Fitting Security into Agile Software Development

Kalle Rindell*, University of Turku
kakrind@utu.fi
Department of Future Technologies
20014 University of Turku
Finland

Sami Hyrynsalmi, Tampere University of Technology
sami.hyrynsalmi@tut.fi
Pervasive Computing
PL 300, 28101 Pori
Finland

Ville Leppänen
ville.leppanen@utu.fi
Department of Future Technologies
20014 University of Turku
Finland

Abstract

Success of the software development process depends on its ability to transform its objectives into requirements, and implementing these into features and functionality. Security objectives in software development are increasingly converging with the business objectives, as requirements for privacy and the cost of security incidents call for more dependable software products. Development of secure software is accomplished by augmenting the software development process with specific security engineering activities. Security engineering, in contrast to the iterative and incremental software development processes, is characterized by sequential life cycle models: the security objectives are thus to be achieved by an approach in apparent conflict with the unaugmented software development processes. In this study, to identify the incompatibilities between the approaches, the security engineering activities from Microsoft SDL, the ISO Common Criteria and OWASP SAMM security engineering models are mapped into common agile software development processes, practices and artifacts. The mapping is done primarily from the point of view of achieving the security objectives set for the software engineering process: setting security requirements for design, the implementation of the security architecture and design, and the required security verification before releasing secure software through efficient software security development process towards secure software maintenance.

Keywords: Agile, Software Engineering, Security Engineering, Methodologies

* Corresponding author
1. Introduction

Software development organizations are hard pressed to meet the increasing demand for secure software (Boehm and Turner, 2005; Subashini and Kabitha, 2011; Fitzgerald and Stol, 2014). Value-driven software development processes are seen lacking in ability to produce secure software, essentially a risk-based process. Responsibility for software security is placed on elements external to the development teams (Beznosov and Kruchten, 2004), deepening the separation of business objectives and security objectives in software development. In agile development, the lessened emphasis to preliminary planning, and the absence of fixed milestones may cause difficulties incorporating external security processes into the iterative development processes: organizations may effectively end up running a non-agile security development life cycle along the agile software development processes. Aligning the business and security objectives, and aligning and integrating the activities is necessary to avoid sacrificing neither efficiency of the agile processes, nor the long-term security objectives.

Agile software development processes call for agile organization, infrastructure and business models according to Baskerville et al. (2005). Self-organizing teams and non-deterministic implementation processes result in task implementation patterns remarkably different from those produced by sequential and pre-planned counterparts of these models. In addition to the organizational dissimilarities, security engineering processes are ultimately driven by risk rather than business value; unlike the agile development processes, they also rely on planned activities executed in a sequence as first outlined by Viega and McGraw (2002), and Howard and Brooke (2006). Sequential software development methodologies aim to reduce the security risk by executing pre-planned tasks at fixed points in the development life cycle. Lightweight, iterative, and incremental processes utilize a profoundly differently structured implementation and verification cycle; thus, security mechanisms fully integrated into agile development processes are required. There exists no inherent obstacle to utilizing agile processes to achieve the security objectives: implement the required security functionality and security assurance, and verify the absence of known security vulnerabilities (cf. Savola et al., 2012).

The agile methods improve productivity by narrowing the scope of implementation into specific features within a fixed time frame (Abrahamsson et al., 2002). Focused development allows for meticulous concentration on the quality and functionality selected into the iteration backlog. By selection of the tasks, the team and the customer can be reasonably assured that the work is done in order to achieve the objectives currently considered most important for the software product under development.

The differences between the methodologies have been broadly categorized: in one categorization approach by Boehm and Turner (2003), software development methods are considered to be either risk-driven or value-driven; hybrid models, such as Disciplined Agile Delivery by Ambler and Lines (2012), set out to reintroduce a set of planned activities – a sequential element – into the iterative work flow. To find out the reasons for the difficulties experienced by
the software and security engineers, software security processes must first be
defined, and the activities analyzed. These differences between the approaches,
values and even the paradigms of software engineering and system engineering
methodologies lead to the primary research question:

**RQ:** How are the software security engineering activities integrated into
the agile practices?

This question, and the motivation of the research, stems from empiric reports,
according to which the software development processes run into difficulties
when charged with security objectives (see e.g. Türpe and Poller, 2017; Lorünser
et al. 2018). Tying security engineering activities directly into the software
development activities, albeit only in a theoretical framework, has the potential
to make security easier to adopt for software developers, and help providing the
“security mindset” necessary for successfully and efficiently applying a security
development lifecycle in a software development project.

The research question is considered and discussed primarily from the viewpoint
of agile software engineering in the following chapters. In Chapter 2, the issues in
software security and the current adaptation of agile software security
engineering activities, practices and artifacts are examined. Chapter 0 presents
used research approach. In Chapter 4, an exhaustive list of common software
security activities are mapped into agile practices, processes and artifacts found
common in the software development industry (Licorish et al., 2016). The
security engineering activities are taken from three security development
models: Microsoft SDL (Security Development Lifecycle), ISO/IEC Common
Criteria, and the Open Web Alliance Security Project’s Software Assurance
Maturity Model (OWASP SAMM). In Chapter 5, the process and the related issues
are discussed in the perspective of achieving both the security objectives and the
business objectives of software development process. Finally, Chapter 6
concludes the article.

2. Background

2.1. **Software Security Engineering**

Software Security Engineering (SSE) introduces several system engineering
practices and activities into the software development process. In academia,
software engineering as a subdiscipline of computer science tends to
systematically exclude unquantifiable variables as ‘user’ and ‘operating
environment’ from its core (see Dijkstra, 1982). However, in practice it is clear
that in order to meet the software engineering’s objectives of delivering working
software in sustainable manner, software engineering and system engineering
will have to meet (Boehm, 2006). Mainstream software development methods
are extremely focused on the value, and as such perform poorly when facing non-
functional requirements (Ramesh et al., 2010). Functional requirements describe
what the system should do, whereas non-functional or qualitative requirements
are typically worded as how the system should perform, or concern the
treating security as a non-functional requirement has provided a convenient argument against the agile methods suitability for security engineering work (Türpe and Poller, 2017), and even suggestions that agile methods are inherently ill-suited to produce secure software (cf. Rindell et al., 2017).

Security standards are guidelines for security implementation, and several international software security regulation frameworks exist. The Common Criteria (ISO/IEC standard 15408-1:2009, 2009) has been developed to quantitatively evaluate security. It contains concrete instructions and requirements for security functionality, and suggests a framework of security objectives, to be elicited into security requirements. The objectives are also used as a basis for security risk management process, and form the outlines of the software system’s security policies. Security policies are implemented by a set of security activities and result in a plethora of functionalities which are verified by security testing and other verification methods. Some of the security activities bear a notable similarity with software quality assurance activities: these include code analyses and reviews, verification documentation and formal verification audits performed by a external certifying entity. However, treating security requirements categorically as non-functional reflects an insufficient understanding of what software security is, and how it is implemented in the software, and clearly departs from the security models provided by the ISO/IEC standards (ISO/IEC standard 15408-1, 2009; ISO/IEC standard 21827, 2008).

Agile software development is characterized by a light-weight management model: pre-planning is minimized, teams and developers have a high degree of autonomy and the development of the software resource itself is the primary target. Agile, as a descriptive adjective itself, is a combination of values and principles first expressed in The Agile Manifesto. Lightweight processes, in regard to software security engineering, is particularly reflected in principle #11, which states “[t]he best architectures, requirements, and designs emerge from self-organizing teams.”

This particular statement has been strongly criticized by well-known proponents of software architecture, such as Bellomo et al. (2014). However, despite so named, in an analysis by Séguin et al. (2012) it was noted that this particular agile principle does not qualify as a software engineering principle: it does not contain a prescriptive statement, it is not testable, nor are its consequences observable. Using these conspicuous statements as excuses to exclude the whole mainstream software development methodologies from being viable to produce secure software appears to reflect a rather poor understanding of both security engineering and agile software development methods. Skill acquired through security training and experience in security engineering is projected to have a remedial effect (Oyetoyan et al., 2016).

### 2.2. SSE Models in an Agile Context

SSE is predominantly performed by sequential models. Security development, i.e., incorporating security functionality into the software is an implementation task among others. In practice, however, the security process is a formal review
at a fixed point in time, not a continual process truly incorporated into the software development process. Injecting inspection points into the agile workflow necessarily requires pre-planning and thus has the potential to disrupt the goals of value-driven processes. Security engineering is an established tradition, with earliest formalized guidelines stemming from the United States Department of Defense in the 1980s (DoD, 1983). The highly formalized and mechanistic process of evaluating a system’s security needs was performed by applying specific evaluation criteria (DoD, 1985) into pre-classified data and pre-classified users, resulting in predetermined requirements of security functionality and security assurance. This approach was then combined with several other standards and developed into a ‘de-militarized’ version today known as The Common Criteria (ISO/IEC standard 15408-1, 2009; Common Criteria Recognition Arrangement (CCRA), 2018).

Early computer security was entirely dependent on approved and certified security products (Yost, 2015). Government regulation eventually mandated the use of security verification and assurance to prevent misuse and tampering of electronic data. Security assurance, originally meant to consist mostly of electronically produced data (i.e., various security logs) and basic functionality descriptions quickly extended into a complex set of security policies and even formalization requirements for the software products implementation itself (see e.g. Security Policy Models in ISO/IEC standard 15408-1, 2009).

At least two common maturity models address also development-time information security issues:

(1) The Software Security Engineering Capability Maturity Model (SSE-CMM) is the ISO/IEC’s heavily process-oriented security management, metrics and implementation framework (ISO/IEC standard 21827, 2008). This model originates from the Capability Maturity Model Integration, developed by the Carnegie Mellon University and currently maintained by the CMMI Institute (CMMI, 2017). As a process model, applying the CMM-based model can be very costly (Diaz et al., 2009), and can be projected not be limited to security improvement only.

(2) The Open Web Alliance Security Project’s (OWASP) open source-licensed Software Assurance Maturity Model (SAMM) contains also development-time activities, and sets best practices for security governance, construction, verification and operations.

OWASP has previously maintained also a security development lifecycle model, the Comprehensive, Lightweight Application Security Process (CLASP); this model has fallen out of common use, and replaced by SAMM. SAMM also bears distinct similarities with the Building Security In Maturity Model (BSIMM) (Synopsys Software Integrity Group, 2017). The BSIMM does not claim to specify security models or frameworks; it is published annually as a list of industry’s state of information security, surveying the current best practices in security engineering.
SAMM, complimented with OWASP’s implementation guidelines such as the OWASP Top 10 Application Security Risks (OWASP, 2018), offers guidelines to build a security framework, complete with governance and security metrics. As a framework, SAMM follows an industry-proven path: security strategy includes governance and metrics and enabled by security education; this can be considered to be equivalent of the Common Criteria’s Security Target; Security threat assessment leads to security requirements, which form the basis for security architecture; Security design and implementation (code and software resources) are verified through analyses and reviews. Security testing is thoroughly addressed, and this stage contains also release criteria for the maintenance phase; this phase contains issue management, environment hardenings and ‘Operational Enablement’, providing instructions for secure DevOps (or, DevSecOps). SAMM is divided into three maturity levels, each with specific objectives, activities, assessment criteria and expected results. These are discussed for each software security development lifecycle phase.

3. Research Description

This study is based on a conceptual-analytical approach is followed (Järvinen, 2004a, 2004b). Producing secure software by introducing security engineering processes and activities to an iterative and incremental software engineering process is challenging. To reverse this, the various phases and activities in software security development are extracted from the Microsoft SDL and the ISO Common Criteria.

![Figure 1: Simplified iterative software development process from security standpoint](image)

No specific agile methodology is used as a reference, but rather agile processes, activities and artifacts, which are linked to the security activities. Some of the terminology is derived from the Scrum method (Schwaber, 1995), currently the most commonly used mainstream software development methodology (VersionOne, 2017; Rodríguez et al., 2012). The development lifecycle is divided into six phases, and the relevant security development activities are set into agile
context. Positive and negative effects are then analyzed from the viewpoint of achieving security objectives. The concept of security objectives is derived from the Common Criteria, and visualized in Figure 1.

The agile processes, activities and artifacts selected to this study are derived from two seminal and most widely-used agile methodologies: Extreme Programming (XP) and Scrum.

The Common criteria provides a framework for evaluating the security of a software-intensive product by setting a rather complex framework. The Security Target consists of the security measures for the software itself (Target Of Evaluation, TOE) and the operating environment; for the purposes of this study, only the security objectives of the TOE are considered and the operating environment omitted.

Security objectives are met by eliciting security requirements. This contributes to creation of security specification, which in turn guides the implementation of security functionality necessary to fulfill the security policies. Some of the security functionality, such as logging, exists to provide additional security assurance. Security assurance is used to verify the existence and effectiveness of the implemented security functionality. Security assurance also functions as the basis for security metrics, and helps detecting potential security breaches later in the software’s operations phase.

The Common Criteria provides ten example software security activities to be applied during the process presented in Figure 1, which in this study are mapped to agile development, together with activities from SDL and SAMM.

The software security development life cycle models are divided into distinct phases, providing the structure for further chapters. The life cycle models phases used are pre-requirement, requirement, design, implementation, verification and release. This model is in close resemblance of the SDL’s model, leaving out the operations phase. A similar life cycle has been used in previous studies by e.g. Baca and Carlsson (2011) and Ayalew et al. (2013). The Common Criteria does not explicitly address the pre-requirement phase, but that is implied to consider the setting of the Security Target and the Security Objectives. Release phase is the preparatory stage for operations, but in iterative development, and specifically continuous delivery models, require specific operations phase activities as well. This is discussed further in Chapter 5.

<table>
<thead>
<tr>
<th>Code</th>
<th>Agile processes and artifacts</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>Iterations</td>
<td>84.2%</td>
</tr>
<tr>
<td>A2</td>
<td>Iteration planning meetings</td>
<td>76.6%</td>
</tr>
<tr>
<td>A3</td>
<td>Iteration backlog</td>
<td>75.5%</td>
</tr>
<tr>
<td>A4</td>
<td>Product backlog</td>
<td>76.1%</td>
</tr>
<tr>
<td>A5</td>
<td>Daily meetings</td>
<td>69.6%</td>
</tr>
<tr>
<td>A6</td>
<td>Iteration reviews/retrospectives</td>
<td>72.3%</td>
</tr>
</tbody>
</table>
Table 2: Agile practices

<table>
<thead>
<tr>
<th>Code</th>
<th>Agile Practices</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP1</td>
<td>Coding standards</td>
<td>81.2%</td>
</tr>
<tr>
<td>AP2</td>
<td>Test-driven development (TDD)</td>
<td>75.0%†</td>
</tr>
<tr>
<td>AP3</td>
<td>Simple design</td>
<td>74.5%</td>
</tr>
<tr>
<td>AP4</td>
<td>Continuous integration</td>
<td>73.9%</td>
</tr>
<tr>
<td>AP5</td>
<td>Refactoring</td>
<td>73.9%</td>
</tr>
<tr>
<td>AP6</td>
<td>On-site customer</td>
<td>49.5%</td>
</tr>
<tr>
<td>AP7</td>
<td>Pair programming</td>
<td>45.1%</td>
</tr>
<tr>
<td>AP8</td>
<td>Planning game</td>
<td>27.7%</td>
</tr>
</tbody>
</table>

The agile practices, process and artifacts, into which the security activities are mapped, are derived from common agile methodologies. These are presented in Table 1 and Table 2. In Table 1 the agile processes and process artifact are listed; these can be considered the ‘core’ of agile development. Table 2 contains the software development practices associated with various agile methodologies, such as Scrum and Extreme Programming (XP). In both tables, the ‘Usage’ column ranks the activity by the reported average usage.

The source survey for Table 1 has also been reported as agile practices’ significance in reducing technical debt (Holvitie et al., 2017). Managing technical deficiencies and recognized technical debt holds remarkable similarities to security engineering: many of the issues reported in this study, such as inadequacy of the architecture, structure, testing and documentation, are directly applicable to security work. In contrast, the actual features, requirements and defects represent a minority of the concerns for technical debt among the respondents, while these are central considerations in security work.

4. Security Activities in Agile Software Development

This section lists all the security activities extracted from Microsoft SDL (Microsoft, 2017; Howard and Lipner, 2006), The Common Criteria (ISO/IEC standard 15408-1, 2009; Common Criteria Recognition Arrangement (CCRA), 2018) and OWASP SAMM (OWASP, 2017). The activities for each lifecycle phase are examined and their adaptability to agile development and a matching agile activity are presented.

The agile processes, practices and artifacts presented in Chapter 3 are mapped into software security development lifecycle phases; the security activities presented in this chapter are mapped to the activities in Tables 1 and 2 by the SDLC phases, as shown in Figure 2 at the end of this chapter.

4.1. Pre-requisite Phase

The security activities in this phase are presented in Table 3. The pre-requisite phase in the SDL contains only one item: core security training. For the purposes of agile development, this part should also contain training in the agile methods: processes, activities, tools, communication procedures,

† Value was not in source and was separately calculated from a figure requested from authors
terminology and other indoctrination. Even good software engineers may be unaware of security issues and have a poor understanding of agile models; security engineers participating in software projects should be made aware how mainstream software projects are conducted and what is their work flow.
Before the project begins, SAMM suggests building security development strategy and roadmap, measuring the relative value of data and software assets, and establishing security and cost metrics; after these, the security expenditure can be assessed. SAMM also calls for establishment of security policies and security compliance; third and from the development process point of view the directly most important category at this stage is security education and guidance for all people relevant for the software and security processes. SAMM also places threat assessment onto level of organizational practices rather than project specific; a well-build organizational threat assessment framework should be able to address project-level security issues as well, and create the necessary input for security requirement gathering. At the third maturity level, SAMM advises to develop and deploy compensating controls for the threats. The threat assessment phase also provides the risk lists to be assessed in the security development projects.

In the CC framework, personnel trained in security is a part of the overall Security Target. Security training of the individuals is one of the Security Objectives to be fulfilled before the project begins. Other security objectives are typically application dependent and more difficult to generalize. However, skill is a universal requirement in both software development (Boehm and Turner, 2003a) as well as security engineering (see e.g. Oyetoyan et al., 2016). As this phase precedes the initiation of the development process, there are no directly applicable agile practices: communication with the on-site customer (AP6) may be started already at this point, and the security items added into a rudimentary version of product backlog (A4). Security training is an essential prerequisite, as skill and knowledge forms the base for all security engineering work (cf. Oyetoyan et al., 2016).

4.2. Requirement Phase

Requirement phase activities, presented in Table 3, contain activities necessary for security requirement elicitation. The SDL deals with the requirement phase activities at quite high level, and does not provide concrete resources, guidance or tools to perform them. SDL also suggests that security requirements and risks are defined only once in the project, although it is doubtful that they will remain static throughout the project. SDL’s approach of setting quality gates is hardly security specific at all, yet this is something that can be addressed through agile practices of Coding Standards. The Common Criteria also stays at rather abstract level, and defines two types of security activities: definition of Security Functional Requirements (SFR) and Security Assurance Requirements (SAR). In

<table>
<thead>
<tr>
<th>Source</th>
<th>Security activity</th>
<th>Agile activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDL</td>
<td>Core Security Training</td>
<td>AP6</td>
</tr>
<tr>
<td>CC</td>
<td>Set security target and objectives</td>
<td>A4, AP6</td>
</tr>
<tr>
<td>SAMM</td>
<td>Strategy &amp; metrics</td>
<td>n/a</td>
</tr>
<tr>
<td>SAMM</td>
<td>Policy &amp; compliance</td>
<td>n/a</td>
</tr>
<tr>
<td>SAMM</td>
<td>Education &amp; guidance</td>
<td>n/a</td>
</tr>
<tr>
<td>SAMM</td>
<td>Threat assessment</td>
<td>A4, AP6</td>
</tr>
</tbody>
</table>
the ISO standardization framework fulfilling both these requirement types is essential in achieving the security objectives, and verifying this.

**Table 4: Requirement phase activities**

<table>
<thead>
<tr>
<th>Source</th>
<th>Security activity</th>
<th>Agile activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDL</td>
<td>Establish Security Requirements</td>
<td>A1, A4</td>
</tr>
<tr>
<td>SDL</td>
<td>Create Quality Gates/Bug Bars</td>
<td>A1</td>
</tr>
<tr>
<td>SDL</td>
<td>Perform Security and Privacy Risk Assessments</td>
<td>A1, A4</td>
</tr>
<tr>
<td>CC</td>
<td>Definition of SFR and SAR</td>
<td>A1, A4</td>
</tr>
<tr>
<td>SAMM</td>
<td>Security requirements</td>
<td>A1, A4</td>
</tr>
</tbody>
</table>

SAMM’s security requirement activities give a logical process how to gather and elicit security requirements: partially it relies on the business requirements, which are then evaluated against the compliance guidance for security requirements; this has been created in the pre-requirement phase. At the second level, an access control matrix is created, and the security risk list from previous phase used to complement the list of security requirements. At third level, SAMM calls for business-oriented security activities of security management for supplier contracts, and an audit program for security requirements.

Agile software development is all about change. Effectively this means efficient and continual requirement management. In Table 1 the only agile activity directly addressing requirement elicitation and prioritization is the Planning Game (Beck, 2000). With 27.7 % adoption rate this technique, originating from the XP methodology, sets an example how requirement elicitation is done iteratively. Security requirement elicitation techniques have been surveyed by Tondel et al. (2008), although this study does not address the issue specifically from agile software developer’s point of view. Requirement elicitation process must be thorough and systematically identify all the relevant security functionality and assurance requirements; iterative approach (A1) directly supports this process.

Agile methods are extremely efficient in prioritizing the implementation queue: identified items are given workload or complexity estimates, and are then placed into the product backlog (A4). Work items will also get assigned an explicit Definition of Done. Eventually, depending on their priority, they will be picked up for the iteration backlog, get implemented and verified. Quality of the requirements is typically ensured through rigorous validation process: methods, such as INVEST for user stories (natural language requirements) and SMART for backlog items (Wake, 2003) are used to review the requirements and transform them into implementable features and functionality.

### 4.3. Design Phase

Design-time activities are listed in Table 5: Design phase activities. Both SDL and CC describe design activities at quite high abstraction level. Both do, however, backtrack the design to requirement elicitation and requirement management. SDL also recommends threat modelling. Threat modeling tools, such as the one
made available by Microsoft⁴, makes this task more manageable. However, modeling a large software system with multiple computing resources, data storages and communication interfaces is quite likely to become a burden, added with the task of maintaining that model through the iterations. In addition, the model should be reviewed as any other artifact in order to maintain a credible security assurance.

Table 5: Design phase activities

<table>
<thead>
<tr>
<th>Source</th>
<th>Security activity</th>
<th>Agile activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDL</td>
<td>Establish Design Requirements</td>
<td>A1, A3, A4, AP3, AP6</td>
</tr>
<tr>
<td>SDL</td>
<td>Perform Attack Surface Analysis/Reduction</td>
<td>A1, AP3</td>
</tr>
<tr>
<td>SDL</td>
<td>Use Threat Modelling</td>
<td>A1, AP3</td>
</tr>
<tr>
<td>CC</td>
<td>Cross-analysis of TOE designs</td>
<td>A1, AP3</td>
</tr>
<tr>
<td>CC</td>
<td>Vulnerability analysis and flaw hypothesis</td>
<td>A1, AP3</td>
</tr>
<tr>
<td>CC</td>
<td>TOE design analysis against the requirements</td>
<td>A1, AP3</td>
</tr>
<tr>
<td>SAMM</td>
<td>Security architecture</td>
<td>A1, AP3, AP6</td>
</tr>
<tr>
<td>SAMM</td>
<td>Design review</td>
<td>A1, AP3</td>
</tr>
</tbody>
</table>

SAMM contains two categories for this phase: the security architecture and a design review. Security architecture consists of a list of practical procedures: first, maintaining a list of recommended software frameworks and applying security principles to the design; second, security services and infrastructure are to be identified and promoted, and security design patterns identified from the architecture. The third level does not include development-time architectural activities; it calls for formal reference architectures and platforms are to be established and frameworks, patterns and platforms validated. Design review, at the first maturity level, should include identification of attack surfaces and design analysis against the security requirements. Requirements for the second maturity level are inspection for complete provision of security mechanisms and the organizational task of deployment of design review service for project teams. The third level is again project specific, containing the activities of developing data-flow diagrams for sensitive resources and establishing release gates for design review.

Agile development support security design notably well: iterations (A1) allow revisiting the earlier decisions and the iteration backlog (A3) as necessary. Agile practices promote simple design (AP3); all security designing and reviews are performed under this activity. TDD (AP2) and pair programming (AP7) convey the security design into implementation and verification phases. Iterations (A1) implicitly offer opportunities to enhance the security design in case the requirements or environmental factors have changed. Having customer on-site for communication (AP6) also supports security design process. Some key components of the security architecture may stem from customer requirements, and maintaining close customer communication should prove beneficial in creating and maintaining this important security artifact.

4.4. Implementation Phase

Table 6 contains the security activities for the implementation phase, necessary to achieve the security objectives. Mainstream software engineering is increasingly dependent on a large set of interconnected and connected tools. Modern development IDEs integrate directly into packet management servers and code repositories; code repositories are in turn part of Continuous Integration and Continuous Delivery (CI/CD) services, and automated unit tests are executed upon each commit. Automated CI/CD systems deploy the tested components into staging areas, from where the code will eventually be released into production after integration and security testing. Although a lot of the security-related implementation-time activity can be automatized, a human-performed static code review is considered very effective. Continuous integration (AP4) is a common agile practice.

<table>
<thead>
<tr>
<th>Source</th>
<th>Security activity</th>
<th>Agile activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDL</td>
<td>Use Approved Tools</td>
<td>A1</td>
</tr>
<tr>
<td>SDL</td>
<td>Deprecate Unsafe Functions</td>
<td>A1, AP1</td>
</tr>
<tr>
<td>SDL</td>
<td>Perform Static Analysis</td>
<td>A7</td>
</tr>
<tr>
<td>CC</td>
<td>Analysis and checking of processes and procedures</td>
<td>A7</td>
</tr>
<tr>
<td>SAMM</td>
<td>Implementation review</td>
<td>A7</td>
</tr>
</tbody>
</table>

Static reviews are the only implementation-time activity in SAMM, and the model gives quite coherent way to conduct them: review checklists are created, and high-risk code is somehow identified and reviewed in detail. At second level automated analysis tools are to be used, and the code analysis to be integrated into the development process. On the third level, code analysis automation is to be made application-specific and release gauges for the review established.

Coding standards (AP1), although established already in the pre-requirement phase, are an important quality improvement practice. It also directly contributes towards security by enabling code reviews and making the source code more structured. Pair programming (AP7) is a very effective quality and security improvement practice (Paulk, 2001); pair programming also acts as a substitute for formal reviews (Cockburn and Williams, 2000). Iterative development (A1) gives opportunities for refactoring (AP5) which also works as security improvement measure; activities and practices such as daily meetings (A5) and underlying TDD (AP2) also can be used to work towards the improvement of software security.

4.5. Verification Phase

Security verification activities are presented in Table 7. In order to achieve security objectives and effectively manage security requirements, the iterative security verification faces two unique issues:

(1) Returning of the failed items into the backlog, accounting requirement and design changes.
(2) Automating the security testing, or performing it in such manner that each potentially shippable iteration has gone through the security verification process.

Test-Driven Development (AP2) is an obvious enabler for security verification practices. Also, proper training in security and testing methods can be considered essential in incorporating the security testing into the software testing suite. Costly and time-consuming fuzz testing, advocated by the SDL, is in practice a quite specialized operation, appropriate for organizations developing APIs and operating systems; application developers should have less use for fuzzing.

The security verification phase is best covered in security engineering methodologies, and has the least direct counterparts in agile activities. Performing these security activities as part of security assurance procurement is to be done already at the requirement specification phase, and the security requirements inserted into the product backlog (A4). Security verification can also be performed during a security specific iteration. Continuous integration (AP4) automates and helps facilitate testing; daily meetings (A5) also often cover testing issues.

<table>
<thead>
<tr>
<th>Source</th>
<th>Security activity</th>
<th>Agile activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDL</td>
<td>Perform Dynamic Analysis</td>
<td>AP4</td>
</tr>
<tr>
<td>SDL</td>
<td>Perform Fuzz Testing</td>
<td>A4</td>
</tr>
<tr>
<td>SDL</td>
<td>Conduct Attack Surface Review</td>
<td>AP2, A4</td>
</tr>
<tr>
<td>CC</td>
<td>Verification of proofs</td>
<td>AP2, A4</td>
</tr>
<tr>
<td>CC</td>
<td>Independent functional testing</td>
<td>AP2, AP4</td>
</tr>
<tr>
<td>CC</td>
<td>Test case and test result review</td>
<td>AP2, A4</td>
</tr>
<tr>
<td>CC</td>
<td>Penetration testing</td>
<td>A4</td>
</tr>
<tr>
<td>CC</td>
<td>Verification of processes and procedures</td>
<td>A4</td>
</tr>
<tr>
<td>SAMM</td>
<td>Security testing</td>
<td>AP2, AP4</td>
</tr>
</tbody>
</table>

SAMM follows its logic of relying on the security requirements also in the verification phase: security test cases are drawn from them. SAMM also calls for penetration testing at the basic maturity level, an activity that requires specific knowledge and tools, and is typically performed by security engineering experts. Only at level two does SAMM require use of automated security testing tools and integration of security testing into development process. Similarly to the implementation verification, on third level, the automation is to be made application-specific and release gates for security testing established.

### 4.6 Release Phase

At the end of each iteration a potentially shippable program increment is released, and the activities in Table 8 are performed. This is a security specific phase in the security development life cycle, and there is in general no direct
correspondence to the agile activities. Many of the security artifacts are produced in this phase (review documentation, certification), and the security activities are performed only once – unless Continuous Integration (AP4) is used. The retrospect (A6) is an important quality improvement measure in iterative development, and directly applicable to security engineering as well; it does not, however, produce the artifacts that are the main product of this SDLC phase. Security engineering activities taking place at later phases of the software lifecycle are crucial to the security objectives, but separate from the development process. Continuous integration (AP4) also extends into the release of the software. On-site customer (AP6) may also participate in release-time security activities.

Table 8: Release phase activities

<table>
<thead>
<tr>
<th>Source</th>
<th>Security activity</th>
<th>Agile activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDL</td>
<td>Create an Incident Response Plan</td>
<td>AP4, AP6</td>
</tr>
<tr>
<td>SDL</td>
<td>Conduct Final Security Review</td>
<td>AP4, AP6</td>
</tr>
<tr>
<td>SDL</td>
<td>Certify Release and Archive</td>
<td>AP4, AP6</td>
</tr>
<tr>
<td>CC</td>
<td>Analysis of guidance documents</td>
<td>AP4, AP6</td>
</tr>
<tr>
<td>SAMM</td>
<td>Issue management</td>
<td>AP4, AP6</td>
</tr>
<tr>
<td>SAMM</td>
<td>Environment hardening</td>
<td>AP4, AP6</td>
</tr>
<tr>
<td>SAMM</td>
<td>Operational enablement</td>
<td>AP4, AP6</td>
</tr>
</tbody>
</table>

Common Criteria uses the term “guidance documents” in reference to security-related maintenance documentation intended for software’s operations in the operational environment. In the Common Criteria, security is verified through two processes: security functionality is verified by functional testing, and security assurance by documentation reviews. At the project’s inception, one of the security-related objectives is setting of the Evaluation Assurance Level (EAL). This level, ranging from 1 (most basic) to 7 (the most rigorous) defines the amount of documentation to review. The formality requirement for the software code itself increases accordingly. The development-time documentation consists of five parallel documentation tracks, number and level of which is increasing as the EAL rises. At EAL 1, only basic functional specification is required. EAL 2 adds a basic design document, and security architectural description; it also requires the basic functional specification to be augmented with specification of security-enforcing functionality. Each level brings additional documentation requirements up to EAL 5, after which the documentation or formalization requirements do not increase. The maintenance-specific documentation is not included in the development-time documentation requirements.

SDL is less concerned with the internal security documentation and concentrates on certification and operational maintenance of the software. This is well in accordance to claims that majority of the cost in software engineering incurs after the release phase (Glass, 2001). This is also reflected in SAMM’s trio of activity categories directed at security of the post-development part of the software lifecycle: SAMM calls for issue management, environment hardenings and enabling the operations teams for security, before the software can be released. The last of these provides actual tasks for the software development phase: at first level, critical security information should be captured, and
procedures (operational instructions) for typical application alerts documented. At second level, change management procedures are created, related to the issue management, and formal operational security guides created and maintained. Third level again concentrates on the business goals, and calls for an audit program and code signing.

5. Discussion
This section is divided into following subsections: first, the key findings of the mapping are discussed; then, some theoretical and practical implications are laid out; finally, the limitations of this research approach and the prospects opened by it are presented.

5.1. Key Findings
The aim of this study is to explore how the software security engineering activities are integrated into the existing agile practices. First, it seems that development-time security activities form the basis for the software security. The security functionality, security assurance and operational documentation are created by the development-time processes, forming the base for the later phases of the software lifecycle.

Figure 2 gives an overview of the difficulty in direct mapping of sequential lifecycle models to agile development practices; very few agile activities are confined into a single lifecycle phase, and even this division appears somewhat artificial in a dynamic agile model.

![Figure 2: Agile activities mapped into security development life cycle](image)

Pre-requisite tasks are mostly policy and process oriented; security training, training in the agile processes, and establishing practices such as coding standards takes place in this phase – and occasionally these practices may have to be revisited even during the course of the implementation process. In iterative development, the phases from here onward are iterative. The phases are bound together by most important requirement elicitation method: communication with the customer, reflected by “on-site customer”. Product backlog is also established already at the pre-requisite phase with the general requirements, which are then complimented with the project specific items at later phases. Test-driven development binds the phases together; pair programming, even if utilized only at crucial points and as an “enhanced code review practice”, is
another iteration-spanning activity useful from the requirement phase onward, until end of implementation phase.

Simple design, iteration planning, and placing the planned items into an iteration backlog bind the requirements and design together; implementation is augmented with daily communication between the developers, and also availability of the customer communications. Planning game (AP8) can be used to properly prioritize the security-related items when compiling the iteration backlog (A3). Continuous Integration, together with TDD, provides security verification management and proper coordination with iteration backlog by identifying the items that require rework; also refactored items requiring security verification are handled through these activities – automatically, with proper tooling. CI process also produces the releases after the verification, to undergo any release-phase security activity after completion of the development processes.

Security development lifecycle models have a strong emphasis on security verification. This is also the phase with least common activities between the agile and security engineering activities. The solution to this discrepancy is twofold: both strong integration of security testing into the functional testing and CI/CD processes is required; also, a level of pre-planning is required: the security engineering activities required to achieve the security objectives have to be recognized early in the development process and the activities placed into the product backlog.

The security lifecycle models do not appear to consider maintenance phase at all, although they do include specific security activities to be performed at release time. SAMM, the most recent of the three security models included, does define a distinct set of activities for issue management, environment hardening and operational enablement. All these three categories are in the traditional responsibility area of maintenance or operations teams, although this distinction and clear division of labor and responsibility is increasingly being dissolved by the emerging continuous delivery models. Continuous delivery infused with security functions, such as DevSecOps, relies on processes similar to the ones in verification phase (see Chapter 4.5): issue management is the function responsible for returning the security-related items into the product backlog, and seeing that they get appropriately prioritized. As the software increments keep iteratively deployed, also the operational environment hardening and operational enablement functions need to be performed iteratively. Continual development models imply a convergence of development and operations; in practice this is achieved by a heavy degree of automation and virtualization. Virtualizing the operational environment may dramatically improve the system availability in case of a serious security incident: the affected environment is simply wiped and new patched instances installed, minimizing the system down time.

Agile methods are geared towards requirement management and getting features implemented and delivered; however, security experts still keep reporting that this is not the case with security objectives (see Türpe and Poller,
2017). While value-driven agile development processes have certain unique shortcomings, fulfilling security requirements could be as simple as a matter of prioritization. As long as the security personnel and security requirements are external, the security objectives are under the threat of getting poorly realized, if at all. This can only be changed by increasing the awareness of the security engineering processes and including the security features and especially verification activities into the development process itself. As long as security engineering is external to the software development, also security objectives are in danger of remaining external – at the cost of potentially inadequate software security.

5.2. Theoretical and Practical Implications

This study has certain implications to development of theoretical understanding in secure software engineering and agile development models. First, this study shows that coexistence of secure practices and agile methodologies seems to be a feasible scenario. However, as it has been discussed, the fusion of these two approaches to a single well-functioning methodology is not simple or likely a generalized solution. Nevertheless, this study provided a framework for aligning software security engineering with agile methods.

Second, the presented framework might help practitioners to acknowledge software secure engineering requirements in daily work. While the study does not offer an off-the-shelf method, the presented discussion might help to select, adapt and improve the engineering practices in companies. Synergy between software engineering and security engineering practices can lower the cost of achieving the security objectives, which in turn helps reduce the lifetime cost of the software being developed.

5.3. Limitations and Future Work

Naturally, this study has certain limitations that should be noted. Firstly, the paper is based on a conceptual-analytic approach (Järvinen, 2004b) and, therefore, the selection of analyzed concepts are crucial. To tackle this threat, we have selected the most widely used agile and SSE practices as the starting point for the work. However, in the future work, more concepts should be included into the analysis. Secondly, the presented model has not been externally validated and experts of software secure engineering should be utilized in the future work.

In agile development, the methodology encourages heavy customization of the activities and processes, based on the individual and often unique requirements of the customer, product, laws and regulations, or the operational environment; this work should function as an enabler for organizations in tailoring a process based on the task at hand. Creating prescriptive frameworks may not be feasible in the diverse field of software engineering, as their applicability even within a specific industry may be very limited; however, it will be very interesting to observe an empiric application of this activity mapping.
In addition, this study opens some interesting avenues for further studies. The presented model should serve as a starting point for inquiries to define a model for agile secure engineering. In addition, further studies should focus on empirical assessment of the feasibility as well as weaknesses of agile security engineering. Identifying clustered security activities and the related agile software development practices provides basis for more efficient and effective software development methods, and more flexible ways to produce secure software.

6. Conclusion

This article has provided a comprehensive mapping of the most common and tested security engineering practices into the agile software development practices used in XP and Scrum. The mapping was made based on extant literature, although a definitive framework for this is yet to be empirically validated. The approach was to take the most prominent existing security development life cycle models and place the agile activities, processes and artifacts into the life cycle phases presented in those models. The resulting matrix should help developers and researchers to improve software security by two main steps: (1) select the security engineering activities necessary to achieve the security objectives, based on security needs; (2) ensure that the agile practices used in the organization support these activities. The continual improvement approach, inherent to agile development, should be applied to further augment the development process and security initiative.

Implementing software with security objectives requires combining software engineering and security engineering processes. This combining of processes should take place on three levels: (1) providing training for the individuals; (2) executing security requirement management; and (3) by integrating and involving the security activities, tools and the security experts into the software development process. With an acceptable level of preliminary planning the security-related work items are to be placed into the product backlog, and completed at a convenient time during the iterative development process.

Achieving security objectives in software development requires security engineering. Software security is an investment: it requires training, tools and time. Integrating security engineering directly into the software development activities, rather than executing it as detached processes, is intuitively an obvious benefit both economically and technically. This study has provided a framework for this alignment, and suggested ways to overcome the potential difficulties in this alignment process. Software development is agile, and security engineering will have to follow suit.

References

PUBLICATIONS.


Common Criteria Recognition Arrangement (CCRA) (2018). The common
criteria.  


