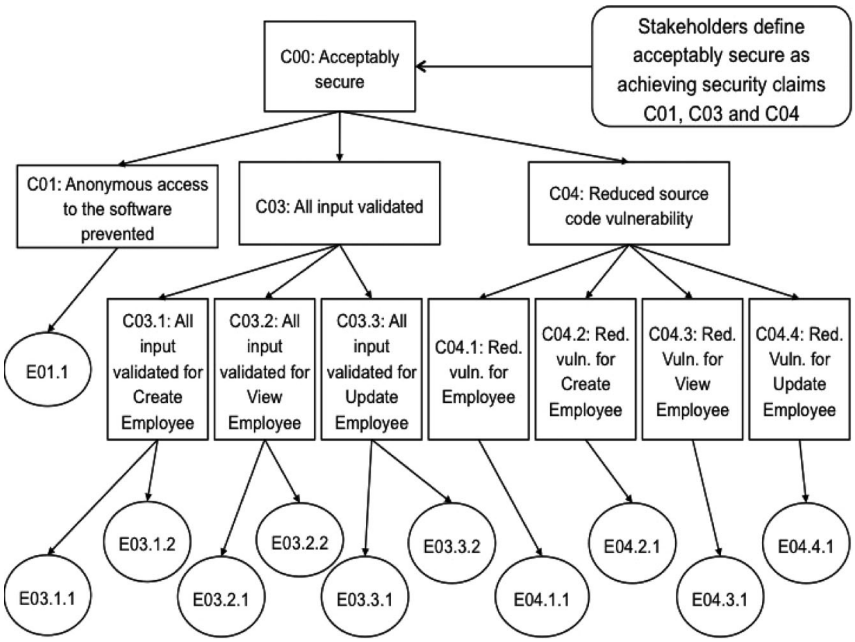

Fig. 6. Security assurance case for Iteration 1. (Table 6 describes the evidence codes.)

input handled successfully by the application. Claim $C04$ is satisfied because its sub-claims $C04.1$, $C04.2$, $C04.3$ and $C04.4$, which state that source code vulnerabilities were reduced as much as possible for the `Employee`, `CreateEmployee`, `ViewEmployee`, and `UpdateEmployee` components respectively, are satisfied. The claim is supported by the results from the code security analysis tool run on the software components. (Fig. 6 shows the evidences collected by the assessment techniques and relates them to the claims they support.)

Iteration 2. This iteration involves implementing user story $U2$. The resulting increment could be used by HR officers to create, view, and update employee records and by employees to view and update their records.

The team members define again the claim "the system is acceptably secure" of the iteration as satisfying the claims $C01$, $C03$, and $C04$. However, implementing user story $U2$ requires hosting the application on a server accessible by HR officers and employees.

The change of the deployment architecture invalidates the argument of evidence $E01.1$. An alternative approach to satisfy claim $C01$ (anonymous access to the software is prevented) is to develop an authentication mechanism, which we formulate as security user story $U4$. This iteration results in adding the `Login` and `Authentication` components to the increment and updating the configuration files of the application to require the user to be authenticated.

The iteration affects the security assurance case. The assurance case is modified as follows:

- Invalidate the evidence $E01.1$ and the claims $C01$ and $C00$.[12] Claim $C01$ is now supported by the evidence from successful tests with the authentication mechanism implemented in addition to a review of the deployment architecture.
- Add a sub-claim, $C03.4$ and associated evidences $E03.4.1$ $E03.4.1$, to claim $C03$. The changes state that we ensured the Login Web form validates its data input.
- Add a sub-claim, $C04.5$ and associated evidences $E04.6.1$ $E04.6.1$, to claim $C04$. The changes state that we ensured the newly added login and authentication components have reduced source code vulnerability.

Note that the security claims for this iteration enforce user authentication to gain access to the system, but do not ensure proper access control for the system resources, i.e., do not distinguish between HR officers and regular employees for authorizing actions.

Iteration 3. This iteration involves implementing user story $U3$. The resulting increment could be used by HR officers to create/view/update employee records and manage holidays, and by employees to view/update their records.

In this iteration, the top level security claim is composed of the three sub-claims: $C02$, $C03$, and $C04$. (Claim $C01$ becomes a sub-claim of claim $C02$.) Claim $C02$ (Prevent unauthorized access and modification of personal information) is satisfied by implementing an access control mechanism, which we formulate as security user story $U5$.

This iteration results in adding the `ManageHoliday`, `Holidays`, `ManageResources`, `Resource`, `AssignAccessControl`, and `AccessControl` components to the increment and updating the `ViewEmployee`, `UpdateEmployee` and `Login` Web forms to enforce the access control policies.

Fig. 7 shows the invalidated evidences as a result of this increment in blue dashed circles, and the invalidated claims in green dashed rectangles. The changes required the following modifications to the security assurance case:

12. Recall that claims become invalid as a result of invalidating supporting evidences or sub-claims.

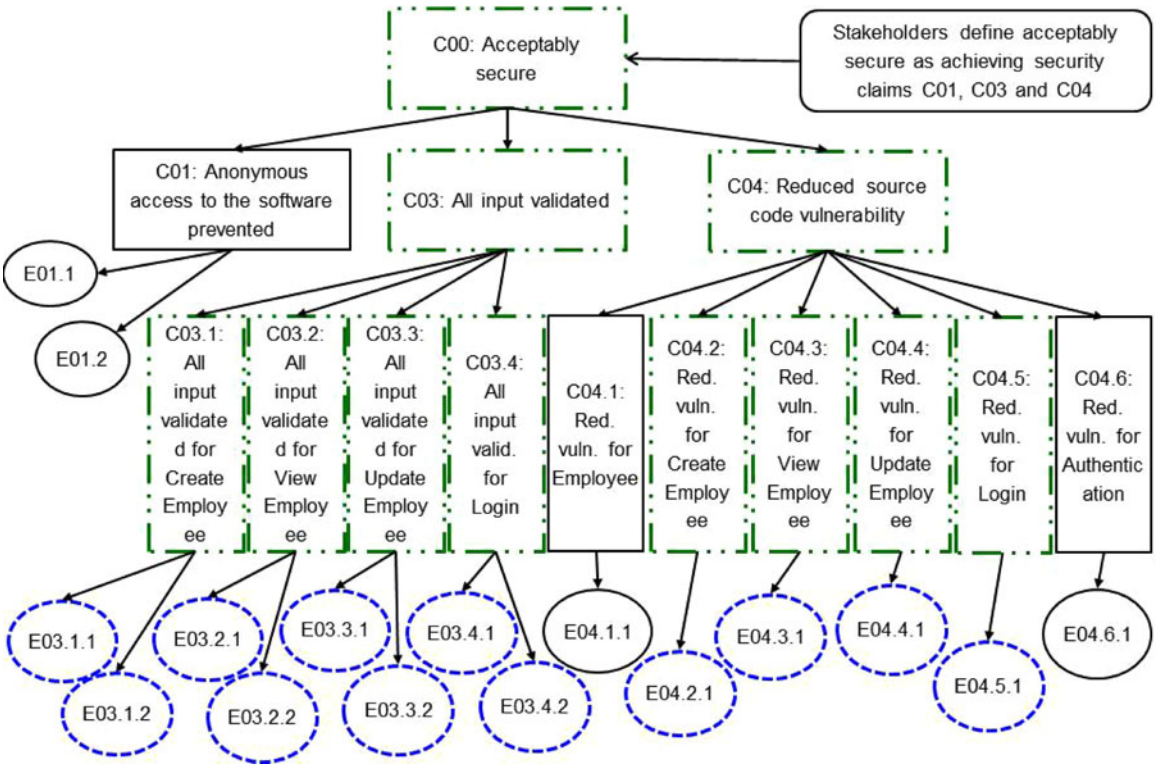

Fig. 7. Invalidated claims and evidences upon the transition from Iteration 2 to Iteration 3. (Table 6 describes the evidence codes.)

- Collect new evidences for the claims $C03.1$, $C03.2$, $C03.3$, and $C03.4$ since the evidences collected in the previous iteration become invalid.
- Collect new evidences for the claims $C04.3$, $C04.4$, and $C04.5$ since the evidences collected in the previous iteration become invalid.
- Add sub-claims and associated evidences to $C03$ to state that the forms `ManageHoliday`, `ManageResources` and `AssignAccessControl` validate the input data they receive.
- Add sub-claims and associated evidences to $C04$ to state that the components `ManageHoliday`, `Holidays`, `ManageResources`, `Resource`, `AssignAccessControl`, and `AccessControl` have reduced source code vulnerabilities.
- Relate the claim $C02$ to the (turned) sub-claim $C01$ and collect evidences from successful test scenarios for the access control mechanism.

6.3 Transition Phase

At this phase, the development team members document the main decisions they make and the security tests of the user stories they perform. They also request the help of the security expert who review the documentation and perform more tests to verify the security user stories $U4$ and $U5$.

7 Discussion

This section discusses how the proposed method preserves the agile development values and produces secure software. It also lists the limitations of the method.

7.1 Preserving the Agile Values

This section discusses how the proposed method preserves the agile values [6].

Individuals and interactions. The proposed method favors individuals and interactions over processes and tools because several of the security engineering activities are performed in collaboration between the developers and the product owner. For instance, the threat modeling and risk estimation activities are performed in workshops that include the developers and the product owner.

Working software. The proposed method favors implementing security mechanisms and uses a security assurance approach (i.e., security assurance cases) that does not require extensive documentation, but rather connects the claims to evidences that justify them. We consider the evidences as light documents that record the results of the security assessment activities.

Customer collaboration. The proposed method considers the customer perspective of secure software instead of the attacker perspective. For instance, the product owner, who represents the customers, collaborates with the developers in identifying the security threats and estimating the risks to the software. He/She is also responsible for selecting the priority of achieving the security claims.

Responding to change. The proposed method accommodates changes through (a) identifying emerging threats related to user stories selected for implementation in a new increment and (b) reassuring the security of the new increment.

7.2 Producing Secure Software

The proposed method produces secure software from a risk management perspective. For instance, it integrates

the sub-processes: risk assessment, security engineering, and security assurance. The sub-processes ensure identification of the threats to the software, engineering of security mechanisms for the main threats as perceived by the product owner, and ensuring the implemented security mechanisms mitigate the threats. The use of security assurance cases builds confidence into the security of the software; it provides the arguments justifying the evidences supporting the claims.

7.3 Limitations of the Method

The proposed method has three main limitations. Their descriptions follow.

Limited scalability. The maintenance of the security assurance cases relies on tracing the impacts of developing new user stories on the components of the software, the evidences, arguments, and claims. The difficulty of tracing the impacts grows very fast as the number of components of the software, the number of claims, evidences, and arguments grow, which limits the use of the method.

We propose to address the issue through grouping the user stories and grouping the components of the software. This requires associating the claims, evidences, and arguments to groups of user stories and groups of components. The loss of granularity in specifying the relationships between the assurance artifacts and the software artifacts limits the efficiency of the method because it increases the number of claims, arguments, and evidences that could be affected by a change to a component.[13]

Extra cost. The periodic security reassurance of a software and the need to reassess a set of claims increase the cost of security assurance. The cost of the reassessment depends strongly on the software changes for the specific iteration. For instance, Fig. 7 shows that for iteration 3 most of the claims (10 out of 14 claims) require a new assessment. The cost is still lower than reevaluating all the claims at the end of each iteration as it would be required if we used checklists of security verification requirements.

The rework cost can be justified for the case of software that evolve based on the needs of the customers–mainly produced by commercial companies. However, it may not be justified if the security goals are all known in advance and the software can be only released for use if all the goals are achieved.

Requirement of high independence between the software components. The method is based on the assumption of *high* independence between the components of the software. The assumption limits the security reassurance of a new increment to the reevaluation of a set of claims associated with the components that have changed and the few claims that are associated with all the components of the software.

The assumption is valid for software that have components-based [46] architecture or Service Oriented Architecture (SOA) [47] because they reduce the number of connections between the components and are highly testable.[14]

This assumption is strong for the general case. For instance, Ren et al. [48] analyzed the impact of code changes on the tests of the functionalities of a software, Daikon, which evolved over the period of a year. They found that, on average, 52 percent of the tests are affected by the changes made in a week. The statistics show the limitation of the efficiency of our approach for software that have high dependency between the components. This requires more research to analyze the impacts of software changes on the different kinds of security claims and the efforts required to reevaluate invalidated claims.

8 Conclusion

This paper concludes that the agile software development approach does not prevent ensuring the security of software increments produced at the end of each iteration. It proposes a method for security assurance of software increments and integrates security engineering activities into the agile software development process. The method enables ensuring the delivery of secure software at the end of each iteration.

The main advantages of the approach are: (1) helping reduce the cost of reassurance of software security, and (2) helping reduce the cost of mitigating threats. The reason for the first advantage is: the development team could reuse parts of the security assurance cases in the assessment of the new increments. The reason for the second advantage is: the approach ensures that security goals achieved in the early iterations of the development are preserved in the subsequent software increments.

The main limitations of the current method are: (1) it is not scalable enough; (2) it requires extra cost; and (3) it applies to modular software, where security claims are associated with specific components.

This paper demonstrates the possibility to develop a secure software using the agile development process through a simple case study that *simulates* the methods we propose. Our future work include automating the proposed development process and security reassurance method, evaluating the cost of security reassurance on the development time, and investigating how to consider security design principles—such as the "fail safe" principal in the assurance.

Acknowledgments

The work was performed while L. ben Othmane was with the Eindhoven University of Technology, the Netherlands.

References

[1] G. McGraw, *Software Security: Building Security In*. Software Security Series, Addison-Wesley, 2006.

[2] CNSS Glossary Working Group, "National Information Assurance (IA) Glossary," CNSS Instruction No. 4009, http://jitc.fhu.disa.mil/pki/documents/committee_on_national_security_systems_instructions_4009_june_2006.pdf, June 2006.

[3] O. Gotel, J. Cleland-Huang, J.H. Hayes, A. Zisman, A. Egyed, P. Grnbacher, A. Dekhtyar, G. Antoniol, J. Maletic, and P. Mder, "Traceability Fundamentals," *Software and Systems Traceability*, pp. 3-22, Springer-Verlag, 2012.

13. A change to a component, member of a group, requires assessing the claims and evidences associated to all the components of the group instead of only the claims and evidences associated to the component itself.

14. Easier to write tests that ensure the required properties.

[4] J. Grossklags and B. Johnson, "Uncertainty in the Weakest-Link Security Game," *Proc. First Int'l Conf. Game Theory for Networks (GameNets '09)*, pp. 673-682, May 2009.

[5] C. Larman and V.R. Basili, "Iterative and Incremental Development: A Brief History," *Computer*, vol. 36, no. 6, pp. 47-56, June 2003.

[6] Agile Alliance, Agile Alliance. Agile Alliance, http://www.agilealliance.org/, Sept. 2012.

[7] K. Beznosov and P. Kruchten, "Towards Agile Security Assurance," *Proc. Workshop New Security Paradigms (NSPW '04)*, pp. 47-54, Sept. 2004.

[8] J. Wayrynen, M. Boden, and G. Bostrom, "Security Engineering and eXtreme Programming: An Impossible Marriage?" *Proc. Fourth Conf. Extreme Programming and Agile Methods*, pp. 117-128, Aug. 2004.

[9] K.M. Goertzel and T. Winograd, "Enhancing the Development Life Cycle to Produce Secure Software,"Technical Report DAN 358844 Defense Technical Information Center (DTIC), https://buildsecurityin.us-cert.gov/bsi/resources/1185-BSI/1191-BSI.html,, 2008.

[10] K.M. Goertzel, T. Winograd, H.L. McKinley, P. Holley, and B.A. Hamilton, "Security in the Software Lifecycle," Draft Version 1.2, www.cert.org/books/secureswe/SecuritySL.pdf, Aug. 2006.

[11] OWASP, "Agile Software Development: Don't Forget Evil User Stories," https://www.owasp.org/index.php/Agile_Software_-Development:_Don 27t_Forget_EVIL_User_Stories, Aug. 2011.

[12] B. Sullivan, "Agile Security; or, How to Defend Applications with Five-Day-Long Release Cycles," Black Hat, DC, http://www.blackhat.com/presentations/bh-dc-10/Sullivan_Bryan/Black-Hat-DC-2010-Sullivan-SDL-Agile-slides.pdf, Jan. 2010.

[13] A. Vähä-Sipilä, "Product Security Risk Management in Agile Product Management," OWASP AppSec Research, https://www.owasp.org/images/c/c6/OWASP_AppSec_Research_2010_Agile_Prod_Sec_Mgmt_by_Vaha-Sipila.pdf, May 2013.

[14] M. Boberski, J. Williams, and D. Wichers, *OWASP Application Security Verification Standard 2009. The Open Web Application Security Project (OWASP)*, https://www.owasp.org/images/4/4e/OWASP_ASVS_2009_Web_App_Std_Release.pdf, June 2009.

[15] J. Vivas, I. Agudo, and J. Lpez, "A Methodology for Security Assurance-Driven System Development," *Requirements Eng.*, vol. 16, no. 1, pp. 55-73, 2011.

[16] K. Schwaber and M. Beedle, *Agile Software Development with Scrum*. first ed., Prentice Hall PTR, 2001.

[17] R.C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. first ed., Prentice Hall PTR, 2006.

[18] S.W. Ambler, The Agile Scaling Model (ASM): Adapting Agile Methods for Complex Environments, IBM, ftp://ftp.software.ibm.com/common/ssi/sa/wh/n/raw14204usen/RAW14204USEN.PDF, Dec. 2009.

[19] M. Jazayeri, "Some Trends in Web Application Development," *Proc. Future of Software Eng. (FOSE '07)*, pp. 199-213, May 2007.

[20] S.W. Ambler. The Agile System Development Lifecycle (SDLC), http://www.ambysoft.com/essays/agileLifecycle.html, May 2013.

[21] Int'l Organization for Standardization and Int'l Electrotechnical Commission, *Information Technology: Systems Security Engineering: Capability Maturity Model (SSE-CMM)*, Std., 2008.

[22] J. Meier, A. Mackman, B. Wastell, P. Bansode, J. Taylor, and R. Araujo, Security Engineering Explained, http://www.microsoft.com/en-us/download/details.aspx?id=20528, May 2013.

[23] R. Shirey. Internet Security Glossary, Version 2. RFC 4949 (Informational), http://www.ietf.org/rfc/rfc4949.txt, Aug. 2007.

[24] R. Kissel, *Glossary of Key Information Security Terms*, Nat'l Inst. of Standards and Technology Std. NIST IR 7298, Rev. 1, http://csrc.nist.gov/publications/nistir/ir7298-rev1/nistir-7298-revision1.pdf, Feb. 2014.

[25] D. Jackson and D. Cooper, "Where Do Software Security Assurance Tools Add Value?" *Proc. Workshop Software Security Assurance Tools, Techniques, and Metrics*, pp. 14-21, Nov. 2005.

[26] S.M. Bellovin, "Security as a Systems Property," *IEEE Security & Privacy*, vol. 7, no. 5, p. 88, Sept./Oct. 2009.

[27] Software Engineering Institute—Carnegie Mellon University. Cert Secure Coding Standards, https://www.securecoding.cert.org/confluence/display/seccode/CERT+Secure+Coding+Standards, Jan. 2013.

[28] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade," *Proc. DARPA Information Survivability Conf. and Exposition (DISCEX '00)*, pp. 119-129, Jan. 2000.

[29] T. Kelly and R. Weaver, "The Goal Structuring Notation—A Safety Argument Notation," *Proc. Dependable Systems and Networks—Workshop Assurance Cases*, July 2004.

[30] J.A. McDermid, "Support for Safety Cases and Safety Arguments Using SAM," *Reliability Eng. & System Safety (Special Issue on Software Safety)*, vol. 43, no. 2, pp. 111-127, 1994.

[31] J. Goodenough, H. Lipson, and C. Weinstock, Arguing Security-Creating Security Assurance Cases, https://buildsecurityin.us-cert.gov/bsi/articles/knowledge/assurance/643-BSI.html, May 2013.

[32] R.E. Bloomfield, S. Guerra, M. Masera, A. Miller, and O.S. Saydjari, "Assurance Cases for Security," *Proc. Workshop Assurance Cases for Security*, http://www.csr.city.ac.uk/AssuranceCases/Assurance_Case_WG_Report_180106_v10.pdf, June 2005.

[33] D. Baca and B. Carlsson, "Agile Development with Security Engineering Activities," *Proc. Int'l Conf. Software and Systems Process (ICSSP '11)*, pp. 149-158, 2011.

[34] Microsoft, Agile Development Using Microsoft Security Development Lifecycle. Microsoft, http://www.microsoft.com/security/sdl/discover/sdlagile.aspx, May 2013.

[35] A. Vähä-Sipilä, Software Security in Agile Product Management, http://www.fokkusu.fi/agile-security/Software%20security%20in%20agile%20product%20management.pdf, May 2013.

[36] X. Ge, R.F. Paige, F.A. Polack, H. Chivers, and P.J. Brooke, "Agile Development of Secure Web Applications," *Proc. Sixth Int'l Conf. Web Eng. (ICWE '06)*, pp. 305-312, July 2006.

[37] M. Cohn, *User Stories Applied: For Agile Software Development*. Pearson Education, 2004.

[38] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. second ed., Addison-Wesley Longman, 2002.

[39] K. Beznosov, "Extreme Security Engineering: On Employing xp Practices to Achieve "Good Enough Security" without Defining It," *Proc. First ACM Workshop Business Driven Security Eng. (BizSec)*, Oct. 2003.

[40] B. Wastell, J.D. Meier, and A. Mackman, "Walkthrough: Creating a Threat Model for a Web Application,"technical report Microsoft Corporation, http://msdn.microsoft.com/en-us/library/ff649749.aspx, May 2013.

[41] I. Alexander, "Misuse Cases: Use Cases with Hostile Intent," *IEEE Software*, vol. 20, no. 1, pp. 58-66, Jan./Feb. 2003.

[42] G. Stoneburner, A. Goguen, and A. Feringa, *Risk Management Guide for Information Technology Systems—Recommendations of the National Institute of Standards and Technology*, Nat'l Inst. of Standards and Technology (US) Book, Online, Special Publication 800-30, http://csrc.nist.gov/publications/nistpubs/800-30/sp800-30.pdf, May 2013.

[43] Software Assurance Forum, Secure Coding. ftp://ftp.sei.cmu.edu/pub/pruggiero/bsi-swa/1/SecureCoding_PocketGuide_v2%200_05182012_PostOnline.pdf, May 2013.

[44] G. Sindre and A.L. Opdahl, "Eliciting Security Requirements with Misuse Cases," *Requirement Eng.*, vol. 10, no. 1, pp. 34-44, Jan. 2005.

[45] The Apache Software Foundation. Apache http Server Project. The Apache Software Foundation, http://httpd.apache.org/, May 2013.

[46] D. Garlan and M. Shaw, "An Introduction to Software Architecture," *Advances in Software Engineering and Knowledge Engineering*, vol. 2, pp. 1-39, World Scientific Publishing Company, 1993.

[47] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, 2005.

[48] X. Ren, F. Shah, F. Tip, B.G. Ryder, and O. Chesley, "Chianti: A Tool for Change Impact Analysis of Java Programs," *Proc. ACM SIGPLAN 19th Ann. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*, pp. 432-448, Oct. 2004.

Lotfi ben Othmane received the BS degree from the University of Sfax, Tunisia, in 1995, the MS degree in computer science from the University of Sherbrooke, Canada, in 2000, and the PhD degree from Western Michigan University (WMU) in 2010. He has extensive experience in the industry as a programmer, software architect, system analyst and technology manager in Tunisia, Canada and USA. He is currently a postdoctoral research fellow at Lero-The Irish Software Engineering Research Center, Ireland. Previously, he was a postdoctoral research associate at the Laboratory for Quality Software (LaQuSo), Eindhoven University of Technology (TU/e), The Netherlands. His main research topics include development of secure systems using Agile approach and safety and security in connected vehicles.

Pelin Angin received the BS degree in computer engineering from Bilkent University, Turkey, in 2007. She is currently working toward the PhD degree at the Department of Computer Science, Purdue University. Her research interests lie in the fields of mobile-cloud computing, cloud computing privacy and data mining. She is currently working under the supervision of Professor Bharat Bhargava on leveraging mobile-cloud computing for real-time context-awareness and development of algorithms to address the associated privacy issues.

Harold Weffers received the MSc degree in computer science in 1993, and the PDEng degree in software technology in 1995. He is a director of the Laboratory for Quality Software (LaQuSo) at the Eindhoven University of Technology. After working for the Royal Netherlands Navy and Philips, he joined the Eindhoven University of Technology in 1998 as the director of the Professional Doctorate in Engineering degree Programme in software technology. In 2008, he moved to his current position and is currently a member of a number of committees as well as a member the NEN Norm Committee on systems and software engineering (related to ISO/IEC JTC1-SC7).

Bharat Bhargava received the BE degree from the Indian Institute of Science, and the MS and PhD degrees in electrical engineering from Purdue University, West Lafayette, IN. He is currently a professor of computer science at Purdue University. His research involves mobile wireless networks, secure routing and dealing with malicious hosts, providing security in Service Oriented Architectures, adapting to attacks, and experimental studies. His name has been included in the Book of Great Teachers at Purdue University. Moreover, he was selected by the student chapter of ACM at Purdue University for the Best Teacher Award. He is a fellow of the IEEE.