Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 56 (2015) 453 – 458

Procedia Computer Science

International Workshop on Enterprise Web Application Dependability (EWAD 2015)

Representing security specifications in UML state machine diagrams

Muhammad Umair Khan*

Centennial College, 941 Progress Avenue, Toronto, M1G3T8, Canada

Abstract

Security specifications are controls and constraints on the behavior of the software and can be used to develop more secure software from the beginning. Many specification languages have been proposed to represent security specifications. However, all these specification languages are at a higher level of abstraction and can only be used to represent overall business-level design decisions. Such specifications provide guidance to the developers but do not lay out the details of the dynamic behavior that has to be implemented during the coding phase. In this paper, we propose to use UML state machine diagrams to represent detailed dynamic behavior of design-level security specifications. We argue that these behaviors when used by the developer for implementation will enable them to avoid crucial security vulnerabilities.

Peer-review under responsibility of the Conference Program Chairs

Keywords: Design-level security specifications; UML; state machine diagrams.

1. Introduction

Software can be made more secure by building them in a secure manner. When a vulnerability is reported in a software, new security requirements, design decisions, and implementation guidelines are devised and proposed to avoid and mitigate that vulnerability in the future. These requirements, design decisions, and implementation guidelines are controls and constraints on the software and are made widely available through databases such as

* Corresponding author. Tel.: +1-647-351-7635.
E-mail address: umair313@yahoo.com

Peer-review under responsibility of the Conference Program Chairs
doi:10.1016/j.procs.2015.07.235

Common Weakness Enumeration[1] (CWE) and Open Web Application Security Project[2] (OWASP). Most of the time, in these databases and also other places, a textual explanation of the vulnerability and its mitigations is provided. In some cases, security specification languages such as UMLsec[3], Misuse Cases[4], Abuse Cases[5], SecureUML[6], and SecureTropos[7] have also been used. However, the higher level of abstraction in diagrammatic specification languages such as UMLsec and SecureTropos and the inherent ambiguity of the textual specification languages can be an impeding factor for the developers and programmer to actually use these mitigations correctly. This results in the developers and programmers in devise their own ways in which to implemented higher-level security specifications.

We classify security specifications into two categories: static and dynamic. Static specifications are those that deal with decisions and guidelines which indicate what should be used and what should be avoided. An example of a static security specification is that it is recommended to avoid the "*printf*" function in the C/C++ language to circumvent the possibility of the format string vulnerability. Dynamic security specifications, on the other hand, define how the software should behave in a particular situation. The behavior of an authentication mechanism is an example of dynamic security specification.

In the current vulnerabilities databases (*e.g.*, CWE and OWASP), none of the dynamic security specifications has a detailed design-level behavioral description (textual or diagrammatic) that can be used by the developers and programmers in the implementation phase. The information about what has to be done or avoided is available but the details of how to achieve the goal is left to the developers' imagination and expertise. For example, a security specification may require the authentication mechanism to lock a user after a certain number of incorrect login attempts. However, developers will have to implement this security specification on their own and may make mistakes in checks and balances (*e.g.*, how many incorrect login attempts, when to unlock, and what time interval if any should be used). This level of detail is necessary in the design phase because the implemented software can then be verified for bugs and flaws in the assurance phase. However, if the design is not at all available, then the security assurance phase will not be able to concentrate its efforts. From example, the test cases against the high-level security specification "lock the user after x incorrect login attempts" will not know the value of x if it is not defined in the design document. The value of x will not be available in the requirements or high level-design document. In the absence of the value of x, the developer will be free to choose a value of x. However, the security demands of the system being implemented may be too hard to allow a large value. In which case, the choice of the developer for a bigger value may have a negative impact on the security of the software. Moreover, when the software is being tested, the assurance team will not know how many incorrect login attempts should be allowed based on the software's security profile.

To solve the above mentioned issues, in this paper, we propose that dynamic security specifications should be represented in UML state machine diagrams which are a widely understood and used behavior representation diagram. Other languages such as SecureUML and UMLsec have attempted to describe dynamic behavior however they are at a higher level of abstraction and the possibility for the developers to incorrectly implement intended security specifications remains. This paper is organized as follows. In Section 2, we present a literature review of existing secure specification languages. In Section 3, we describe, through the use of a case study, how UML state machines can be used to effectively represent a number of security specifications. Section 4 concludes this paper by summarizing the work and discussing future work.

2. Related work

UMLsec[3] is a stereotype based extension of UML to develop secure software systems. It uses stereotypes, tags, and constraints to specify security specifications in use case diagrams, class diagrams, state machine diagrams, activity diagrams, sequence diagrams, and deployments diagrams. UMLsec defines 21 stereotypes to represent fair exchange, non-repudiation, role-based access control, secure communication link, confidentiality, integrity, authenticity, freshness of a message, secure information flow among components, and guarded access. Some stereotypes also have associated tags and constraints. In our proposal we do not use tags and stereotypes to represent security specifications. Although, tags and stereotypes do convey the overall security requirements, they lack the "how" component of a security specifications. In our proposal, we use state machine diagrams to represent the intended dynamic behavior of security mechanisms.

SecureUML[6] is an extension of UML which can be used for specifying role-based access control policies. SecureUML stereotypes 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. SecureUML is different from out proposed approach as we use state machine diagrams to represent all those security specifications that can be represented in a state machine. However, in SecureUML, the authors only discuss how role-based access control policies can be represented.

SecureTropos[7] uses the notions of actor (person(s), organization(s), or software), goals, soft goal (a goal whose fulfillment cannot be explicitly determined), task, resource, security constraint, secure goal, secure task, and 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. SecureTropos is different from UML state machine as it used goals and actors. This level if more close to a UML use case diagram. Although beneficial, SecureTropos does not address the issue of providing detailed instructions to the developer on how a security specification should be implemented.

A misuse case[4] is a UML use case to represent undesirable behavior. Misuse cases can be used to elicit more use cases to neutralize the threats. Two special relations called "prevents" and "detects" relate use cases and misuse cases. The authors provide a stepwise process according to which, 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. 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"[8].

Another way to specify undesirable behavior of a piece of software using UML use case diagrams is to develop an abuse case[5] 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. As compared to the approach proposed in this paper, both Abuse and Misuse cases remain in the requirement specification phase and do not contribute to the design of the target software.

3. UML state machine diagrams for security specifications

Security specifications are controls and constraints on the software that are necessary to avoid security vulnerabilities[9-11]. Security specifications restrict the manner in which the software provides its functionality[11]. Security specifications may be elicited during any of the requirements, design, and implementation phases[9] of secure software development. In the following subsections, we first classify security specifications and then describe how security specifications can be represented in UML state machine diagrams.

3.1. Classifying security specifications

Security specifications can be classified based on the type of constraint they impose on the software. Avoiding the use of pointers (dangling pointers vulnerability) and *printf* (format string vulnerability) function are guidelines that restrict the use of a certain part of the C/C++ programming language. Similarly, using security mechanisms such as authentication and role based access control are decisions that are made during the requirements phase. The aforementioned security specifications are examples of *static* security specifications as they identify what should and should not be used but do not specify how something should be used.

In contrast, some security specifications require the software to function in a specific manner. The intended behavior of the authentication and role based access control mechanisms are examples of dynamic security specifications. A new implementation of the *printf* function that circumvents the possibility of the format string

vulnerability while providing the same functionality will also be regarded as a dynamic security specification. Note that such a function is already implemented and available in C/C++ language. The correct allocation and destruction of the allocated memory after the software has finished using it, and opening and closing of ports in a network protocol after the assigned task is finished are further examples of dynamic security specifications.

3.2. Authentication mechanism related security specifications

Authentication mechanisms are widely used to ensure that only legitimate users are allowed access to a particular data or service. Its importance demands that there are no vulnerabilities in the authentication mechanism. Over the years, many vulnerabilities within the implementations of authentication mechanisms have been identified and their corresponding security specifications have been proposed in the literature. Table 1 lists some of the crucial vulnerabilities that have been reported. The third column of the table presents the security specifications that should be used to avoid and remove these vulnerabilities (according to the CWE and OWASP databases). Note that all these security specifications are dynamic in nature.

Table 1. Some of the security specifications related to the authentication mechanism listed in CWE and OWASP.

	Vulnerablity name	Example of the vulnerability	Security specification to remove the vulnerability
1	CWE-303: Incorrect implementation of authentication algorithm (logic error).	Incorrect login attempts are not recorded.	For every incorrect attempt, the component should increment the attempts counter by exactly 1.
2		Access is granted even when the user is locked out.	Access should be denied if username is incorrect or if the user is already locked out.
3	CWE-307: Improper restriction on excessive authentication attempts.	A user can attempt to login without any restriction on the number of incorrect attempts.	Access should be denied if the password is incorrect and the current attempt is first or second. If the current attempt is the third and password is incorrect, then the user should be locked out.
4	CWE 262: Not using password aging.	The password has no age limit.	If the user does not change an old password, then he/she should not be given access.

3.3. UML representation of dynamic security specifications

Each dynamic security specification represents a specific behavior which has to be implemented to fulfil the goal of the corresponding security specification. Extended finite state machines, which are the foundation of UML state machine diagrams, are widely used to represent dynamic behaviors of systems. Hence, we argue that every dynamic security specification can be adequately represented in the form of a UML state machine diagram. All the constraints in the security specifications have to be represented as decision points based on values of appropriate variables in the UML state machine diagram. The final states will represent the eventual goals (positive or negative) of the security specification. We further elaborate this in this section by constructing a UML state machine for the security specifications listed in Table 1.

The first security specification in the table indicates that the algorithm of the authentication mechanism should be correct. A concrete instance of this security specification may be that at the occurrence of an incorrect login attempt, the counter that keeps track of the incorrect attempts should be incremented by exactly one. This scenario can be represented as the variable *attempts* is incremented as an action on the transition (S3 → S4) when the state *S3* detects an incorrect attempt. Similarly, the authentication algorithm should check whether the user has already been locked out due to previous incorrect attempts before granting access. This check can be applied as a guard condition on the transition (S2 → Final).

The third security specification places a restriction on the number of incorrect login attempts. The number of allowed incorrect attempts should be reasonable. Usually a minimum of three incorrect login attempts are allowed after which the user is locked. An excessive number of allowed incorrect login attempts may open the door for brute force attacks like the dictionary attack[1 & 2]. In our case, we have set this limit at three as indicated by the event on the transition from S3 → S6. This indicates that a check can also be represented as an event that triggers a transition. Moreover, according to the specification, if the password is incorrect and the current login attempt is only the first or

the second, then the authentication mechanism should only deny access. This condition is again denoted as a guard on the transition between states S4 and Final.

The UML state machine representation of the security specifications listed in Table 1 is presented in Figure 1. The notations for the relational and assignment operations in the figure are consistent with the C/C++ programming language. Each constraint in the security specifications is represented as a decision point which in turn is depicted as a state with two or more outgoing edges. Each outgoing edge is representative of one option of the decision point. For example, we have five decision points in the state machine diagram in Figure 1. These decision points are at states S1, S2, S3, S4, and S5.

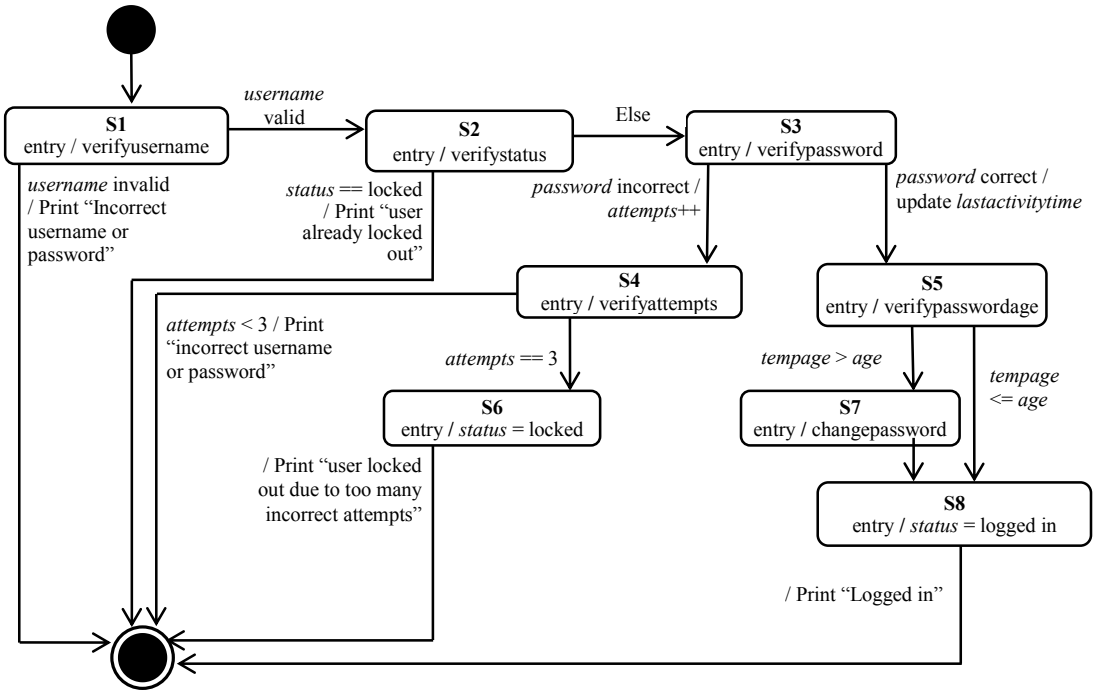

Fig. 1. UML representation of dynamic security specifications for the authentication mechanism

S1 branches off based on the validity of the username; however this is not the result of any security specification. S2 branches off based on whether the user is currently locked out or not and S3 branches off based on the correctness of the password. Branches in S2 and S3 are a result of the constraints in the second security specification. S4 branches off based on number of incorrect password attempts and is based on the second half of the third security specification. Finally, S5 branches off based on the age of the password and is based on the fifth security specification. It should be noted that all the decision points directly correspond to the word "if" in each of the security specifications. Correspondingly, all these decision points will be implemented as if-else statements in the actual code. The first security specification is different in that it does not need a decision point: it is a statement and is fulfilled by an action *i.e.*, *attempts*++ on the transition from S3 to S4.

3.4. Benefits of the proposed approach

The representation of low-level design which can be directly translated to code has the immense benefit of removing any ambiguity and inconsistencies that may arrive when a high-level design is handed over to the developers. It should be noted that most of the security vulnerabilities are a direct result of not following the secure design decisions or ignoring security requirements in the implementation phase. Another advantage of representing

security specifications using UML state machine diagrams is that the security specifications from the requirements and high-level design stages can be easily traced in the low-level design and eventually in the code.

Developing the UML state machine diagrams with security specifications will have some associated cost. However, UML state machine diagrams are already used to represent low-level design details during software development and security specifications only serve as restrictions on the already developed dynamic behavior. Hence, the overhead of introducing security specifications into UML state machine diagrams will be minimal. A further advantage will be that the same state machine diagrams can be used to develop test cases that will also include security scenarios. Therefore, any additional effort regarding security testing will not be required.

4. Conclusions

Software can be designed to function in a secure manner while it is being developed. This involves incorporating security specifications in the design of the software. Traditionally, only the high-level design is annotated with security related information such as the inclusion of certain security mechanisms. However, security specifications dictate certain constraints on the behavior of the software. These constraints are necessary to fulfill the security specifications. When the high-level design is used for implementation purposes, there is a possibility of incorrect implementation. Hence, there is a need to present the developers with detailed low-level design of the dynamic behavior of the software incorporating the constraints imposed by the security specifications.

To achieve this goal, we, in this paper, argue through the example of an authentication mechanism that UML state machine diagrams are capable and should be used to represent low-level dynamic behavior imposed by security specification. We have incorporated some of the existing security specification from the CWE and OWASP databases in our authentication mechanism. As UML state machine diagrams are already being used in the software industry therefore any additional effort for learning a new security specification language is not required. Moreover, vulnerabilities due to inconsistencies between design and code are also avoided because the developers have been provided with low-level design. A future work maybe to develop UML state machine diagrams for all the security specifications from CWE and OWASP and store them in an open-access repository for designers and developers to freely access.

References

1. Common Weakness Enumeration (CWE), www.mitre.org/cwe, Last visited May 2015.
2. OWASP, www.owasp.org, Last visited May 2015.
3. J. Juerjens, *Secure Systems Development with UML*, Springer, 2005.
4. 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.
5. 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.
6. 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.
7. 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.
8. D.G. Firesmith, “Security Use Cases,” *Journal of Object Technology*, 2003, vol. 2, no. 3, pp. 53-64.
9. M. U. Khan and M. Zulkernine, “Building Components with Embedded Security Monitors," In *Proceedings of the 2nd ACM SigSoft International Symposium on Architecting Critical Systems (ISARCS)*, Boulder, Colorado, USA, 2011, ACM Press, pp. 133-142.
10. M. U. Khan and M. Zukernine, M., “A Hybrid Monitoring of Software Design-Level Security Specifications,” In *Proceedings of the 2014 International Conference on Quality Software (QSIC’14)*, Dallas, Texas, USA, 2014, pp. 111-116.
11. J. Wilander, "Policy and Implementation Assurance for Software Security," Doctoral dissertation, Department of Computer and Information Science, Linköping University, Sweden, 2005.