S2D-ProM: A Strategy Oriented Process Model for Secure Software Development

Mehrez Essafi, Lamia Labe, and Henda Ben Ghezala
National School for Computer Studies, RIADI Laboratory
University of Manouba
Tunis - Tunisia
mehrez.essafi@riadi.rnu.tn, Lamia.Labe@isg.rnu.tn, henda.benghezala@ensi.rnu.tn

Abstract—Building Secure Software is about taking security into account during all phases of software development. This practice is missing in, widely used, traditional developments due to domain immaturity, newness of the field and process complexity. Software development includes two views, a product view and a process view. Product view defines what the product is, whereas process view describes how the product is developed. Here we are concerned with the process view. Modelling the process allows simulate and analyze a software development process, which can help developers better understand, manage and optimize the software development process. In this paper we present our approach S2D-ProM, for Secure Software Development Process Model, which is a strategy oriented process model. This latter, capture steps and strategies that are required for the development of secure software and provide a two level guidance. The first level guidance is strategic helping developers choosing one among several strategies. The second level guidance is tactical helping developers achieving their selection for producing secure software. The proposed process model is easily extensible and allows building customized processes adapted to context, developer’s finalities and product state. This flexibility allows the environment evolving through time to support new securing strategies.

Keywords-component—Secure Software Development; Strategic Process Model; Strategic guidance; Tactical Guidance.

I. INTRODUCTION

The term “secure software” describes the state of software, where no unauthorized access, no manipulation, and no attacks on the software itself are successful. Such security is measured according to the existing number of security vulnerabilities. Software without any vulnerability is defined as maximum secure. Software with, at least, one vulnerability is defined as insecure [32].

Current software security, often, fail because their development is generally based on ad-hoc foundations without the support of appropriate methodologies able to manage the high complexity of the securing process. In fact, most secure software weakness result from the use of traditional development processes –not focusing on security concerns– and the lack of necessary knowledge and guidance on what is a secure software and how to produce it [31].

As a long-term, complex and multifaceted problem, improving software security and safeguarding the information technology requires multiple solutions and the application of resources throughout the whole life cycle [2][20][33].

This paper describes our approach S2D-ProM (Secure Software Development Process Model). The main feature of the proposed approach is that it is a strategy oriented process model, with respect to the MAP formalism, that allows to provide two level guidance: (a) a strategic guidance helping the developer to choose one among compilations of existing techniques, methods, standards [4] and best practices [3][29] which are useful for producing secure software and (b) a tactical guidance on how to achieve his selection.

II. SECURITY IN SOFTWARE DEVELOPMENT

Disregarding preferences of a specific software development philosophy, all software projects use the same base to solve development problems. It doesn’t matter if the project uses the Waterfall-model, the Spiral model, incremental-iterative approaches, or agile methods. Reduced to a minimum, the basics of every software project are basics from software engineering: 1) requirements, 2) design, 3) implementation, and 4) testing.

Software security cannot be addressed at only one development life-cycle sub-segment [21]. In fact, a secure design without secure implementation, even if it was formally specified and verified, is nothing. Security testing might help locate vulnerabilities, but secure implementation before it reduces further corrective overhead.

Security required in a software system must be addressed at all stages of the development process, independent of used development philosophy. To have a secure infrastructure, the supporting software must contain few, if any, vulnerabilities. Vulnerability could be: (a) a defect that is an implementation and/or design error [25]. It may lie dormant in software for several years and then surface in a fielded system with major consequences, (b) a bug, which is an implementation-level software error. Bugs refer to low-level implementation errors that could be remedied by limited code analysis of the external environment, or, (c) a flaw that is a subtle defect at a deeper level [1].
Improving software security requires multiple solutions and security at the end of the development process which may imply highest costs with relative effectiveness. Therefore, a lot of progress has to be made in studying and analysing software artefacts before their deployment. Improving software security requires multiple solutions and the application of resources throughout the whole development life cycle.

IV. THE PROBLEM

A software system that is able to withstand attack provides some confidence that it is secure, but there are no generally accepted ways to prove that it is absolutely secure. With current methods, we can only prove that it is not secure. Beside, we still unable to completely verify that deployed, enhanced, repaired, or remediate secure software will preserve its security properties unaltered.

Face to the domain immaturity, newness of the field, absence of a support environment and the process complexity, software engineers and developers who lack security-engineering expertise, need sufficient knowledge and efficient guidance on how to find out the convenient and appropriate way for reaching the required software security level. This knowledge and guidance must be modelled and ideally supported by an environment to prosper better results.

V. RELATED WORKS

Several efforts are undertaken, to improve software security. However, developers still miss or misuse acquired knowledge. This is mainly due to domain immaturity, newness of the field, process complexity and absence of environments supporting such development. In fact, in the software engineering domain, no processes or practices have been shown to consistently produce secure software.

From these efforts resulted many analysis techniques –such as threat modelling, risk analysis and code analysing tools, many extensions to already existing formalisms –such as UMLSec and SecureUML, and many security related standards such as the Common Criteria. Indeed, substantial reduction has been demonstrated in overall software design and implementation defects, as well as in security vulnerabilities.

In software engineering domain the main interested is in how to enhance existing lifecycle phases, artifacts and techniques used in each phase, or perhaps introduce new techniques, to support security. The holy grail of this field is software that is secure by construction. We believe that security will be improved only by focusing on its development process since the early phases.

Many researchers have focused on considering so-called best practices in the software lifecycle, as those proposed by CLASP (Comprehensive, Lightweight Application Security Process). As the name implies, a security development lifecycle is a Software Development Lifecycle (SDL) where a special emphasis is placed on software security in each phase. In this perspective, two SDLs have been proposed in the state of the art which integrates software security into the lifecycle, one is by Microsoft as part of its Trustworthy Computing Initiative and the other by McGraw. We refer to these efforts as Microsoft’s SDL and McGraw’s SDL. Obviously, both SDLs are too recent, are activity-oriented approaches and have a lot in common. They enumerate security best practices applied to various software artefacts such as threat analysis, security review, security push/audit, etc. The two SDLs suggest cycle, by iteration, on the proposed practices more than once as the software evolves. The high level of abstraction and the lack of guidance and details in these enhanced lifecycle models limit their usefulness in actually supporting and helping engineers in developing secure software.

VI. DISCUSSION

Secure software development processes need to be accessible to most developer’s profiles ranging from beginners to experts. Consequently, such processes need to be flexible and extensible to suit the developer affinities instead of being enforced with a collection of predefined process models. Such flexibility increases with the multiplicity of approaches proposed in the state of the art to accomplish each step in the development process. In fact, there is never one way to proceed, but several ones.

Required activities to be added to secure software development processes include, especially, those dealing with security risk management such as: (a) artefact analysis which examines the possible risk it could occur, (b) risk ranking to prioritise and determine the risks that should be avoided, (c) risk mitigation which should be conducted in a coherent and a cost-effective manner, and (d) validate that software artefacts and processes no longer bear unacceptable risk. Such development processes imply many resources and needs knowledge that most software engineers miss or misuse, that’s why processes should provide guidance support for better usefulness.

To model the intended process, five kinds of process models can be used: activity-oriented, products oriented, decision oriented, context oriented and strategy-oriented process models. Our process model should help developers selecting the appropriate way to produce software with required security level. Thus the decision-oriented process, and activity or product-oriented models are not appropriate. In addition, the process needs to capture knowledge about how to progress in order to allow developer construct a personalized process. The latter depends on the application domain, developer’s experience and previous choices. Many ways are then possible to achieve the desired goal, and strategy-oriented process seems to be the best candidate for this modeling. Hence, we assume security-engineering processes to be intention-oriented. At any moment, the engineer has an intention, a goal in mind that he wants to fulfill.
VII. ADOPTED MODELING FORMALISM OVERVIEW

To model our development process, S2D-ProM, we choose the MAP formalism which is a strategy oriented process model [14][20][7]. The MAP could be seen as a labelled directed graph with intentions (which is a goal expressed as a natural language statement comprising a verb–that can be achieved by performing a process) as nodes and strategies (defining the way to achieve the target intention) as edges between intentions. Therefore, the MAP identifies the set of sections each of which is a triplet <Ii, Ij, Sij>, where Ii is source intention, Ij is a target intention and Sij is a strategy from the source intention to the target intention. Start and Stop intentions represent the intentions to start and to stop navigating in the MAP. The MAP can evolve through time to support new sections in order to satisfy new requirements.

The MAP can also be seen as a set of process descriptions. Using dynamic selections (reflecting the flexibility) from these descriptions, the best particular prescription adapted to the current situation of the software product and the developer affinities is chosen. In that sense, the MAP is a multi-model that allows the developer determine, through guidelines, the best way that fits its requirements. Guidelines result from previous acquired experiences and provided solutions. Three kinds of guidelines are attached to the map: (a) “Intention Achievement Guideline” (IAG), which helps to fulfill the intention selected by the engineer—it could be non-formal, tactic or strategic (b) “Intention Selection Guideline” (ISG) and (c) “Strategy Selection Guideline” (SSG). The two last ones help the developer progressing in the MAP and selecting the right section and is always tactic.

VIII. SECURE SOFTWARE DEVELOPMENT TAXONOMY

Software development includes two views: product view and process view. Product view defines what a product is, whereas process view describes how a product is developed. It can also says, product view explains knowledge “knowing-what” and “who knows the product”; and process view illustrates “know-how” and “know-why” knowledge. Here we are concerned with the process view. Modelling the process will allow to simulate and analyze secure software development process, which can help developers better understand, manage and optimize the secure software development process.

Secure software development could be classified into four categories: reactive, corrective, proactive and hybrid approaches. This classification results from the views previously described.

Reactive approaches are adopted in maintenance phase. They deal with developing patches after attacks have been made on the product, so we acquired knowledge about a product vulnerability, which must be addressed via a maintenance activity.

Corrective approaches, also known as “Start in the middle approaches”. They are used during the development process by iterative transition between implementation and test phases. They are based on captured knowledge about the product state and an existing knowledge on the required process.

Proactive approaches also qualified as preventive or top down approaches. They deal with security concern since the requirements phase will drive the next development steps. This approach tries to get a complete knowledge about the product before beginning the design and the implementation process.

Hybrid approaches are a mixture of previously cited ones. These approaches seem to best fit the development of secure software since there is no processes or practices that have been shown to consistently produce this kind of software [2].

In Fig 1., we illustrate a MAP modelling the software life cycle. This model gives an abstraction view on the different ways on how an application engineer can proceed to develop secure software through the different approaches.

Figure 1. Secure software development MAP

From this MAP, an application engineer could instantiate at least four different software securing processes according to the previously cited approaches: (1) he develops the application using a traditional approach—which does not necessarily deal with security—and deploys the developed software. Then, he returns to development intention with the reactive approach where he can use a securing strategy (2) he develops an application with a traditional approach, then he secures it using a corrective approach, (3) he combines 1 and 2 or 2 and 1 to get an hybrid approach. With these three approaches, security is treated at the end of the development process (4) he develops an application with a proactive approach that can include the Microsoft’s and McGraw’s SDL.

IX. PROPOSED SOLUTION

Our solution aims at providing a strategy oriented process model for addressing software security issues early, during the development process that provides guidance to developers, especially those who lack security experience. The proposed approach does not impose activities to do, but suggest what can be done (via selection guidelines) and how it can be done (via tactical guidelines) in order to reach the desired security level.

The proposed process model, illustrated by Fig 2, can be viewed as an alternative strategy to the proactive approaches that application engineers can choose for developing secure software. In fact, it is a possible intention achievement guideline (IAG) for “Develop an application” intention choosing a proactive approach. On Fig 2, we find intentions that represent required steps and strategies to develop secure software. Strategies represent possible ways for releasing a step in the process. First level intentions added here to the traditional developing steps (specify, design, develop and test)
in order to prevent security problems in a proactive mode are: (a) “Analyse security risk” intention, which seeks at identifying and understanding possible security flaws. This intention could be achieved from “specify requirement”, “Design” and “Implement” for studying associated artefacts using adequate strategies and (b) “Mitigate security risk” intention aims at bringing potential resolutions to the identified risks.

Intention achievement guidelines define the way an intention could be released and could be, strategic, tactical, or atomic. For example, “Test Strategy” strategy, in <Implement, Analyse security risk, Test Strategy> section, could be refined into: (a) “functional test”: Thus a <Implement, Analyse security risk, functional test> section required as the lowest security assurance level [1], and (b) “structural test”: Thus a <Implement, Analyse security risk, structural test> section required as the second level security assurance, etc. (see Fig 4).

Using the same formalism to describe the method let developers using a unique logic, which may enhance the process model exploitability and understanding.

X. S2D-ProM Approach Experimentation

The approach experimentation consists in instantiating the proposed process model. The instantiation produces many artefacts, particularly security related ones such as, abuse cases, threat trees, attack trees, attack scenario, security test plans, security charts, rules, etc.

For instance, the application engineer can define a security policy artefact using a global security approach and then specify the application requirement with respect to the defined policy. Then he can analyse security risks depending on the application domain using an analysis method such as FHM. This technique will allow identifying possible threats and vulnerabilities to consider later in the other phases. Next, the application engineer can model attacks using an attack modelling technique, which is another instance of the “Using an analysis method” to achieve the intention “analyse security risk”.

Figure 4. Testing section strategy selection guideline

As example of a strategic guideline, Fig 5 illustrates the FHM (Flow Hypothesis Method) [30], modelled using the MAP formalism. This method represents an instance of the strategy “Using an Analysis Method” to achieve the intention “analyse security risk”.

Figure 5. FHM modeled as a strategy guideline

The strategy selection guideline (SSG 2.1): Associated to the test strategy section is illustrated in Fig 4. This guideline will help the developer choose on the basis of the related arguments between the two candidate strategies (1) and (2).
As a rule, that could be defined in a global security policy is that “each input is considered as malicious until it is validated to be accepted, rejected or corrected”. This rule implies that we must attach validation mechanisms to the container to control inputs content and prevent inputs related risks.

Security risk analysis phase helps understanding how an attacker could operate on the content and/ or the container in order to violate security properties. This understanding allows identifying the appropriate security risk mitigation solution and thus the adequate validation mechanisms.

Attack trees modeling is one of the techniques suggested to be used for analyzing security risks. Fig 6 and Fig 7 illustrate the two forms of threats respectively associated to container and content elements.

![Disturbing input container attack tree](Image)

Figure 6. Disturbing input container attack tree

Fig 6 explains how a pirate could disturb a container which could be an input file or a memory field. For a file, he could destroy it and thus it will not be available for future use, or, he could modify one of the file properties such as its name, its location, its access control flags, etc. which may compromise the file availability and/ or the software system integrity. For the memory field, the pirate could, for example, overflow the field limits to access other memory locations and may violate their confidentiality or integrity properties.

Respectively, Fig 7 refines how a pirate could disturb content such as transforming or altering it or by introducing its canonical form.

![Disturbing input content attack tree](Image)

Figure 7. Disturbing input content attack tree

An example of attacks, which is very common, that use malicious inputs to release an exploit is buffer overflow. Fig 8, illustrates an attack tree which is the result of a security risk analysis step. This tree allows understanding how to realize the attack and the required conditions for its achievement.

![Buffer overflow attack tree artefact](Image)

Figure 8. Buffer overflow attack tree artefact

As shown on Fig 8, to exploit a buffer overflow, many attack scenarios are possible with logic operators between exploiting preconditions (here only AND operators are illustrated). So, to realize such kind of attacks, one must (have the source code OR Notice an abnormal behavior of the program) AND (use a Trojan Hors to form a malicious input) AND (overflow a buffer using a long input OR violate the buffer boundaries limits to modify a return address value and so on). Some of these attack scenarios could be used during security testing to verify if the defined solutions are efficient to prevent this kind of exploit. Solutions are defined in contrast with the preconditions to break them and prevent their satisfaction.

With ANDs between preconditions, partial solutions are thus possible. As sufficient solution that could be identified in the “Mitigate security risk” step using a defining formal security rules strategy as solving approach, the application engineer could try verifying the following constraints for each input buffer (that could be a command line argument, a parameter, a file, a register, a field, a dialog box, etc.):

1. \( \text{BufferCapacity} \leq \text{DestinationBufferSize} \)
2. \( \text{BufferCapacity} \geq \text{SourceBufferSize} \)
3. \( \text{BufferIndex} \geq \text{BufferLowerBound} \)
4. \( \text{BufferIndex} \leq \text{BufferUpperBound} \)
5. \( \text{BufferUpperBound} \leq \text{BufferSize} \)

If included as prevention treatments and verified in code implementation, these constraints would relax the precondition “Force the program jumping to a target address” and then the attack will miss an important part for its release success. As we can notice, attack trees artifacts help understanding how to perform attacks. This provides developers a thinking support on how and where to act to solve threats and test them later. These attack trees support the use of logic operators between nodes and could be reused and/ or augmented for other exploit forms and thus evolve through time. Other strategies instantiation such as “anti-requirement reasoning” will produce other useful artifacts (for example abuse cases). Abuse cases, allow modeling the non desired interaction (anti-requirement) between the application and its actors and
enhance the security risks understanding from another viewpoint.

Produced security artifacts could be stored for reuse in similar product development which thwart developers from wasting their time in reproducing them or taking ad hoc considerations.

XI. CONCLUSION

Security is now critical for software systems and needs to be part of an end-to-end development either than being considered as an afterthought. Security policies, standards, guidelines and best practices should be inherent in secure software development processes and should be accessible to developers who lack knowledge about how to address security issues. The high level of abstraction in existing process models that try to include additional steps to address security problems limits their usefulness in actually supporting and helping in producing secure software.

In this paper we proposed a strategy-oriented process model, denoted S2D-ProM, which is able to cover existing securing approaches and provides: (a) a wide range of granularity with strategic and tactical guidance to thwart model, denoted S2D-ProM, which is able to cover existing informal techniques stated for addressing software security developers –especially novice ones– from taking hypothesis and ad hoc decisions when studying, defining and implementing security requirements, (b) flexibility that allows instantiating a personalised process adapted to the developer implementing security requirements, (c) support for non-formal, semi formal and ad hoc decisions when studying, defining and implementing security requirements, (d) flexibility that allows instantiating a personalised process adapted to the developer implementing security requirements, (e) support for non-formal, semi formal and ad hoc decisions when studying, defining and implementing security requirements, (f) flexibility that allows instantiating a personalised process adapted to the developer implementing security requirements, (g) support for non-formal, semi formal and ad hoc decisions when studying, defining and implementing security requirements, (h) flexibility that allows instantiating a personalised process adapted to the developer implementing security requirements, (i) support for non-formal, semi formal and ad hoc decisions when studying, defining and implementing security requirements.

An environment denoted “ASASI” (Addressing Software Application Security Issues) is being implemented to provide a support tool for the S2D-ProM approach.

REFERENCES