A Survey on Requirements and Design Methods for Secure Software Development*

Muhammad Umair Ahmed Khan and Mohammad Zulkernine

School of Computing
Queen's University
Kingston, Ontario, Canada K7L 3N6
{umair & mzulker}@cs.queensu.ca

Technical Report No. 2009–562

This report should be cited as follows:

M. U. A. Khan and M. Zulkernine, *A Survey on Requirements and Design Methods for Secure Software Development*, Technical Report No. 2009 – 562 , School of Computing, Queen's University, Kingston, Ontario, Canada, August 2009.

ABSTRACT

Traditionally, security of software is not considered from the very beginning of a software development life cycle, and it is only incorporated in the later stages of development as an afterthought. As a consequence, there are increased risks of security vulnerabilities that are introduced into software in various stages of development. To avoid security vulnerabilities, there are many secure software development efforts in the directions of secure software development life cycle processes, security specification languages, security requirements engineering processes, secure design languages, and secure design guidelines. In this paper, we compare and contrast various secure software development processes based on a number of characteristics that such processes should have. We also analyze security specification languages with respect to desirable properties of such languages. Furthermore, we identify activities that should be performed in a security requirements engineering process to derive comprehensive security requirements. We compare different security requirements engineering processes based on these activities. Finally, we investigate the state-of-the-art in secure design languages and secure design guidelines. Our analysis shows that many of the secure software requirements and design methods lack some of the desired properties. The comparative study presented in this paper will provide guidelines to software developers for selecting specific methods that will fulfill their needs in building secure software applications.

Contents

List of Tables

1 Introduction

Security of software has become an important issue with the increasing integration of software in various aspects of human society. Software is considered to be secure if it does not allow the confidentiality, integrity, and availability (widely referred to as CIA) of its data, code, or service to be compromised [1]. However, most of today's software are not secure and contain security vulnerabilities that can be exploited by people with malicious intent to cause financial and/or physical harm.

Traditionally, security of software is not considered from the very beginning of a software development life cycle (SDLC), and it is only incorporated in the later stages of development as an afterthought. As a consequence, there are increased risks of security vulnerabilities that are introduced into software in various stages of development. The traditional approach to security of software has led to penetrate-and-patch approaches. In a penetrate-and-patch approach, security of software is assessed by attempting to break into the software from its environment by exploiting known security vulnerabilities. If such a penetration attempt is successful, a patch is developed and deployed for the vulnerability that allowed the break-in. Penetrate-and-patch approaches have many major drawbacks [2-6]. First, developing and deploying a patch to remove an error that causes a security vulnerability can be up to 200 times more expensive [7] than if the error had been removed as soon as it was introduced during development. Second, there is no assurance that the developed patch itself does not have any new security vulnerabilities (assuming that the traditional approach of software development was followed). Third, there is no guarantee that all the existing security vulnerabilities have been identified. Finally, major harm could have already occurred before a security vulnerability is even detected [6].

Secure software engineering aims to avoid security vulnerabilities in software by considering security aspects from the very beginning and throughout the SDLC. Secure software engineering is the process of designing, building, and testing software so that it becomes secure. *Software security* concerns are different from "application security" issues. Application security is about protecting software after it is developed and deployed. It usually includes input filters, intrusion detection systems, firewalls, and other protection mechanisms [2-5].

Software security includes secure software development life cycle (SSDLC) processes and secure software development (SSD) methods. A SSDLC process is either a SDLC process augmented with various SSD methods or a set of independent SSD methods. An SSD method provides ways to incorporate security in software during its development. An SSD method maybe a security specification language, security requirements engineering process, secure design specification language, set of secure design guidelines, secure design pattern, secure coding standard, and software security assurance method (*e.g.*, penetration testing, static analysis for security, and code reviews for security).

In this paper, we analyze the existing SSDLC processes, requirements methods, and design methods for secure software development based on their strengths and weaknesses. We first identify a number of characteristics of a SSDLC process that make it complete and effective. These characteristics are then used to compare and contrast various SSDLC processes. We also present a detailed comparison of security specification languages based on a number of desirable properties. This comparison focuses on identifying languages that can effectively specify security requirements. Moreover, we identify activities that should be performed as part of any security requirements engineering process. Based on these activities, we compare various security requirements engineering processes and identify their strengths and weaknesses. Finally, we provide an analysis of the state-of-the-art of different secure design languages and guidelines.

The analysis presented in this paper can be useful in a number of ways. The comparison of various SSD methods will help software developers in selecting a particular SSDLC process, security specification language, security requirements engineering process, or secure design language for their particular development scenarios. The identified properties of software security specification and design languages can be useful to translate one specification language into another. Such a translation is particularly useful when a user of one language intends to use security tools developed for other languages.

The rest of this paper is organized as follows. Chapter 2 defines the software security terms used in this paper. Chapter 3 presents an in-depth critical analysis of SSDLC models or processes. Chapter 4 analyzes methods for

software security requirements engineering. Methods for secure software design are discussed in Chapter 5. Chapter 6 concludes this paper by summarizing our findings and identifying the corresponding open research issues.

2 Definitions Related to Software Security

A number of terms are used in software security literature, sometimes with varying meanings. In this chapter, we explain the major terms used in this paper.

Asset

An asset can be anything considered valuable [8] and should be protected. An asset is the "target of threats, the possessors of exposures, or the beneficiary of countermeasures" [9]. According to Verdon and McGraw [10], an asset can be a system component, data, or a complete system. The National Institute of Standards and Technology [11] defines an asset to be "a major application, general support system, high impact program, physical plant, mission critical system, or a logically related group of systems". We consider something as an asset that can be harmed. An asset can be code (binary or source) of the software, data or information used or stored by the software, and the service provided by the software. Successful exploitation of these assets can lead to financial loss, physical injury, and sometimes even death.

SSD Activity

We define an SSD activity as the use or execution of a SSD method (as defined in Chapter 1).

Vulnerability

Flechais *et al.* [12] define a vulnerability as a part of the software which is prone to undesirable action. A vulnerability is the result of a software defect that an attacker can exploit to cause harm [10]. More precisely, "a software vulnerability is an instance of an error in the specification, development, or configuration of a software such that its execution can violate the security policy. … a software vulnerability can result from many errors, including errors in the specification, design, or coding of a system, or in environmental assumptions that do not hold at runtime" [8].

Software Security Error

The term *error* has been very rarely used in software security literature. Most of the authors use the term *vulnerability* and do not distinguish between vulnerability and error. Sometimes the term *flaw* is also used instead of error [13]. In software reliability engineering, the term error is defined as an incorrect or missing action by a person that results in a fault [14]. A software security error is a tangible manifestation of a mistake in any of the SDLC artifacts (requirement specifications, design, or source code) of a piece of software that leads to a vulnerability [8, 13, 15]. A software security error can be one of the following, *if it is related to security threats.*

a) *Requirement specification error* - An incorrect or missing requirement in the requirement specifications due to a mistake made in the requirement specification phase of the SDLC.
b) *Design error* - An incorrect logical decision (the decision itself or the representation of a decision) in the design due to a mistake made in the design phase of the SDLC.
c) *Source code error* - An incorrect representation of the design decisions in the source code due to a mistake made in the implementation phase of the SDLC.

Software Security Requirement

A software security requirement [15], sometimes also referred to as a countermeasure or safeguard [9-11], can be defined as a requirement needed to avoid a specific software security error during development. More specifically, a software security requirement is a control or constraint which if not implemented may lead to a vulnerability. As a

software security error can be present in any of the SDLC artifact, therefore, a software security requirement can be specified for any SDLC artifact [15].

The term *security requirement* has also been used as a synonym to *attack scenario* in the security literature [16, 17]. However, while we believe that attack scenarios can provide a basis to derive a security requirement, they themselves cannot and should not be considered as a security requirement.

Attack

An attack is a deliberate attempt by a malicious entity or person to harm an asset of a piece of software by exploiting an existing vulnerability. An attack can be successful or unsuccessful. An attack is also called intrusion.

Attack Specification

An attack on a piece of software may consist of a single or multiple steps. An attack specification is the representation of the sequence of these steps defined formally using a specification language. An attack specification is sometimes also referred to as an attack signature, attack scenario, or intrusion scenario [11, 17, 18].

Risk

The risk posed by a particular attack is the product of the probability of that attack taking place and the combined damage that can be caused to different assets if the attack is successful [10, 11, 15].

Risk Analysis

Risk analysis is the process of calculating risk. It can be performed during any phase of software development. Risk analysis takes into account knowledge about existing vulnerabilities, attacks, possible impact of attacks, and probability of attacks [10, 15].

Threat Modeling

Threat modeling [19] is a method to identify possible threats (exploitable or not) to a piece of software. It takes into account the assets of the software and potential attackers and their goals.

Attack Surface

An attack surface of a piece of software is the collection of entry or access points into that software. An entry (or access) point is defined as a possible way in which the user can interact with the software and possibly attack it. Examples of entry (access) points include user interfaces, configuration files, and data files. A large number of entry points indicate that the software has a large attack surface and vice versa.

3 SSDLC Processes

As the idea of incorporating security into software from the very beginning of development has gained acceptance, various secure software development (SSD) methods were suggested. However, software are still being developed by following the conventional SDLC models or processes, and various SSD methods are used at different stages as deemed fit to the developers. Many secure SSDLC processes have been proposed in the literature which we summarize in Section 3.1. We identify characteristics that a SSDLC process should have and provide a detailed comparison of the processes based on these characteristics in Section 3.2.

3.1 Summary of Existing SSDLC Processes

A number of SSDLC processes have been reported in the literature with varying characteristics. These are McGraw's SSDLC process [2], Trustworthy Computing Security Development Life Cycle or Microsoft Software Development Life Cycle (MS SDL) [20, 21], Appropriate and Effective Guidance for Information Security (AEGIS) [12, 22], Secure Software Development Model (SSDM) [23], Aprville and Pourzandi's Secure Software

Development Life Cycle Process [16], Secure Software Development Model (SecSDM) [24], Software Security Assessment Instrument (SSAI) [4, 25], mapping of the most prevalent vulnerabilities to various secure design and implementation activities [26], Comprehensive, Lightweight Application Security Process (CLASP) [27], Secure Software Development Process Model (S2D-ProM) [28], and Team Software Process for Secure Software Development (TSP-Secure) [29]. The following paragraphs summarize these SSDLC processes.

McGraw's Secure Software Development Life Cycle Process

McGraw [2] proposes to augment a SDLC process (waterfall or iterative) with certain SSD activities. In essence, McGraw's process focuses on incorporating security requirements, performing risk analysis during different development phases, and applying security assurance methods such as risk-based security testing, static analysis, and penetration testing. The process also suggests using risk analysis during the design phase. For the security assurance phase, McGraw suggests using abuse cases and security requirements to guide penetration testing.

Microsoft Software Development Life Cycle (MS SDL)

There are two important facets to the MS SDL [20, 21]. First, there is a policy that a security expert overlooks and guides the development of secure software. Second, it follows the principle of secure by design (secure software development), secure by default (software should run with the least necessary privileges to minimize the attack surface), secure by deployment (tools and guidance should accompany the software to help users in using the software securely), and communication (from developers to end users about vulnerabilities and patches). This principle is also called SD^3 + C. The following paragraphs summarize the secure by design aspect as the rest of the aspects are beyond the scope of this paper.

MS SDL is itself a spiral model augmented with various SSD activities. During the requirement specification phase, MS SDL proposes to use customer (end-user) demands to identify security objectives and required security features. The incorporation of these security features are guided by industry standards and certification criteria. The assigned security expert also specifies the exit criteria for each phase of development during this phase. MS SDL suggests the following activities to be performed during the design phase: identification of the components that are critical to security, identification of design techniques/guidelines (*e.g.,* application of least privilege principle, minimization of attack surface), identification of entry or access points of the attack surface, threat modeling and risk analysis on a component-by-component basis, identification of security requirements to mitigate threats, identification of components that need special attention during testing and code reviews, and completion criteria for the software.

MS SDL recommends following the secure coding standards in the implementation phase. The MS SDL process places emphasis on the security assurance by recommending testing, static code analysis using tools, and code reviews during the final stages of the implementation phase. After implementation is complete, the completed software is again verified for security by performing security code reviews and security testing. This security testing specially focuses on the security critical components of the software (*e.g.,* entry points of the attack surface).

Appropriate and Effective Guidance for Information Security (AEGIS)

AEGIS [12, 22] is a SSDLC process based on the spiral model and focuses on the specification of security requirements by identifying assets and performing risk analysis. The requirement specification and design phases of AEGIS are closely related and are not separately addressed by the authors. The authors propose to have four design sessions between the developers and the stakeholders of the software. The first and second sessions model the software assets and their relationships and identify the high level confidentiality, integrity, and availability requirements (*i.e.,* what security properties should be preserved for which assets) by using abuse cases. In the third session, risk, vulnerabilities, and threats to the software are identified. The fourth design session selects the security requirements to remove the identified vulnerabilities.

AEGIS also suggests a risk analysis method to be used during the third and fourth design sessions. This risk analysis method has the following major steps:

a) Determination of vulnerabilities.
b) Determination of cost and likelihood of an attack in the deployed environment (including roles played by people interacting with the software and the tasks that will be performed on the software).
c) Selection of security requirements based on security expert's advice.
d) Cost-benefit assessment of the selected security requirements.
e) Comparison between the cost and likelihood of each attack against the cost of security requirements.
f) Selection of security requirements based on cost effectiveness.

Secure Software Development Model (SSDM)

SSDM [23] is a process that incorporates various security activities into a waterfall software development life cycle model. According to SSDM, threat modeling should be performed in the requirement specification phase. The threat modeling results in lists of all potential vulnerabilities and attacks. Such lists can be helpful to avoid these vulnerabilities throughout development.

After threat modeling, a security policy is to be defined that should clearly state how security will be achieved. Such a policy, as outlined by SSDM, is a set of high level management decisions such as avoiding errors throughout the development process and the correction of errors as soon as they are detected. For the design phase, SSDM advises to follow the security policy. Penetration testing is the only SSD activity for the security assurance phase, as identified by SSDM.

Aprville and Pourzandi's Secure Software Development Life Cycle Process

Aprville and Pourzandi [16] propose a SSDLC process based on their experience while developing an instant messaging software. According to their process, the first step in the requirements specifications phase is to identify the high level security objectives (confidentiality, integrity, and availability) of the software being developed by considering its deployment environment. For low level security objectives, threat modeling should be used to help in building a set of security requirements. These security requirements can then be prioritized according to the results of risk analysis. In the design phase, the authors recommend using UMLsec [6] to represent design decisions and security design patterns. For the implementation phase, the authors suggest to choose a programming language that best fulfills their security objectives. In addition, special emphasis should be placed on avoiding buffer overflow and format string vulnerabilities. They also point out that already verified algorithms should be used, especially for cryptography. For the security assurance phase, code reviews, static vulnerability code scanners, ad-hoc unit and system security testing, fuzz testing, and various testing tools should be used.

Secure Software Development Model (SecSDM)

SecSDM [24] uses risk analysis in the requirement specification phase to prioritize threats to the software. High level security objectives such as confidentiality, integrity, and availability are then identified based on these threats. In the design phase, security features are selected that can mitigate threats and achieve security objectives. SecSDM proposes following secure coding standards in the implementation phase.

Software Security Assessment Instrument (SSAI)

SSAI [4, 25] is a set of activities which use certain resources and tools to help in developing secure software. The first resource SSAI provides is an online vulnerability database [30] containing information about various vulnerabilities and their exploits and mitigations. The second SSAI resource is a security checklist that can be used to guide development in a secure fashion. Authors present details on how to develop such a checklist and potential items that can be included [4]. The third resource is a list of publicly available static code scanning tools. SSAI also provides flexible modeling framework (FMF) which is a modeling tool. The FMF can be used to develop models and verify them for desired properties using model checking. The authors claim that this tool can "delay" the effects of state space explosion problem as it verifies the desired properties first on individual components and then on the strategic combinations of these components. Lastly, SSAI provides a property-based testing tool (PBT) which uses the security properties specified in the security checklist or FMF as a basis to test the software.

Hadawi's Set of Secure Development Activities

Hadawi [26] identifies 25 common vulnerabilities to avoid during development. He also proposes a number of design and implementation security requirements that, if incorporated, would help in avoiding these vulnerabilities. During the implementation phase, the only SSD activity is the selection of an appropriate (secure) programming language. For the security assurance phase, Hadawi recommends using security code reviews and static code analysis tools.

Comprehensive, Lightweight Application Security Process (CLASP)

Comprehensive, Lightweight Application Security Process (CLASP) [27] is a set of SSD activities which are categorized according to the roles performed during development. CLASP also suggests the assignment of a security expert from the beginning of development. For the requirement specification phase, CLASP advocates performing risk analysis and threat modeling. An attacker profile should also be developed based on the potential attacker and their resources. Moreover, the attack surface and security features that need to be implemented should be identified.

According to CLASP, risk analysis and threat modeling should be performed again during the design phase. CLASP proposes to annotate class diagrams with security information. In the security assurance phase, CLASP recommends performing security code reviews, security code scanning, and security testing. CLASP also provides a list of common vulnerabilities with comprehensive information about how and when they can be introduced during development and how to avoid them.

Secure Software Development Process Model (S2D-ProM)

S2D-ProM [28] specifies multiple possible strategies to advance from one development phase to another. The main idea behind this process is to provide developers with flexible options. These strategies always result in an SSD activity. The process proposes to conduct risk analysis during requirement specification, design, and implementation phases. Risk analysis, according to S2D-ProM, can be performed in different ways for each development phase. For example, error checklists or personal experience can be used in the requirement specification phase, model checking or design reviews can be used in the design phase, and testing or code reviews can be used in the implementation phase. The identified risks can then be mitigated using various strategies (*e.g.*, defining security rules, using security mechanisms).

S2D-ProM also provides multiple options while advancing from one development phase to another. For example, security standards or personal experience can be used to develop requirement specifications, a security modeling language or security patterns can be used to construct design from requirements, and secure coding rules or personal experiences can used to develop source code from design. S2D-ProM does not specify whether only one strategy should be used while moving from one phase to another or multiple strategies can be used at the same time.

Team Software Process for Secure Software Development (TSP Secure)

TSP-Secure [29] addresses security in the following three ways: planning for security, managing quality and security throughout the development life cycle, and educating developers about security related aspects. During the planning phase the team identifies security goals and produces a detailed plan to guide development. Development activities in the plan may include, but are not limited to, identifying security risks, eliciting security requirements, secure design, code reviews, unit testing, fuzz testing, and static code analysis. The team may choose any SSD activity as deemed necessary. According to TSP-Secure, one team member performs the role of a security manager who is responsible to ensure that all security related activities are taking place.

3.2 Detailed Comparison of Existing SSDLC Processes

Each of the proposed SSDLC processes has its strengths and weaknesses. Gregoire *et al.* [31] have compared MS SDL and CLASP based on the structure of the process, the resources provided by the process, and the SSD activities performed during different phases of the software life cycle (project inception, education, analysis, design, implementation, testing, verification, deployment, and support phases). Modifying the comparison criteria used by Gregoire *et al.* [31], we propose that an SSDLC process should have the following characteristics: specification of

SSD activities for requirements engineering, design, implementation, and security assurance phases, resources available to the developers, use of artifacts of an SSD activity in the later phases of development, and usage in the industry. We further elaborate these characteristics and compare and contrast various SSDLC processes based on these characteristics in the following subsections. Table 1 presents a comparison of these SSDLC processes.

3.2.1 SSD Activities for Requirements Engineering

Every SSDLC process should explicitly specify the use of certain SSD activities for each of the development phases (*i.e.*, requirements engineering, design, implementation, and security assurance). Gregoire *et al.* [31] have only provided a comparison of activities performed in MS SDL and CLASP. We, however, identify SSD activities that must be performed during requirements engineering, design, implementation, and testing phases to develop more secure software. These SSD activities are based on the ones proposed by different SSDLC processes. In the requirements engineering phase, the following activities should be included:

a) Threat modeling to identify possible threats to the software so that appropriate security features and mechanisms can be specified.
b) Requirements specification review/inspection to find security errors by possibly using a checklist of potential requirements specification security errors.
c) Security requirements specification using a security requirements specification language to remove the identified errors.
d) Risk analysis to prioritize the identified security requirements and security assessment to evaluate the security state of the requirement specifications [2, 15].

Almost all the SSDLC processes (except SSAI [4, 25], MS SDL [20, 21], and set of SSD activities by Hadawi [26]) perform threat modeling and/or risk analysis during the requirements specification phase. Moreover, all processes except SSAI, SSDM [23], and Hadawi's set of activities recommend defining security features to mitigate identified threats. A formal review of the specified requirement specifications is only advocated by SSAI which uses model checking. None of the processes propose the use of any security requirements specification language. Note that we do not consider abuse cases used by McGraw [2] as a security requirements specification language. In summary, many processes recommend a similar set of SSD methods to be used during requirements engineering.

3.2.2 SSD Activities for Design

For the design phase, the following SSD activities should be performed to have a more secure design:

a) Secure design guidelines and principles should be followed while developing the initial design. Secure design patterns should either be followed or used for guidance. Secure design decisions should be specified using a secure design specification language.
b) Threat modeling should be performed on the initial design to identify possible threats to the software. It should be noted that the results of threat modeling during requirements specification and design phase would be different due to the additional information present in the design.
c) Design reviews should be performed to identify security errors. It is possible that an error found in the design is due to an error in the requirements specification. In such a case, the error in the requirements specification should be fixed.
d) The identified threats and errors should be used as a basis to specify security requirements for the design.
e) Risk analysis should be performed to prioritize the security requirements and for design. Security of design should also be assessed [2, 15].

Most of the processes pay very little or no attention to SSD activities in the design phase. For example, SSAI [4, 25] does not propose any specific SSD method, McGraw [2] only advises performing risk analysis, Hadawi [26] only outlines a set of secure design guidelines to be followed, Apvrille and Pourzandi [16] only recommend the use of UMLsec [6] and security patterns, and so on. S2D-ProM [28] does suggest multiple SSD activities to be performed

to develop a secure design, however, each of these activities are related to a unique strategy. Only MS SDL [20, 21] and CLASP [27] provide a comprehensive set of SSD methods for the design phase. Both of these processes do not explicitly propose the use of a specific security design language, however, they do advise to incorporate security decisions in the design. MS SDL recommends identifying security critical components so that these can receive additional attention.

3.2.3 SSD Activities for Implementation

For the implementation phase, a secure programming language should be selected to minimize security errors. Moreover, secure coding standards and guidelines should be followed. Apvrille and Pourzandi [16], Hadawi [26], and S2D-ProM [28] suggest that an implementation language that is secure should be used for development. MS SDL [20, 21], SecSDM [24], McGraw's SSDLC process [2], S2D-ProM, CLASP [27], and Hadawi propose to follow secure coding standards and/or guidelines. MS SDL goes further and suggests that certain security assurance activities should be performed during implementation.

3.2.4 SSD Activities for Security Assurance

Many different SSD activities can be performed during the security assurance phase. These activities include security testing (*e.g.,* penetration testing [2]), security static analysis using code scanning tools, security code reviews, risk analysis, and security assessment. These activities do not guarantee security of the software but identify errors and vulnerabilities. Moreover, these activities complement each other. We propose that all of the above mentioned SSD activities for security assurance should be performed to achieve greater assurance regarding the security of software. With the exception of AEGIS [12, 22], SSDM [23], and SecSDM [24], all the processes recommend using multiple security assurance methods such as security testing, code reviews, static code analysis, and so on.

3.2.5 Resources

Certain secure development resources should be available to the developers. Such resources can be lists of potential errors, vulnerabilities, and possible mitigations for each error or vulnerability. This knowledge is helpful in avoiding the errors that may lead to vulnerabilities in the first place [31]. Knowledge about possible errors and vulnerabilities is also helpful in assessing the security state of an SSDLC artifact [15] and deciding when to move to the next phase of development.

Many online databases, such as Common Vulnerabilities and Exposures [32] with over 32,000 reported vulnerabilities, are maintained to provide information about reported vulnerabilities. Vulnerabilities have also been categorized with their possible mitigations [27, 30, 32-35]. However, this available knowledge is not organized with respect to the criticality of vulnerabilities, time of introduction during development, and software application domain. As is evident from Table 1, only CLASP [27], SSAI [4, 25], and Hadawi [26] provide developers with any resources. These three SSDLC processes present potential vulnerabilities, their mitigations, and development guidelines. However, resources provided by CLASP are more extensive. Moreover, MS SDL [20, 21] and McGraw [2] provide secure design guidelines.

3.2.6 Use of Artifacts of an SSD Activity in the Later Phases of Development

The purpose of performing any SSD activity is to increase the security posture of the SDLC artifact on which the activity is performed. However, if the product of any SSD activity can also be used in later phases of development, it would increase the effectiveness of that particular SSD activity. For example, if the abuse cases derived by the abuse case specification activity (SSD activity for the requirement specification phase) can be used to generate test cases in the testing phase, it would mean that this SSD activity is more effective as compared to other SSD activities whose artifacts are not or cannot be used in the later phases. Both AEGIS [12, 22] and McGraw's SSDLC process [2] specify abuse cases without explicitly mentioning their future use within the SDLC. Nevertheless, abuse cases have been used to generate test cases [36, 37]. CLASP [27] uses security requirements to guide testing.

SDLC Processes / Characteristics	McGraw SDL [2]	MS SDL [20, 21]	AEGIS [12, 22]	SSDM [23]	Apvrille & Pourzandi [16]	SecSDM [24]	SSAI [4, 25]	Hadawi [26]	CLASP [27]	S2D-ProM [28]	TSP Secure [29]
SSD Activities for Requirements Engineering	Specification of abuse cases and security requirement. Risk analysis.	Identification of interfaces, security objectives and required security features. Definition of exit criteria.	Risk analysis. Identification of assets, abuse case and high level security requirements.	Threat modeling. Specification of security policy.	Identification of high level security objectives. Threat modeling. Specification of security requirements. Risk analysis.	Risk analysis. Identification of security objectives.	Modeling using flexible modeling framework (FMF). Model checking for security.	None	Risk analysis. Threat Modeling. Identification of attackers and attack surface. Specification of misuse cases with their mitigations, security features.	Following security standards. Risk analysis. Identification of errors. Specification of security features.	Threat modeling. Specification of abuse cases. Risk Analysis.
SSD Activities for Design	Risk analysis.	Identification of critical components, attack surface, design methods, and completion criteria. Threat modeling. Risk analysis. Definition of secure architecture.	Design decisions based on security requirements.	Following the security policy.	Use of UMLsec and security patterns.	Identification of security features to fulfill security objectives.	None	Following design guidelines.	Following design guidelines. Annotation of class diagrams with security information. Threat modeling. Risk analysis.	Using security modeling language, security patterns. Risk analysis. Design reviews. Model checking.	Design patterns. State machine design and verification.
SSD Activities for Implementation	None	Following secure coding standards. Code reviews. Use of testing and static code analysis tools.	None	None	Use of a secure programming language. Use of established security algorithms. Avoiding buffer overflow and format string vulnerabilities.	Following secure coding standards.	None	Use of a secure programming language. Following secure coding standards and guidelines.	Following secure coding guidelines.	Following secure coding standards. Use of a secure programming language.	Following secure programming standards and guidelines.
SSD Activities for Security Assurance	Risk-based and penetration testing. Use of static code analysis tools.	Code reviews. Security testing.	None	Penetration testing.	Code reviews. Ad-hoc unit and system security testing. Fuzz testing. Use of static code analysis tools.	None	Use of static code analysis tools. Property-based testing using the PBT tool.	Code reviews. Use of static code analysis tools.	Code reviews. Use of static code analysis tools. Security testing.	Risk analysis. Code reviews. Use of static code analysis tools.	Fuzz testing. Penetration testing. Use of static code analysis tools. Code reviews.
Resources	Secure design guidelines.	Secure design guidelines.	None	None	None	None	Vulnerabilities and their mitigations, a list of security code scanning tools, and potential items for a security checklist to guide development.	Common vulnerabilities to avoid and secure design and implementation guidelines.	Secure design and implementation guidelines. A comprehensive list of more than 200 types of vulnerabilities with their possible mitigations.	None	None
Use of Artifacts of an SSD Activity in the Later Phases of Development	Abuse cases can be used to derive test cases.	None	Abuse cases can be used to derive test cases.	None	None	None	None	None	Security testing is based on requirements.	None	None
Usage in the Industry	Reported	Reported	Not Reported	Not Reported	Not Reported	Not Reported	Not Reported		Reported	Not Reported	Not reported

Table 1. Comparison of SSDLC Processes

3.2.7 Usage in the Industry

A crucial test for the applicability of any SSDLC process is its usage in the industry. A model or process which fulfills the needs of the industry would be more widely used. Although many of the existing processes might have been applied in industrial practice, only MS SDL [20, 21], McGraw's SSDLC process [2], and CLASP [27] have been reportedly used in industry. MS SDL has a slight edge over CLASP and McGraw's SSDLC process as it was developed based on the experiences gained from previously developed commercial software.

4 Software Security Requirements Engineering

Requirements engineering is the foundation of developing quality software. It has been estimated that an error introduced in the requirements specification phase, if not removed immediately, can cost up to 200 times more to correct in later stages of development [7]. The main objective of software security requirements is to specify that the confidentiality, integrity, and availability of the software should be preserved. Usually, these requirements specify the security features needed for the software system. However, such requirements are not precise enough and need to be more explicit about who can do what and when [38].

Research reported in the field of security requirements engineering are primarily on security specification languages and security requirements engineering processes. In Section 4.1, we first summarize existing security specification languages and then compare them based on the desirable properties of these languages. In Section 4.2, we compare and contrast various security requirements engineering processes based on a set of activities that should be part of such a process.

4.1 Security Specification Languages

Many different languages (or notations) have been proposed in the literature to represent security specifications (security requirements or attack specifications). Some authors use the term *security requirement* as a synonym to *attack specification* [17, 39]. However, a security requirement [2, 8, 9, 11, 15] is different from an attack specification in that a security requirement is a control or constraint which if not implemented may lead to a vulnerability. On the other hand, an attack specification represents the sequence of the steps of an attack. Nevertheless, it is true that attack specifications can form a basis to identify vulnerabilities in software. Moreover, attack scenarios are also helpful in developing intrusion-aware software [40].

A security specification language can be a software specification language used to specify attacks (AsmL [41] and UML state charts [18]), an extension of a software specification language to represent attacks (Misuse Cases [42], Abuse Cases [43], AsmLSec [39], and UMLintr [44]) and security requirements (UMLsec [6], SecureUML [45], Secure Tropos [46], and Misuse Cases [42]), or an attack specification language (*e.g.,* STATL [47] and Snort Rules [48]).

In this section, we first summarize (Section 4.1.1) and then compare (Section 4.1.2) the software specification languages and their extensions that have been used in the context of security. Similarly, from the various attack specification languages reported in the literature [17], we analyze only those that have been employed with software specification languages.

4.1.1 Summary of Existing Security Specification Languages

UMLsec

UMLsec [6] is an extension of UML for secure systems development and uses stereotypes, tags, and constraints to specify security requirements. Stereotypes serve as labels for UML model elements to introduce information to the model and specify constraints that have to be met by the model. Tags are associated with stereotypes and are used to explicitly specify a simple property of a model element. UMLsec defines 21 stereotypes to be used to represent the following security requirements: fair exchange (no cheating between parties), non-repudiation (an action cannot be denied), role-based access control, secure communication link, confidentiality, integrity, authenticity, freshness of a message (*e.g.,* nonce), secure information flow among components, and guarded access (use of guards to enforce

access control). Seven of these stereotypes have associated tags and nine stereotypes have constraints. These stereotypes can be used for use case diagrams, class diagrams, state charts, activity diagrams, sequence diagrams, and deployments diagrams to specify security requirements in a UML model (for both requirement specifications and design). Tools have also been developed that allow the developer to model using UMLsec and then verify these models (using model checking) [49].

SecureUML

SecureUML [45] is another extension of UML that focuses on specifying role-based access control policies (these policies can be considered as security requirements) in a model. SecureUML proposes nine stereotypes that can be used to annotate a class diagram with role-based access control information. SecureUML uses the Object Constraint Language (OCL) to specify constraints for resources, actions, and permissions. Contrary to UMLsec, these constraints can be specified according to the individual software's requirements.

Secure Tropos

Secure Tropos [46] can be used for developing secure software and is an extension of the Tropos development methodology [50]. Secure Tropos uses the notions of actor (person(s), organization(s), or software), goal (objectives which actors want to achieve), soft goal (a goal whose fulfillment cannot be explicitly determined), task (a particular way to accomplish a goal), resource (physical or data), security constraint (specified as high level statements), secure goal (used to fulfill a security constraint), secure task (a particular way to achieve a secure goal), and secure resource (a resource that is related to a security constraints, secure goal, secure task, or another secure resource). An actor can depend on another actor to accomplish a goal or soft goal, perform a task, or deliver a resource. The SecureTropos notation can be used to represent security constraints (requirements) on interactions between actors during the requirement specification phase.

Misuse Cases

A misuse case [42] is a special kind of UML use case that describes undesirable behavior of the software. A misuse case is initiated by a special kind of actor called mis-actor (*e.g.*, actor with malicious intent, bad luck, and honest actor performing a mistake). Misuse cases and mis-actors can be used to elicit more use cases to neutralize the threats posed by misuse cases. Misuse cases and mis-actors are represented in solid black color to distinguish them from use cases and actors, respectively. Two special relations called "prevents" and "detects" relate use cases and misuse cases. The authors provide a stepwise process to develop a use case diagram (including misuse cases). This process can also be used iteratively, if needed. According to this process, first use cases and actors and then misuse cases and mis-actors should be specified. After this, the potential "include" relationships between misuse and use cases should be identified because often the software's functionality is used to attack it. Next, new use cases should be specified to detect or prevent misuse cases. These new use cases form the high level security requirements of the software and they are called as "security use cases" by Firesmith [51].

Abuse Cases

Another way to specify undesirable behavior of a piece of software using UML use case diagrams is to develop an abuse case [43] model. An abuse case model specifies harmful interactions using actors and abuse cases. There is no notational difference between the components of a UML use case diagram and an abuse case model. The authors propose that all the potential harmful interactions should be specified *e.g.*, different potential approaches to perform a denial of service attack should be specified as separate abuse cases. The authors recommend using a tree structure to elicit these multiple approaches. This adds more detail to the model and allows in identifying all possible security measures. Details about actors such as their resources, skills, and objective should also be included as text. According to the authors, abuse cases can be used to guide design and testing.

UMLintr

UMLintr [44] is an extension of UML that uses stereotypes and tags to specify intrusions (attacks) using use case diagrams, class diagrams, state charts, and package diagrams. Authors divide attacks into four different types. Each type is represented as a stereotyped package. There are three stereotypes defined for classes and twelve for the use case diagram. Stereotypes for classes also have tags.

Abstract State Machine Language (AsmL)

AsmL is an extended finite state machine-based executable software specification language which has also been used to specify attack scenarios [41]. The authors argue that due to the extended finite state machine-based nature of AsmL, attacks with multiple steps can be specified in AsmL. Such attack scenarios can be automatically translated into Snort rules which can then be used with an extension of the IDS Snort [41]. Such attack scenarios are able to capture more attacks with multiple steps using context information. Snort rules, the standard input for Snort, cannot represent attacks with multiple steps.

AsmLSec

AsmLSec [39] is an extension of AsmL that was developed to specify attack scenarios [43]. AsmLSec uses states, events, and transitions to represent attacks. Each transition has a source and a destination state, a set of conditions to be met to fire the transition, and actions to be performed in case a transition is fired. The attack scenarios represented in AsmLSec can be automatically translated into AsmL through a specially developed compiler. An IDS has also been developed that takes the translated attack scenarios as input.

UML State Charts for Security

UML state charts (without any extension) have been used to specify attacks [18]. The authors have related the attacks specified in state charts to Snort rules. These state charts can be manually translated into Snort rules and then used with an extension of the IDS Snort. By using state charts, they are able to represent complex multiple step attacks which are not possible to be represented in an ordinary Snort rule.

Snort Rules

Snort is a widely used network intrusion detection system (IDS). It uses attacks scenarios specified as rules [48] to detect attacks over the network. A Snort rule specifies what action should be taken if the rule is matched to a network packet, the source and destination IP addresses and ports, the protocol of the observed network, and direction of network packet. A number of options can also be specified. These options range from logging a message to searching for a particular string in the packet.

STATL

STATL [47] is an extended finite state machine-based executable attack specification language. STATL uses two main constructs to specify an attack: state and transition. Each transition must have an associated event which when occurs fires the transition. Transitions also have optional actions that are performed once a transition is fired. State and transition specifications can also have executable code within them. A development environment for STATL is also available which can be used to, among other things, visualize the specified attack scenario as a state machine.

4.1.2 Detailed Comparison of Existing Security Specification Languages

Different security specification languages have different properties. In the following subsections, we identify desired properties of these languages and then use these properties to compare the languages. These properties are as follows: mandatory requirements for a security requirements specification language [6], similarity with software specification languages, use of specified security requirements or attack scenarios in later phases, and tool support.

The following paragraphs present a comparison of various security specification languages based on these properties, while Table 2 summarizes this comparative study.

Mandatory Requirements for a Security Requirements Specification Language

Juerjens [6] proposes seven "mandatory" requirements that must be met by any security requirements specification language based on UML. Based on these requirements, we use the following three requirements as a basis to compare the existing security specification languages (see column 2 in Table 2).

a) Ability to formulate basic security requirements such as confidentiality, integrity, and availability (high level security requirements).
b) Ability to consider various situations that may lead to different possibilities of attacks (usage scenarios and potential attack scenarios).
c) Ability to represent security mechanisms such as access control and low level security requirements such as constraints.

Our analysis shows that only UMLsec [6] and Secure Tropos [46] fulfill all three requirements. Misuse cases [42] only lack low level details.

Similarity with Software Specification Languages

To integrate security concerns during software development seamlessly, it is essential that security specification languages are as close as possible to the software specification languages. The major reason for this is that developers would have to spend very little time in becoming well versed in these languages. For example, all the languages based on UML (such as Abuse cases [43], Misuse cases [42], UMLsec [6], SecureUML [45], and UMLintr [44]) would be much easier to learn for the software practitioners who know UML. The languages based on the notion of extended finite state machine and the ones with syntax close to existing programming languages (*e.g.*, STATL [47]) may gain popularity more quickly among developers.

Desired Properties / Security Specification Languages	Mandatory Requirements by Juerjens [6]			Similarity with Software Specification Language (Yes/No)	Use of Specified Security Requirements and Attack Specifications in a Later Phase (Name of the Phase)	Tool Support (Yes/No)
	Ability to Formulate Basic Security Requirements (Yes/No)	Ability to Represent Usage Scenarios (Yes/No)	Ability to Represent Security Mechanisms and Low Level Security Requirements (Yes/No)			
UMLsec [6]	Yes	Yes	Yes	Yes	Testing [53]	Yes [6, 49]
SecureUML [45]	No	No	Yes (RBAC Policy only)	Yes	Model Checking and Testing [54]	Yes [55]
Secure Tropos [46]	Yes	Yes	Yes (Constraints)	Yes	No	Yes [56]
Misuse Cases [42]	Yes	Yes	No	Yes	Testing [36, 37]	No
Abuse Cases [43]	No	Yes	No	Yes	Testing [36, 37]	No
UMLintr [44]	No	Yes	No	Yes	Intrusion Detection [44]	Yes [52]
AsmL [41]	No	Yes	No	Yes	Intrusion Detection [41]	Yes [41]
AsmLSec [39]	No	Yes	No	Yes	Intrusion Detection [39]	Yes [39]
UML State Charts for Security [18]	No	Yes	No	Yes	Intrusion Detection [18]	Yes [18]
Snort Rules [48]	No	Yes	No	No	Intrusion Detection [48]	Yes [48]
STATL [47]	No	Yes	No	No	Intrusion Detection [47]	Yes [47]

Table 2. Comparison of Security Specification Languages

Use of Specified Security Requirements or Attack Specifications in a Later Phase

The main purpose of specifying attack scenarios and security requirements is to ensure the incorporation of security concerns in the software being developed. However, their effectiveness would increase if they can also be used later. For example, security requirements can be used to derive test cases, and attack scenarios can be used with intrusion detection systems. Abuse cases [36, 37], Misuse cases [36, 37], UMLsec [53], and SecureUML [54] have been reported to be helpful in guiding testing. If attack scenarios are specified using attack specification languages in the requirements specification phase, they can be used as input for intrusion detection systems. While intrusion detection is not a part of software development, it plays an important role in the security of software.

Tool Support

Another concern with respect to various security specification languages is the availability of supporting tools. Languages with tool support are more likely to be used in the industry. Nearly all of the languages have some kind of tool support. This can be a design environment (for UMLintr [52], UMLsec [6, 49], SecureUML [55], UML state charts [18], and Secure Tropos [56]) or an intrusion detection system (for AsmL [41], UML state charts [18], Snort rules [48], AsmLSec [39], and STATL [47]).

4.2 Security Requirements Engineering Processes

Security requirements specification languages are used to specify security requirements. However, a process is needed to derive these requirements. Such a process is usually called a security requirements engineering process and has multiple activities. Some major activities are identification of threats, analysis of risk due to these threats, specification of security mechanisms to be used, requirement inspections to identify security requirement errors, and elicitation and prioritization of low level security requirements (to remove errors). To derive precise security requirements, such activities are necessary because the level of security required is different from software to software (depending on user requirements and software domain).

4.2.1 Summary of Existing Security Requirements Engineering Processs

Secure Tropos

The major activities to elicit security requirements in Secure Tropos [46] can be summarized as follows:

a) Actors, goals, soft goals, their dependencies, and their security constraints (requirements) are identified.
b) An in-depth analysis of security constraints imposed on goals is performed. There might be many alternative ways to achieve a goal and there could be many sub goals to accomplish a goal. The objective of such analysis is to identify further goals and soft goals.
c) Based on the newly identified goals, new security constraints are identified.
d) Tasks are defined to achieve sub goals (goals at the leaf nodes).

In Secure Tropos, security constraints are the security requirements imposed on the operation of the software. After the completion of the above mentioned four activities, one would have identified the goals of the software, how it will achieve them, and what are the constraints that need to be respected while accomplishing these goals. Si* is a modeling tool for Secure Tropos [56].

Security Quality Requirements Engineering (SQUARE)

SQUARE [57] is a process for eliciting, categorizing, and prioritizing security requirements. SQUARE has the following nine activities.

a) Agree on security definitions.
b) Identify safety and security goals.
c) Select requirement elicitation techniques.
d) Document architectural diagrams, use cases, misuse cases, attack trees, assets, and services.
e) Elicit safety and security requirements.

f) Categorize safety and security requirements.
g) Perform risk analysis (assessment).
h) Prioritize requirements based on their importance to the software.
i) Perform requirement inspections.

Comprehensive, Lightweight Application Security Process (CLASP)

CLASP [27] also proposes a security requirements engineering process to formulate security requirements. Viega [58] elaborates on this process and suggests that for each security service a set of specific, measurable, attainable, and traceable requirements should be defined. The security requirements engineering processes within CLASP has four activities which are outlined as follows.

a) Resources and/or assets and roles (owners and users of resources/assets) are identified.
b) Resources and assets are categorized into classes according to the level of security required.
c) The interactions that a resource or asset will have over its lifetime are identified.
d) Based on the interactions of resources/assets, different security services are identified. CLASP recommends authorization, authentication and integrity, confidentiality, availability, and accountability as possible security services.

Process by Haley et al.

Haley *et al.* [38] propose a process for security requirements engineering. The activities to be performed in this process are as follows:
a) Functional requirements are identified.
b) Security goals are identified.
c) Assets that can be harmed are identified.
d) Threats to assets are identified.
e) Organizational policies are applied to the functionality and assets of the system. The result is a set of goals that should be achieved to accomplish the security goals identified in Activity b.
f) Based on the goals identified in Activity e, security requirements (constraints on functionality) are identified.
g) Structured formal and informal argumentations are used to verify that the software's security goals are met. Otherwise, the stakeholders may decide to use a weaker constraint, intrusion detection, and attack recovery.

Security Requirements Engineering Process (SREP)

SREP [59] is a process that defines activities and roles to develop security requirements. The authors also recommend having a security resources repository (SRR) to store all reusable security requirements elements. The activities of SREP are as follows:
a) Agree on security definitions.
b) Identify vulnerable and/or critical assets (SRR can be used).
c) Identify security objectives based on the organizational policy (SRR can be used).
d) Identify threats using misuse cases or attack trees (SRR can be used).
e) Assess risk.
f) Elicit security requirements (SRR can be used).
g) Categorize and prioritize requirements.
h) Requirements inspection.
i) Adding reusable components to SRR.

The roles defined by SREP roles are business modeler, security requirement engineer, risk expert, security expert, security developer, quality assurer, and the inspection team.

4.2.2 Detailed Comparison of Existing Security Requirements Engineering Processs

The main objective of any security requirements engineering process is to identify security requirements that can reasonably ensure security of the developed software. A security requirements engineering process should have the following activities (A1 to A23). This set of activities is based on the activities suggested by various security requirements engineering processes (except for A20 which is suggested by us) [27, 38, 46, 57, 58, 59]. Each of these activities results in information that can help in identifying security requirements. Performing all of these activities might not be absolutely essential and the sequence of these activities is also flexible. We propose that activities A17 to A23 should be performed iteratively till a satisfactory level of security for the requirement specifications is attained.

	Activities																						
	A1	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**	**A8**	**A9**	**A10**	**A11**	**A12**	**A13**	**A14**	**A15**	**A16**	**A17**	**A18**	**A19**	**A20**	**A21**	**A22**	**A23**
Security Requirements Engineering Processes	High level functional requirements	Deployment environment	Identify assets and resources	Identify the value of assets and resources	Identify users	Identify potential attackers	Identify attacker's interest	Identify attacker's capabilities and resources	Specify use cases	Specify misuse scenarios	Identify potential threats	Identify security goals	Specify high level security requirements	Specify security policy and constraints	Specify low level functional requirements	Define exit criteria	Requirements inspection	Risk analysis	Perform security assessment	Specify low level security requirements to remove errors	Perform cost/benefit analysis	Categorize and prioritize low level security requirements	Include selected low level security requirements
Secure Tropos [46]	X	X			X				X			X	X	X									
SQUARE [57]	X	X	X	X					X	X	X	X	X			X	X	X	X		X	X	X
CLASP [27, 58]		X	X	X	X	X	X	X					X										
Haley *et al*. [38]	X		X								X	X	X	X	X			X					
SREP [59]			X								X	X	X	X	X		X	X				X	

Table 3. Comparison of Security Requirements Engineering Processes

A1 Specification of high level functional requirements (services provided by the software).
A2 Identification of the deployment environment of the software (including interactions with other software).
A3 Identification of assets and resources of the software.
A4 Valuation of assets and resources of the software.
A5 Identification of users of the software.
A6 Identification of potential attackers of the software.
A7 Identification of attacker's interest in the resources/assets of a piece of software.
A8 Identification of attacker's capabilities and resources.
A9 Specification of use cases.
A10 Specification of misuse scenarios.
A11 Identification of potential threats to the software (attack trees [60-62] or anti-models [63] can be used to decompose threats into smaller ones).
A12 Identification of security goals (derived through discussion with the stakeholders).
A13 Specification of high level security requirements such as security mechanisms to be incorporated.

A14 Specification of security policy and constraints on the working of the software derived by discussing with the stakeholders (*e.g.*, access control policy).
A15 Specification of low level functional requirements (including requirements of security mechanisms *e.g.*, password length).
A16 Definition of exit criteria depending on the security state of the requirement specifications (the security state can be calculated by using security index [15]).
A17 Requirements inspection to identify security errors.
A18 Risk analysis.
A19 Performing security assessment of the requirement specifications (if the assessment meets the exit criteria then exit).
A20 Specification of low level security requirements to remove security errors.
A21 Cost/benefit analysis (security errors vs. security requirements).
A22 Categorization and prioritization of low level security requirements.
A23 Inclusion of selected low level security requirements in the requirement specifications.

5. Secure Software Design

Software design represents the static structure and dynamic behavior of software. It is necessary to make design decisions that are secure and do not introduce any security vulnerabilities in the completed software. Many secure design languages have been proposed to help the developer represent secure design decisions [6, 45, 46]. A few works have also been reported in the literature providing developers with secure design guidelines [64-67]. These guidelines provide guidance to avoid design flaws that led to security vulnerabilities in previously developed software. In this chapter, we first briefly discuss existing secure software design languages. Then we compare various sets of secure design guidelines.

5.1 Secure Design Languages

Many of the languages for specifying security requirements are also used for specifying design. This is due to the fact that low level requirements are very close to static and dynamic design. These languages (*e.g.*, UMLsec [6], SecureUML [45], and Secure Tropos [46]) have already been discussed in Section 4.1. There are two major concerns that should be considered in selecting a secure design language. These concerns are the variety of diagrams available to represent a design from various aspects and levels of abstraction and the availability of tools. UMLsec provides a variety of diagrams and has tools available. SecureUML can also be used for secure software design; however, it is limited to representing only role-based access control notions in a UML class diagram. Secure Tropos proposes to use Agent UML capability diagrams, plan diagrams, and agent interaction diagrams [68]. These diagrams are similar to UML activity diagrams (plan and capability) and sequence diagrams (agent interaction).

5.2 Secure Design Guidelines

Secure design guidelines are suggestions and principles based on the experience the developers have gained while solving recurring software security problems. Presumably, the first set of such guidelines were proposed by Saltzer and Schroeder [65]. Bishop [1] has elaborated on these guidelines by providing examples. Viega and McGraw [66] and Howard and LeBlanc [67] have also proposed sets of secure design guidelines. There are many guidelines that are common among these sets. Peine [64] has analyzed the aforementioned sets of guidelines and suggested a consolidated set of guidelines, adding a few new ones, and removing some existing ones. These four sets of guidelines are summarized in Table 4 following Peine's work.

Each of the sets has some unique guidelines. In his work, Peine did not include three guidelines recommended each by Saltzer and Schroeder and by Howard and LeBlanc (listed in the first and third columns in Table 4 with grey background). Moreover, we disagree with Peine's analysis of the guideline "Remember that security features != secure features" proposed by Howard and LeBlanc (represented in italics in the third column of Table 4) as equivalent to "secure the weakest link not the easiest and most obvious one". Hence, we have added the meaning not considered by Peine in the remarks column.

Saltzer & Schroeder [65]	Viega & McGraw [66]	Howard & LeBlanc [67]	Peine [64]	Remarks about Guidelines
Economy of mechanism.	Keep it simple.		Do not be more general than necessary.	Keep the design as simple and small as possible.
		Minimize attack surface.	Minimize attack surface.	Minimize attack surface.
Psychological acceptability.			Make the secure way the easy way.	The security mechanisms should be easy to use.
Fail-safe defaults.	Fail securely.	Fail to a secure mode.	Stay secure in case of failures.	In case of software failure, access should be denied.
Complete mediation.				Every access to every object must be checked for authority.
Open design.		Never depend on security through obscurity alone.	Do not depend on attacker's ignorance.	The design should not depend on being a secret.
Separation of privilege.	Practice defense in depth.	Use defense in depth.	Use several layers of defense.	Having multiple conditions or mechanisms to access an object (more secure as compared to a single condition *e.g.*, two passwords).
Least privilege.	Follow the principle of least privilege.	Use least privilege.	Use the least possible privilege.	Every program and user should operate using the least set of privileges needed to complete the job.
Least common mechanism.				Use of mechanisms common among different users should be minimized.
Work factor.				A security mechanism for which the cost of bypassing is greater than the resources of an attacker should be used.
Compromise recording.				Sometimes it might be feasible to implement a security mechanism that records a compromise. This information can be used to render the stolen data useless.
	Secure the weakest link.	*Remember that security features != secure features.*	Secure the weakest link not the easiest and most obvious ones.	Secure the easiest to exploit vulnerability (not easiest to fix). ***Implementation of security features might have vulnerabilities of their own.***
	Compartmentalize.		Make components with differing privileges.	Break the software into components and isolate them as much as possible to contain the effects of security failure.
	Promote privacy.		Do not reveal more than necessary.	Inform on a need to know basis.
	Remember that hiding secrets is hard.		Be careful when storing secrets.	Secret data should be carefully stored in the software.
	Be reluctant to trust.	Assume external systems are insecure.	Validate all data from lower-privileged sources.	Data from all sources, internal or external should be validated.
	Use your community resources.			Use available security resources.
		Do not mix code and data.	Separate code and data.	Code and data might need different levels of security (*e.g.*, if code is compromised then all data might be at risk).
		Fix security issues correctly.		No more security issues should arise due to fixing the existing ones.
		Learn from mistakes.		Learn from experience (own & others).
		Plan on failure.		What should be done in case of failure?
		Employ secure defaults.	Employ secure defaults.	Default software configuration should be secure to the maximum extent.
		Remember that backward compatibility will always give you grief.	Beware of backward compatibility.	Backward compatibility in a piece of software might lead to difficulties in making it secure.
			Assess your threats.	Threat assessment can help in avoidance.
			Beware of components with conflicting security assumptions.	Especially true for components that are reused. Components being reused might have been developed with different security assumptions.
			Recognize and answer attacks.	Software should be aware of intrusions.
			Use only publicly scrutinized cryptography.	Encryption algorithms and cryptographic protocols are not easy to devise. Established ones should be used.
			Use a truly random source to create secrets.	Using true random number generators guarantees added security.
		Establish a security process		Follow a secure software development life cycle.
		Define the product's security goals		Determine the threats to the product and derive the security objective.
		Consider security as a product feature		Consider security features as an integral part of the software and incorporate them from the beginning of development life cycle.

Table 4. Comparison of Secure Design Guidelines based on Peine's Work [64]

5. Summary and Future Research Directions

Software security has become an important concern in today's software dependent society. Secure software are achieved by employing various secure software development methods. This paper analyzes the current SSDLC processes, software security requirements methods (security specification languages and security requirements engineering processes), and secure design methods (secure design languages and guidelines).

The major contributions of this paper are as follows. First, this paper presents a comparison of various SSDLC processes and SSD methods for requirements and design. Such a comparison can be helpful for software developers in selecting a SSDLC process, security specification language, security requirements engineering process, and secure design language according to their needs. The properties identified to compare various software security requirements specification and design languages can be used to choose a particular language depending on the application requirements. Second, the identified properties can guide software developers in designing a mechanism for translating one language into another. Translating one specification language to another language is useful as the tools developed for the target language can also be used to for the source language (after translation). Finally, this work also presents a summary of the state-of-the-art of secure design guidelines. The following paragraphs summarize our analysis on various SSDLC processes, SSD methods for requirements, and SSD methods for design along with the related open issues.

This paper first presents a survey of various SSDLC processes and proposes seven desirable characteristics of a SSDLC process. These characteristics are then used to compare and contrast the existing SSDLC processes. The analysis of SSDLC processes shows that many of them do not incorporate SSD methods to address security concerns throughout the development cycles (only in some phases). Among the processes, MS SDL and CLASP cover more aspects of secure software development.

We also analyzed many security specification languages and proposed properties that should be present in any effective security specification language. These properties were used to analyze security specification languages. Our findings indicate that UMLsec has most of the identified desirable properties of a security specification language.

As mentioned before, a security requirements engineering process specifies the activities that need to be followed to derive security requirements. We have identified 23 activities that are essential to engineer complete and detailed security requirements. We use these 23 activities as a basis to compare five different requirements engineering processes. Our analysis shows that SQUARE incorporates more of these activities than other processes. However, none of the existing processes includes all the activities. It is true that many of these activities are sometimes implicitly performed, and their results are incorporated in the process. However, as with other SSD methods, clear guidance should be provided to the developers, most of whom are not well versed in security issues. As a future work, we recommend to design an improved security requirements engineering process based on the 23 proposed activities. This would ensure that all the relevant information has been considered and precise security requirements have been defined (both high and low level).

We have observed that security requirements specification languages have also been used for secure software design. However, there is a need to further develop security requirements and secure design specification languages. We compared many secure design guidelines reported in the literature. It is expected that these guidelines will be enhanced, and they will grow in number as more security issues are encountered in practice.

The analysis performed in this paper only covers the requirements and design methods for secure software development. It does not include secure design patterns, secure coding standards, software security assurance methods such as security testing (*e.g.*, penetration testing), static analysis, model checking, and code reviews for security. It also does not discuss security standards and criteria, organizational processes for developing secure software, and the specification of other related issues such as trust, reliability, and privacy.

References

[1] M. Bishop, *Introduction to Computer Security*, Addison Wesley, 2004.

[2] G. McGraw, *Software Security: Building Security In*, Addison Wesley, 2006.

[3] K.R. Jayaram and A.P. Mathur, "Software Engineering for Secure Software – State of the Art: A Survey," CERIAS tech. report 2005-67, Department of Computer Science, Purdue University, Indiana, USA. 2005.

[4] D.P. Gilliam, T.L. Wolfe, J.S. Sherif, and M. Bishop, "Software Security Checklist for the Software Life Cycle," In *Proc. of the 12th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'03)*, Linz, Austria, IEEE CS Press, 2003, pp. 243-248.

[5] G. McGraw, "Testing for Security During Development: Why we should Scrap Penetrate-and-Patch," *IEEE Aerospace and Electronic Systems*, IEEE CS Press, 1998, vol. 13, no. 4, pp. 13-15.

[6] J. Juerjens, *Secure Systems Development with UML*, Springer, 2005.

[7] B.W. Boehm and P.N. Papaccio, "Understanding and Controlling Software Costs," *IEEE Transactions on Software Engineering*, IEEE CS Press, 1988, vol. 14, no. 10, pp. 1462-1477.

[8] I.V. Krsul, "Software Vulnerability Analysis," doctoral dissertation, Department of Computer Sciences, Purdue University, Indiana, USA, 1998.

[9] A. Jaquith, *Security Metrics*, Addison Wesley, 2007.

[10] D. Verdon and G. McGraw, "Risk Analysis in Software Design," *IEEE Security and Privacy*, IEEE CS Press, 2004, vol. 2, no. 4, pp. 79-84.

[11] R. Kissel, "Glossary of Key Information Security Terms," National Institute of Standards and Technology, NISTIR-7298, 2006.

[12] I. Flechais, M.A. Sasse, and S.M.V. Hales, "Bringing Security Home: A Process for Developing Secure and Usable Systems," In *Proc. of the New Security Paradigms Workshop (NSPW'07)*, Ascona, Switzerland, ACM Press, 2003, pp. 49-57.

[13] C.E. Landwehr, A.R. Bull, J.P. McDermott, and W.S. Choi, "Taxonomy of Computer Program Security Flaws," *ACM Computing Surveys*, ACM, 1994, vol. 16, no. 3, pp. 211-254.

[14] J. Musa, *Software Reliability Engineering*, McGraw Hill, 1998.

[15] M.U.A. Khan and M. Zulkernine, "Quantifying Security in Secure Software Development Phases," In *Proc. of the 2nd IEEE International Workshop on Secure Software Engineering (IWSSE'08)*, Turku, Finland, 2008, IEEE CS Press, pp. 955-960, 2008

[16] A. Apvrille and M. Pourzandi, "Secure Software Development by Example," *IEEE Security and Privacy*, IEEE CS Press, 2005, vol. 3, no. 4, pp. 10-17.

[17] M. Hussein, M. Raihan, & M. Zulkernine, "Software Specification and Attack Languages," In *Advances in Enterprise Information Technology Security*, D. Khadraoui and F. Herrmann, Eds. IGI Global, 2007.

[18] M. Zulkernine, M. Graves, & M.U.A. Khan, "Integrating Software Specification into Intrusion Detection," *International Journal on Information Security*, Springer, 2007, vol. 6, pp. 345-357.

[19] A. Myagmar, A.J. Lee, and W. Yurcik, "Threat Modeling as a Basis for Security Requirements," In *Proc. of the Symposium on Requirements Engineering for Information Security*, Paris, France, IEEE CS Press, 2005, pp. 94-102.

[20] S. Lipner and M. Howard, "The Trustworthy Computing Security Development Lifecycle," 2005. http://msdn.microsoft.com/en-us/library/ms99519.aspx. Last Accessed March 2009.

[21] S. Lipner, "The Trustworthy Computing Security Development Lifecycle," In *Proc. of the 20th Annual Computer Security Applications Conference (ACSAC '04)*, CA, USA, 2004, IEEE CS Press, pp. 2-13.

[22] I. Flechais, C. Mascolo, and M.A. Sasse, "Integrating Security and Usability into the Requirements and Design Process," *International Journal of Electronic Security and Digital Forensics*, Inderscience Publishers, Geneva, Switzerland, 2007, vol. 1, no. 1, pp. 12-26.

[23] A.S. Sodiya, S.A. Onashoga, and O.B. Ajayi, "Towards Building Secure Software Systems," *Issues in Informing Science and Information Technology*, Informing Science Institute, California, USA, 2006, vol. 3, pp. 635-646.

[24] L. Futcher and R.v. Solms, "SecSDM: A Model for Integrating Security into the Software Development Life Cycle," In *IFIP International Federation for Information Processing, Volume 237, Proc. of the 5th World Conference on Information Security Education*, Springer, 2007, pp. 41-48.

[25] D. Gilliam, J. Powell, E. Haugh, and M. Bishop, "Addressing Software Security Risk and Mitigations in the Life Cycle," In *Proc. of the 28th Annual NASA Goddard Software Engineering Workshop (SEW'03)*, Greenbelt, Maryland, USA, 2003, pp. 201-206.

[26] M.A. Hadawi, "Vulnerability Prevention in Software Development Process," In *Proc. of the 10th International Conference on Computer & Information Technology (ICCIT'07)*, Dhaka, Bangladesh, 2007,

[27] OWASP CLASP Project, http://www.owasp.org/index.php/Category:OWASP_CLASP_Project. Last Accessed March 2009.

[28] M. Essafi, L. Labed, and H.B. Ghezala, "S2D-ProM: A Strategy Oriented Process Model for Secure Software Development," In *Proc. of the 2nd International Conference on Software Engineering Advances (ICSEA'07)*, Cap Esterel, French Riviera, France, 2007, p. 24.

[29] N. Davis, "Secure Software Development Life Cycle Processes: A Technology Scouting Report", technical note CMU/SEI-2005-TN-024, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsalyania, USA, 2005.
[30] DOVES: Database of Vulnerabilities, Exploits, and Signatures, http://seclab.cs.ucdavis.edu/projects/DOVES/. Last Accessed March 2009.
[31] J. Gregoire, K. Buyens, B. De Win, R. Scandariato, and W. Joosen, "On the Secure Software Development Process: CLASP and SDL Compared," In *Proc. of the 3rd International Workshop on Software Engineering for Secure Systems (SESS'07)*, Minneapolis, Minnesota, USA, IEEE CS Press, 2007, pp. 1-1.
[32] Common Vulnerabilities and Exposures, http://cve.mitre.org/. Last Accessed March 2009.
[33] Common Weakness Enumeration, http://cwe.mitre.org/. Last Accessed March 2009.
[34] M. Howard, D. LeBlanc, and J. Viega, *19 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*, McGraw Hill, 2005.
[35] OWASP, http://www.owasp.org/index.php/Main_Page. Last Accessed March 2009.
[36] G. McGraw and B. Potter, "Software Security Testing," *IEEE Security and Privacy*, IEEE CS Press, 2004, vol. 2, no. 5, pp. 81-85.
[37] S.d. Vries, "Software Testing for Security," *Network Security*, ScienceDirect, 2007, vol. 3, pp. 11-15.
[38] C.B. Haley, J.D. Moffett, R. Laney, and B. Nuseibeh, "A Framework for Security Requirements Engineering," In *Proc. of the International Workshop on Software Engineering for Secure Software (SESS'06)*, Shanghai, China, ACM Press, 2006, pp. 35-41.
[39] M. Raihan and M. Zulkernine, "AsmLSec: An Extension of Abstract State Machine Language for Attack Scenario Specification," In *Proc. of the 2nd International Conference on Availability, Reliability and Security (ARES'07)*, Vienna, Austria, IEEE CS Press, 2007, pp. 775-782.
[40] Z.J. Zhu and M. Zulkernine, "A Model-Based Aspect-oriented Framework for Building Intrusion-aware Software Systems," *Information and Software Technology Journal*, Elsevier Science, to appear.
[41] M. Graves and M. Zulkernine, "Bridging the Gap: Software Specification Meets Intrusion Detector," In *Proc. of the 4th Annual Conference on Privacy, Security and Trust (PST'06)*, Ontario, Canada, pp. 265-274.
[42] G. Sindre, A.L. Opdahl, "Eliciting Security Requirements by Misuse Cases," In *Proc. of the 37th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'00)*, Sydney, Australia, IEEE CS Press, 2000, pp. 120-131.
[43] J. McDermott and C. Fox, "Using Abuse Case Models for Security Requirements Analysis," In *Proc. of the 15th Computer Security Applications Conference (ACSAC'99)*, USA, IEEE CS Press, 1999, pp. 55-64.
[44] M. Hussein and M. Zulkernine, "UMLintr: a UML profile for specifying intrusions," In *Proceedings of the 13th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, Potsdam, Germany, IEEE CS Press, 2006, pp. 279–286.
[45] T. Lodderstedt, D.A. Basin, and J. Doser, "SecureUML: A UML-Based Modeling Language for Model Driven Security," In *Proc. of the 5th International Conference on the Unified Modeling Language (UML'02)*, Dresden, Germany, Springer, 2002, LNCS 2460/2002, pp. 426-441.
[46] H. Mouratidis, P. Giorgini, and G. Manson, "When Security Meets Software Engineering: A Case of Modeling Secure Information Systems," *Journal of Information Systems*, Elsevier Science, 2005, vol. 30, no. 8, pp. 609-629.
[47] S.T. Eckmann, G. Vigna, and R.A. Kemmerer, "STATL: An Attack Language for State-Based Intrusion Detection," *Journal of Computer Security*, IOS Press, Amsterdam, 2002, vol. 10, no. 1/2, pp. 71-104.
[48] Snort, www.snort.org. Last Accessed March 2009.
[49] UMLsec Tools, mcs.open.ac.uk/jj2924/umlsectool/index.html. Last Accessed March 2009.
[50] P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini, "TROPOS: An Agent- Oriented Software Development Methodology," *Journal of Autonomous Agents and Multi-Agent Systems*, 2004, vol. 8, no 3, pp. 203-236.
[51] D.G. Firesmith, "Security Use Cases," *Journal of Object Technology*, 2003, vol. 2, no. 3, pp. 53-64.
[52] M. Hussein and M. Zulkernine, "Intrusion Detection Aware Component-Based System: A Specification-Based Framework," *Journal of System and Software*, Elsevier Science, 2007, vol. 80, no 5, pp. 700-710.
[53] J. Juerjens, "Model-based Security Testing using UMLsec," In *Proc. of the 4th Workshop on Model-Based Testing (MBT'08)*, Budapest, Hungry, 2008.
[54] Model-Driven Security with SecureUML, http://www.infsec.ethz.ch/people/doserj/mds. Last Accessed March 2009.
[55] SecureUML Tool, http://www.foundstone.com/us/resources/proddesc/secureumltemplate.htm. Last Accessed March 2009.
[56] Si, http://sesa.dit.unitn.it/sistar_tool/home.php?7. Last Accessed March 2009.

[57] N.R. Mead, E. Hough, and T. Stehney, *Security Quality Requirements Engineering (SQUARE) Methodology*, tech. report CMU/SEI-2005-TR-009, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 2005.

[58] J Viega, "Building Security Requirements with CLASP," In *Proc. of the 2005 International Workshop on Software Engineering for Secure Systems (SESS)*, St. Louis, Missouri, USA, 2005, pp. 1-7.

[59] D. Mellado, E. Fernandez-Medina, and M. Piattni, "A Common Criteria-Based Security Requirements Engineering Process for the Development of Secure Information Systems," *Computer Standards and Interfaces*, Elsevier Science, 2007, vol. 29, pp. 244-253.

[60] V. Saini, Q. Duan, and V. Paruchuri, "Threat Modeling using Attack Trees," *Journal of Computing Sciences in Colleges*, Consortium for Computing Sciences in Colleges, USA, 2008, vol. 23, no. 4.

[61] B. Schneier, "Attack Trees," *Dr. Dobb's Journal of Software Tools*, CMP Media, Dec 1999, pp. 21-29.

[62] R.J. Ellison, "Attack Trees," 2006, https://buildsecurityin.us-cert.gov/daisy/bsi/articles/ best-practices/requirements/ 236-BSI.html. Last Accessed March 2009.

[63] A.v. Lamsweerde, "Elaborating Security Requirements by Construction of Intentional Anti-Models," In *Proc. of the 26th International Conference on Software Engineering (ICSE'04)*, Edinburgh, UK, IEEE CS Press, 2004, pp. 148-157.

[64] H. Peine, "Rules of Thumb for Developing Secure Software: Analyzing and Consolidating Two Proposed Sets of Rules," In *Proc. of the 3rd International Conference on Availability, Reliability and Security (ARES'08)*, Barcelona, Spain, IEEE CS Press, 2008, pp. 1204-1209.

[65] J.H. Saltzer and M.D. Schroeder, "The Protection of Information in Computer Systems," In *Proc. of the IEEE*, 1975, vol 63, no. 9, pp. 1278-1308.

[66] J. Viega and G. McGraw, *Building Secure Software*, Addison Wesley, 2002.

[67] M. Howard and D. LeBlanc, *Writing Secure Code 2nd Edition*, Microsoft Press, 2003.

[68] B. Bauer, J. Muller, and J. Odell, "Agent UML: A Formalism for Specifying Multi-agent Interaction," In *Agent Oriented Software Engineering*, P. Ciancarini and M. Wooldridge, Eds. Springer, 2001.