Defining Security Requirements Through Misuse Actions

Eduardo B. Fernandez, Michael VanHilst, Maria M. Larrondo Petrie, and Shihong Huang
Department of Computer Science & Engineering
Florida Atlantic University
777 Glades Road, SE-300, Boca Raton, Florida 33431-0991 USA
{ed, mike, maria, shihong}@cse.fau.edu
URL: http://www.cse.fau.edu/~security

Abstract. An important aspect of security requirements is the understanding and listing of the possible threats to the system. Only then can we decide what specific defense mechanisms to use. We show here an approach to list all threats by considering each action in each use case and analyzing how it can be subverted by an internal or external attacker. From this list we can deduce what policies are necessary to prevent or mitigate the threats. These policies can then be used as guidelines for design. The proposed method can include formal design notations for validation and verification.

1 Introduction

Defining security requirements is difficult and there is no generally accepted way [1], [2], [3], [4], [5]. An important aspect of security requirements is the listing of the possible threats to the system. Only then can we decide what specific defense mechanisms to use. A threat is a potential attack, while an attack is an actual misuse of information. Most approaches consider only the effect of low-level attacks; e.g., taking control of the database system through a buffer overflow attack. There are two problems with this approach: the number of such threats is very high, and we need to make assumptions about a system that has not yet been built. A way to avoid the first problem is the use of sets of generic attacks [6], but this approach cannot avoid the second drawback.

We believe that we should look at the higher levels of the system. An attacker has an objective or goal that he wants to accomplish, e.g., steal the identity of a customer, transfer money to his own account, etc. Security requirements should

define the needs of the system without committing to specific mechanisms. We show here an approach to list threats by considering each action in each use case and seeing how it can be subverted by an internal or external attacker. We assume that the functional use cases have already been defined or are being defined concurrently. From the list of threats we can deduce what policies are necessary to prevent or mitigate the attacks. The proposed method is extendable to include formal design notations for validation and verification; we explore some possibilities. While there is no guarantee that our approach produces all possible threats, it appears superior to other approaches with similar objectives.

A related approach is the concept of misuse cases [1], [7]. Misuse cases are independent use cases initiated by external attackers to the system. That approach, by itself, lacks completeness because it is not clear what misuse cases should be considered. Another related approach is risk analysis. In risk analysis, threats to the successful completion and use of the system are identified and analyzed. Threat likelihood and consequences are considered in a cost benefit analysis, and plans are made to address them. Risk analysis, per se, lacks a method of systematically identifying the threats, it concentrates on the effect of threats on the system.

In previous work we introduced a methodology for secure systems design that uses architectural layers and security patterns [8], [9]. An important aspect of that methodology is the emphasis on approaching security at all stages. The approach presented here would be one of the first stages in using that methodology.

Section 2 discusses some background on use cases. Section 3 presents the concept of misuse actions and shows through an example of how to relate threats to use cases. Section 4 shows how we can define policies to prevent the identified attacks. Section 5 compares our approach to other approaches. The paper ends with some conclusions.

2 Use cases, threats, and policies

Use cases are interactions of a user with the system [10]. The set of all use cases is described by a UML Use Case diagram. Each use case is described by a textual template identifying actors (or stakeholders), preconditions, postconditions, normal flow of execution, and alternate flows of execution. Sequence diagrams may complement the textual descriptions. Use cases are not atomic but consist of a sequence of actions. For example, in a use case to borrow a book from the library one must check if the user has a valid account (first action), she is not overdue (second action), the copy of the book is set to not available (third action), etc. Complex use cases may have many actions. Since use cases identify the actor that performs the use case, we can also identify who is the possible attacker.

As indicated earlier, an attacker has an objective or goal that he wants to accomplish. To accomplish his purposes, he must interact with the system trying to subvert one or more actions in a use case (he might do this indirectly). Low level actions, such as attacking a system through a buffer overflow, are just ways to accomplish these goals but not goals in themselves. Looking at use cases is consistent with the idea that security must be defined at the highest system levels, a basic principle for secure systems [11].

There is a large variety of possible security policies and it is not clear in general, which ones are needed in a given system. Once we understand the possible threats, we can define policies to stop them. These policies are used in turn to guide the selection and implementation of security mechanisms; for example where in the system we should use authentication and the type of authentication required. If the threats indicate that we require authorization we can then find the specific authorization rules that are needed. In an earlier paper we proposed a way to find all the rights needed by the actors of a set of use cases in an application [12]. The idea is that all the use cases of an application define all the possible interactions of actors with the application. We need to provide these actors with rights to perform their functions. If we give these actors only those rights, we are applying the basic principle of least privilege. If we define appropriate rights, attacks can be prevented or mitigated.

3 Threats and actions

We illustrate our approach through an example. Consider a financial company that provides investment services to its customers. Customers can open and close accounts in person or through the Internet. Customers who hold accounts can send orders to the company for buying or selling commodities (stocks, bonds, real estate, art, etc.). Each customer account is in the charge of a custodian (a broker), who carries out the orders of the customers. Customers send orders to their brokers by email or by phone. A government auditor visits periodically to check for application of laws and regulations. Figure 1 shows the Use Case diagram for this institution.

Figure 2 shows the activity diagram for the use case "Open account" in this institution, indicating the typical actions required to open an account for a new customer. We indicate "swimlanes" for Customer and Manager, the two actors involved in this use case [13]. These actions result in new information, including objects for the new customer, her account, and her card-based authorization.

Potentially each action (activity) is susceptible to attack, although not necessarily through the computer system. Figure 3 shows the same activity diagram showing possible threats and including a new swimlane for an external attacker. For this use case we could have the following threats:

- A1. The customer is an impostor and opens an account in the name of another person
- A2. The customer provides false information and opens an spurious account
- A3. The manager is an impostor and collects data illegally
- A4. The manager collects customer information to use illegally
- A5. The manager creates a spurious account with the customer's information
- A6. The manager creates a spurious authorization card to access the account
- A7. An attacker tries to prevent the customers to access their accounts (denial of service)
- A8. An attacker tries to move money from an account to her own account

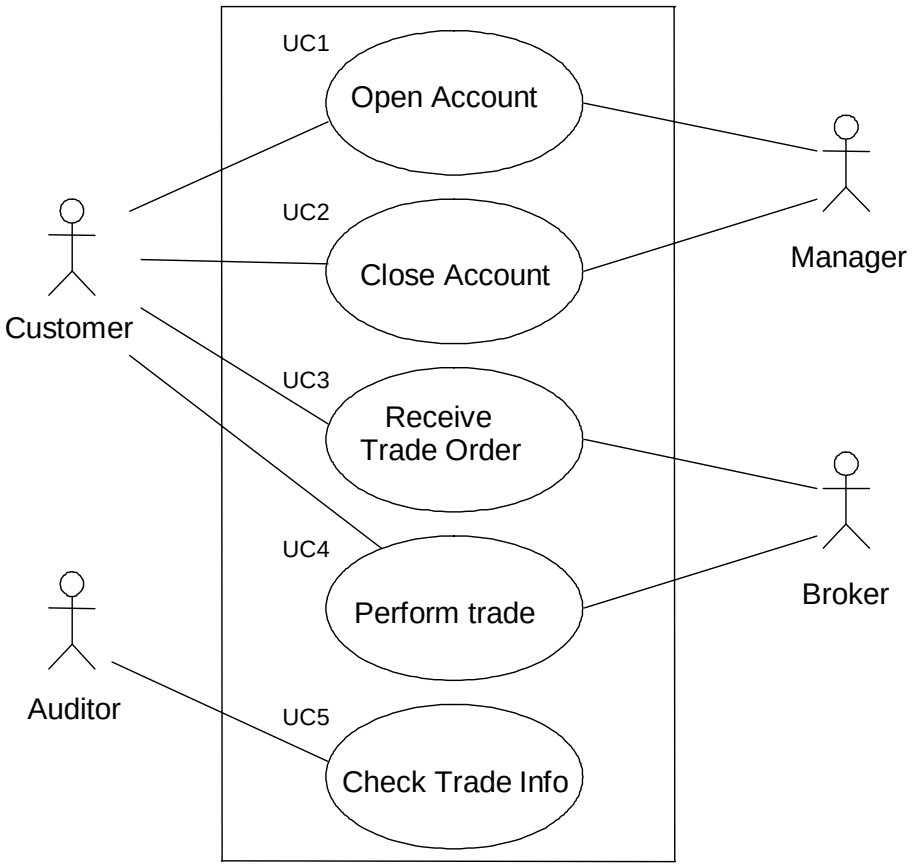

Fig. 1. Use cases for a financial institution

In the activity diagram in Figure 3 the attacks are shown as misuse actions (dotted lines). Undesired consequences in the form of additional or alternative objects (dotted lines) have also been added. With these annotations, the attacks and vulnerabilities presented by the use case become part of our understanding of the use case and are explicit in its analysis.

Note that:

- We can identify internal and external attackers. The actors in these attacks could be external attackers (hackers), acting as such or hackers impersonating legitimate roles. It is also possible that a person in a legitimate role can be malicious (internal attacks). For example, A1 and A3 are performed by external attackers; A2, A4, A5 and A6 are performed by insiders, while A7 and A8 are performed by either external or internal attackers.

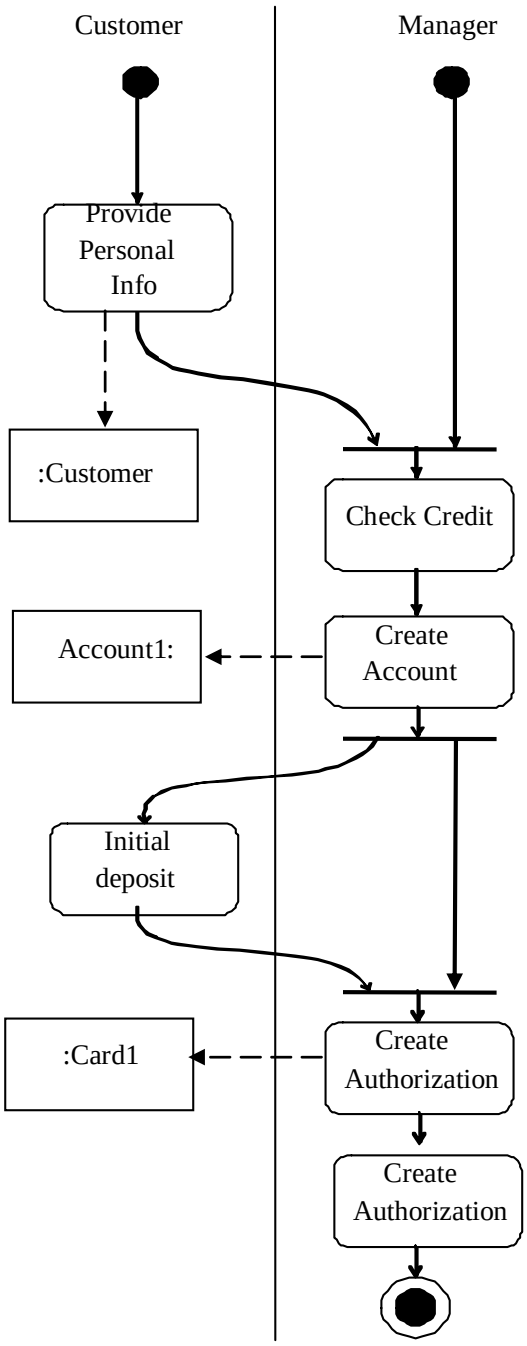

Fig. 2. Activity diagram for use case “Open account”

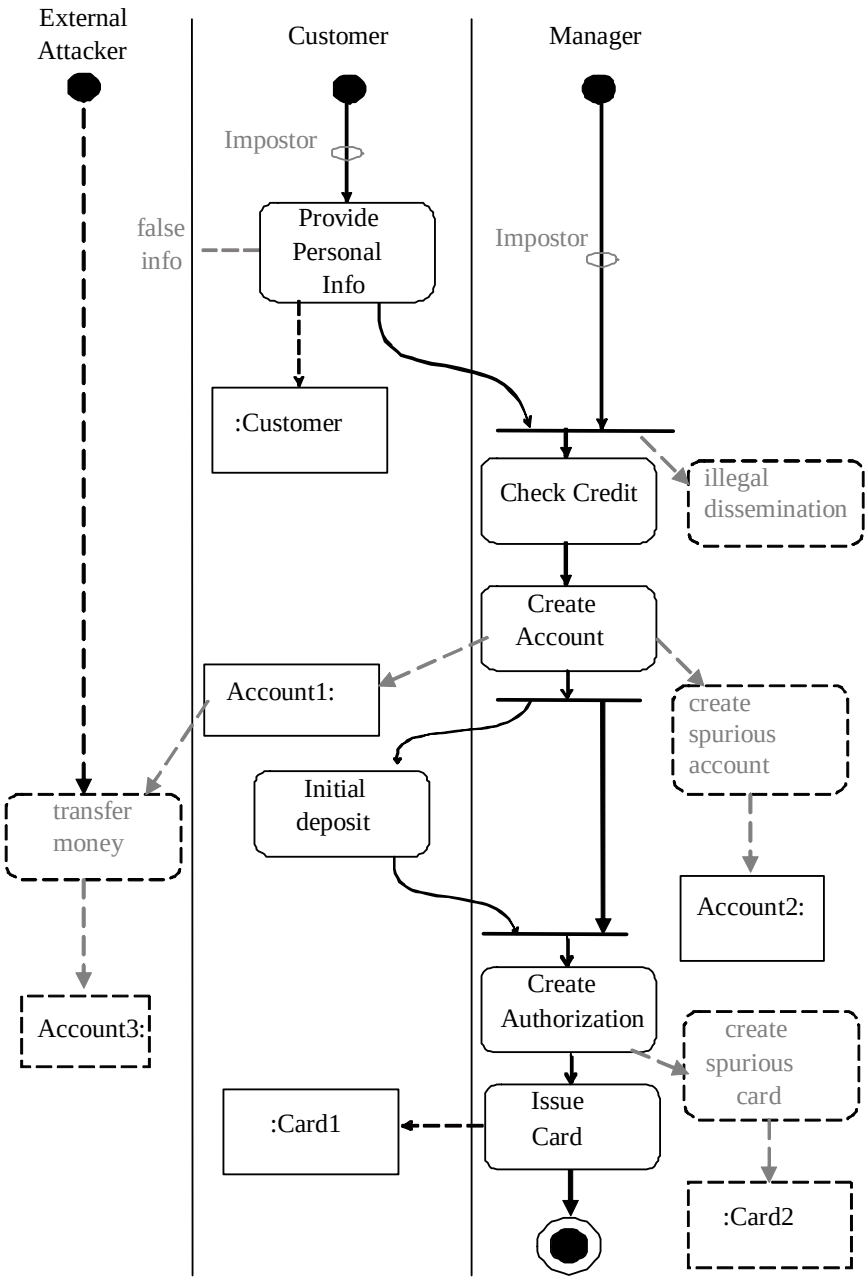

Fig. 3. Activity diagram for use case “Open account” showing misuse actions

- We can list systematically all (or most) possible application threats. While completeness cannot be assured, the fact that we consider all actions in a use case gives us some confidence that we considered at least all important possible attacks. The threats that we postulate come from our experience, from the knowledge of the application, and from the study of similar systems (banking systems have similar threats).
- We can later identify the target of the low-level attacks. Starting from the threats to actions we can look at the lower levels of the systems already designed and search for possible realizations of the threats, e.g. a buffer overflow, bypassing entry points of a procedure, etc.
- Note that we only consider attacks to our system. Attacks to systems that collaborate with our system are beyond our control. For example, credit checking is normally performed using an external service. If that service was compromised we could receive erroneous information about a potential customer and make a wrong decision about his account.
- We are not restricted to analyze each use case in isolation. Some workflows require several use cases, e.g. “Approve a purchase order” can be followed by “Send a purchase order”. We can consider attacks that take advantages of this sequence, for example, by bypassing some steps that perform checks. These threats, in general, are harder to find.
- The sequence used in the example to open an account in a financial institution is very similar to opening an account in a bank, in a club, or in a library. In fact, we can think of it as a pattern and it could be an addition to a pattern for building the corresponding software [14]. Having threat patterns simplifies finding threats for new systems.

4 Stopping or mitigating the attacks

We can now find out what policies are needed to stop these attacks. For this purpose, we can select from the typical policies used in secure systems [11]. This selection should result in a minimum set of mechanisms instead of mechanisms piled up because they might be useful. For example, to avoid impostors we can have a policy of I&A (Identification and Authentication) for every actor participating in a use case.

To stop or mitigate the attacks in the example we need the following policies:

- A1. A3. Mutual authentication. Every interaction across system nodes is authenticated.
- A2. Verify source of information.
- A4. Logging. Since the manager is using his legitimate rights we can only log his actions for auditing at a later time.
- A5. A6. Separation of administration from use of data. For example, a manager can create accounts but should have no rights to withdraw or deposit money in the account.
- A7. Protection against denial of service. We need some redundancy in the system to increase its availability. Intrusion detection and filtering policies should also be useful.

- A8. Authorization. If the user is not explicitly authorized he should not be able to move money from any account.

The lower levels of the system should enforce these policies. If they are properly designed we do not need to identify every low-level threat.

5 Formalization

The analysis of attacks and their prevention can be formalized as shown in Figure 4. The preconditions for undesired consequences are presented in comments. For the analysis we focus only on sufficient preconditions that should not normally be present at that point in the execution of the use case. In some cases the preconditions are simple conjunctions, where all conditions must be present. In other cases, the preconditions may involve more complicated logical relationships among preconditions.

To express relationships among preconditions, we have adopted the concise notation from RSML [15]. Preconditions are represented in tabular form as disjunctions of conjunctions (disjunctive normal form). Each column in the table is a sufficient set of preconditions. Within each column, the role of a precondition literal (True, False, or don't care) is given by the letters T, F, or X. For example, a spurious account could be created either when a malicious manager acts without customer approval, or when there is an error (intended or unintended) in the customer information.

Figure 5 shows the equivalent fault tree representation for one set of preconditions. A fault tree analysis allows probabilities of occurrence to be estimated for each condition or event. The fault tree can be expanded, with sub-dependencies, to assist in this process. In a fault tree a circle or ellipse represents a basic condition, while a diamond represents a condition that could be further elaborated. An error in the customer info is treated as basic – it doesn't matter how or why the error was made. Customer approval could be further expanded, for example to show an "or" condition between customer signing an acknowledgement or customer receiving notification. Similarly, alternative preconditions for a malicious person acting in the role of manager could be explored.

In analyzing risks and their prevention, it is important to make a distinction between the actual desired condition, and the mechanism that is used to achieve it. For example, a good manager is a desired condition for secure transactions. Authorization is a mechanism to reduce the likelihood of a bad manager being able to accomplish his purposes. But authorization is, itself, not the desired goal, and may, in fact, be neither sufficient nor the only means of achieving the goal condition. In this sense, our analysis approach is consistent with the spirit of goal oriented practices [2, 16].

In the formalized analysis, the defense policies and mechanisms must be shown to reduce the probability of each sufficient set of preconditions to an acceptable level of risk. An actual formal analysis is beyond the scope of the present paper. However, we can give a sense of how such analyses could be performed using fault tree and model checking techniques.

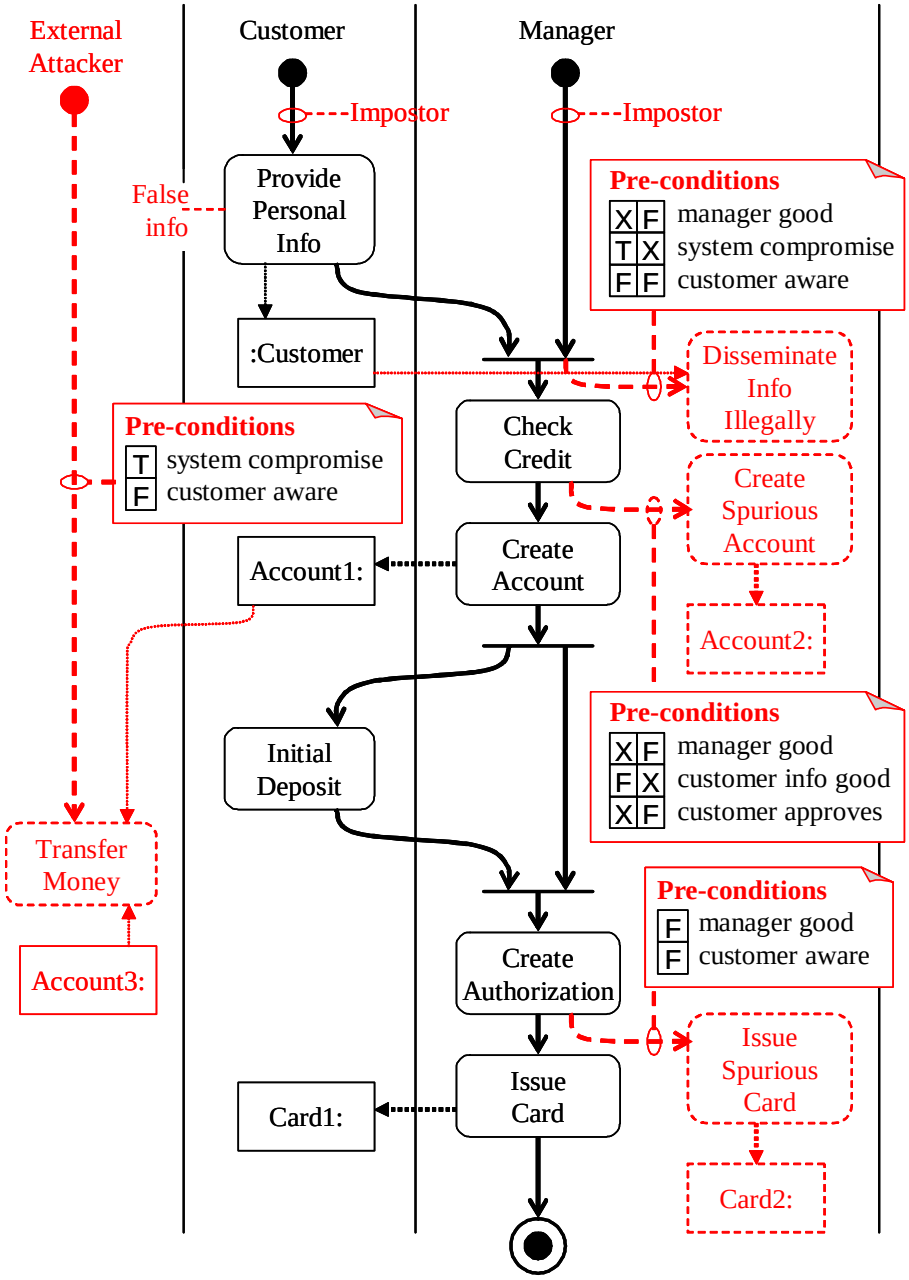

Fig. 4. Formalizing the analysis of attacks and preventions

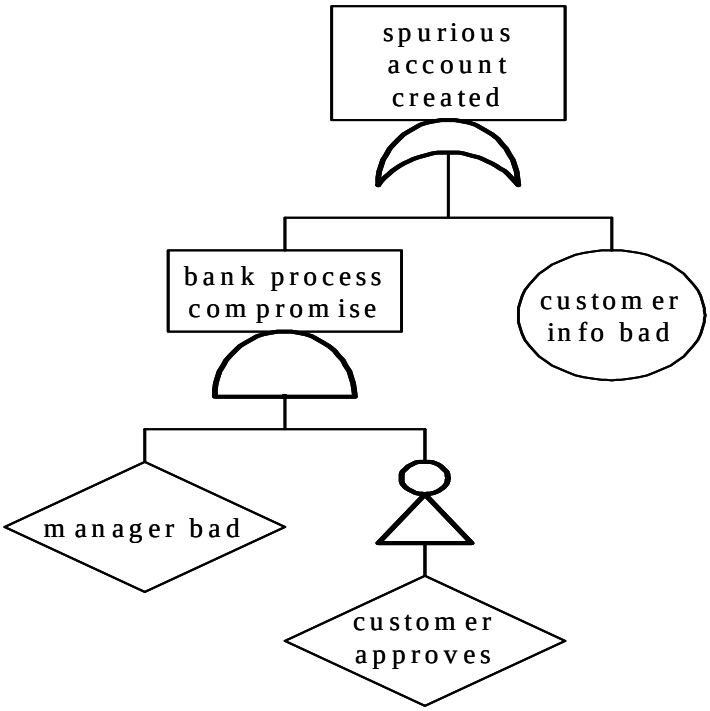

Fig. 5. Fault tree for spurious account creation

Fault tree analysis can assess the effectiveness of chosen defense mechanisms for achieving desired levels of assurance. Fault tree notation is similar to attack tree notation [3], but is more appropriate for risk-benefit analyses and is widely supported by commercially available tools. Probability values are estimated, where needed, and then combined to compute a probability for the occurrence for an insecure or unsafe combination of conditions and events. Continuing the example from above, a fault tree analysis would assign a non-zero value to the likelihood of a dishonest manager receiving authorization.

To perform model checking, the activity diagram can be converted to a state machine. Activities become states (of performing the activity). Precondition sets become the transition conditions to pass from one state to another. Initial values for literals appearing in the transition conditions must be set (to True, False, or Don't know). Defense mechanisms included in the state machine change the values of literals when visited.

6 Discussion

The closest approach to ours is clearly the one based on misuse cases [1], [7]. Misuse cases are not developed systematically and it is easy to miss important attacks. That approach also uses other use cases to mitigate or prevent attacks. Use cases are interactions of users with the system but attack prevention cannot be done in general through additional interactions. We need instead security policies and the corresponding mechanisms to implement them. Misuse cases because of their reliance on whole use cases they need to define new stereotypes such as “threaten” and “mitigate” use cases, while we just use standard use cases. We do not think that the emphasis on protecting assets is also the best for information systems. Emphasis on assets makes sense when we are talking of physical assets that can be stolen. Information security is about preventing illegal reading or modification of information as well as assuring its availability. It makes then more sense to defend against specific actions, e.g. stealing identity, instead of protecting the identity database.

The group at the Open University in the U.K. has done a significant amount of work on security requirements [17], including the use of abuse frames to lead to security requirements (an abuse frame is similar to a misuse case but using Jackson’s problem frames [18].

[2] discusses requirements for secure systems using the concept of goal-oriented requirements. Other authors also have focused on security requirements [5], [19] but none of them consider use cases. Mouratidis and his group use a special methodology, Tropos, to model security. Their approach to develop requirements does not consider use cases either [20].

Van Lamsweerde considers anti-models, which describe how specifications of model elements could be maliciously threatened, why and by whom [21]. His approach combines ideas from misuse cases and goal-oriented requirements.

All these models consider a coarser unit that can be attacked and are less systematic than our approach.

7 A methodology to build secure systems

This work is part of a methodology to build secure systems. Of course, it does not need to be applied as part of this approach but the methodology provides a context for our development. A main idea in the proposed methodology is that security principles should be applied at every stage of the software lifecycle and that each stage can be tested for compliance with security principles. Another basic idea is the use of patterns to guide security at each stage [9]. Figure 6 shows a secure software lifecycle, indicating where security can be applied (white arrows) and where we can audit for compliance with security principles and policies (dark arrows).

This project proposes guidelines for incorporating security from the requirements stage through analysis, design, implementation, testing, and deployment. Our approach considers the following development stages:

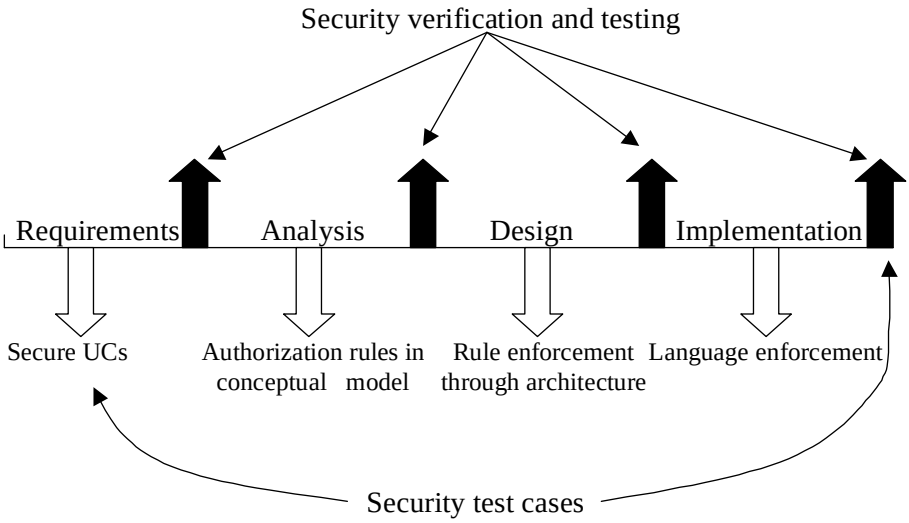

Fig. 6. Secure software lifecycle

Domain analysis stage: A business model is defined. Legacy systems are identified and their security implications analyzed. Domain and regulatory constraints are identified. Policies must be defined up front, in this phase. The suitability of the development team is assessed, possibly leading to added training. Security issues of the developers, themselves, and their environment may also be considered in some cases. This phase may be performed only once for each new domain or team.

Requirements stage: Use cases define the required interactions with the system. Applying the principle that security must start from the highest levels, it makes sense to relate attacks to use cases. We study each action within a use case and see which threats are possible (this paper). We then determine which policies would stop these attacks. From the use cases we can also determine the needed rights for each actor and thus apply a need-to-know policy. Note that the set of all use cases defines all the uses of the system and from all the use cases we can determine all the rights for each actor. The security test cases for the complete system are also defined at this stage.

Analysis stage: Analysis patterns can be used to build the conceptual model in a more reliable and efficient way. Security patterns describe security models or mechanisms. We can build a conceptual model where repeated applications of a security model pattern realize the rights determined from use cases. In fact, analysis patterns can be built with predefined authorizations according to the roles in their use cases. Then we only need to additionally specify the rights for those parts not covered by patterns. We can start defining mechanisms (countermeasures) to prevent attacks.

Design stage: Design stage: when we have the possible attacks to a system, design mechanisms are selected to stop these attacks. User interfaces should

correspond to use cases and may be used to enforce the authorizations defined in the analysis stage. Secure interfaces enforce authorizations when users interact with the system. Components can be secured by using authorization rules for Java or .NET components. Distribution provides another dimension where security restrictions can be applied. Deployment diagrams can define secure configurations to be used by security administrators. A multilayer architecture is needed to enforce the security constraints defined at the application level. In each level we use patterns to represent appropriate security mechanisms. Security constraints must be mapped between levels.

Implementation stage: This stage requires reflecting in the code the security rules defined in the design stage. Because these rules are expressed as classes, associations, and constraints, they can be implemented as classes in object-oriented languages. In this stage we can also select specific security packages or COTS, e.g., a firewall product, a cryptographic package. Some of the patterns identified earlier in the cycle can be replaced by COTS (these can be tested to see if they include a similar pattern).

7 Conclusions

We have presented an approach that produces all (or most) of the threats to a given application. This happens because we consider systematically all actions within a use case and we see how they could be subverted. While all this could be done in the textual version of the use case, the use of UML activity diagrams produces a clear and more intuitive way to analyze these attacks. From the threats we derive necessary policies to stop or mitigate them.

We have now completed the requirements stage and we are ready to start defining the solution to our design problem. Each identified threat can be analyzed to see how it can be accomplished in the specific environment. The list can then be used to guide the design and to select security products. It can also be used to evaluate the final design by analyzing whether the system defenses can stop all these attacks. As we indicated earlier since use cases define all the interactions with the system we can find from them the rights needed by these roles to perform their work (need to know). Future work will concentrate in the transition from the policies to the mechanisms.

When dealing with a complex safety-critical software system, the number and complexity of threats will increase; for example, there may be more than one way to attack a particular action. Without proper mechanisms to represent this information, software developers will have difficulty to effectively digest the information and to validate the design and implementation. Another future work is to find a better way, considering layout, style etc, to document the misuse action diagrams, that can be effective even for complex systems. Some work has been done to assess the efficacy of UML diagrams as one type of graphical documentation [22], [23]. For example, we can use annotated UML activity diagrams and Interaction Overview Diagrams to assess the best way to document misuse actions, according to quality attributes such as completeness and effectiveness.

Acknowledgements

The referees made useful comments that improved this paper. This work was supported by a grant from the US Department of Information Security Agency (DISA), administered by Pragmatics, Inc.

References

1. Alexander, I.: Misuse cases: Use cases with hostile intent. In *IEEE Software*, Vol. 20, No. 1, January/February 2003, IEEE Computer Society Press, Los Alamitos, California (2003) 58-66.
2. Liu, L., Yu, E. and Mylopoulos, J.: Security and privacy requirements analysis within a social setting. In *Proceedings of the 11th IEEE International Conference on Requirements Engineering (RE'03)*, Monterey, California, 8-12 September 2003, IEEE Computer Society Press, Los Alamitos, California (2003) 151-161.
3. Schneier, B.: Attack Trees: Modeling Security Threats. In *Dr. Dobb's Journal*, Vol. 24, No. 12, December 1999, CMP Media LLC., Manhasset, New York, USA (2003) 21-29.
4. Whitmore, . J. J.: A method for designing secure solutions. In *IBM Systems Journal*, Vol. 40, No. 3, IBM, Riverton, New Jersey, USA (2001) 747-768. http://www.research.ibm.com/journal/sj
5. Zuccato, A.: Holistic security requirement engineering for electronic commerce. In *Computers & Security*, Vol. 23, No. 1, Elsevier, UK (2004) 63-76.
6. Howard, M., and LeBlanc, D. *Writing secure code*, (2nd Ed.), Microsoft Press, Redmond, Washington, USA (2003).
7. Sindre, G. and Opdahl, A.L.: Eliciting Security Requirements by Misuse Cases. In *Proceedings of the 37th International Conference on Technology of Object-Oriented Languages and Systems* (TOOLS-Pacific 2000), Sydney, Australia, 20-23 November 2000 IEEE Press, Los Alamitos, California, USA (2000) 120 –131.
8. Fernandez, E. B.: A methodology for secure software design. In *Software Engineering Research and Practice: Proceedings of the International Conference on Software Engineering Research and Practice*, SERP '04, Las Vegas, Nevada, USA, Vol. 1, 21-24 June 2004, H. R. Arabnia and H. Reza (eds.), CSREA Press, USA (2004) 130-136.
9 Fernandez, E. B., Larrondo-Petrie, M. M., Sorgente, T. and VanHilst M.: A methodology to develop secure systems using patterns. In *Integrating security and software engineering: Advances and future vision*, H. Mouratidis and P. Giorgini (Eds.), Idea Group, Hershey, Pennsylvania, USA (2006).
10. Larman, C.: *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rdedition.), Prentice-Hall, Englewood Cliffs, New Jersey, USA (2005).
11. Fernandez, E. B., Gudes, E. and Olivier, M.: *The Design of Secure Systems*, Addison-Wesley, Reading, Massachussetts, USA (2007).
12 Fernandez, E. B., and Hawkins, J.C.: Determining Role Rights from Use Cases. In *Proceedings of the 2nd ACM Workshop on Role-Based Access Control*, RBAC'97, Fairfax, Virginia, USA, 6-7 November 1997, ACM Press, New York, New York, USA (1997) 121-125.
13. Booch, G., Rumbaugh, J. and Jacobson, I.: *The Unified Modeling Language User Guide* (2nd Ed.), Addison-Wesley, Upper Saddle River, New Jersey, USA (2005).
14. Fernandez, E. B. and Liu, Y.: The Account Analysis Pattern. In *Proceedings of EuroPLoP 2002 (Pattern Languages of Programs)*, Irsee Germany, 3-7 July 2002, Universitätsverlag Konstanz, Konstanz, Germany, (2002). http://www.hillside.net/patterns/EuroPLoP2002/

15. Leveson, N. G., Heimdahl, M. P. E., Hildreth, H. and Reese, J. D.: Requirements specification for process control systems. In *IEEE Transactions on Software Engineering*, Vol. 20, No 9, September 1994, IEEE Computer Society Press, Los Alamitos, California, USA (1994) 684-707.
16. Cleland-Huang, J., Denne, M., Mahjub, G., and Patel, N.: A goal-oriented approach for mitigating security and continuity risks. In *Proceedings. of the IEEE Inernational. Symposium on Secure Software Engineering (ISSSE'06)*,13-15 March 2006, Arlington, Virginia, USA (2006) 167-177.
17. Haley, C.B., Laney, R.C., and Nuseiben, B.: Deriving security requirements from crosscutting threat descriptions. In *Proceedings of the 3rd. International Conference on Aspect-Oriented Software Development (AOSD'04)*, Lancaster, UK, 22-26 March 2004, ACM Press, New York, New York, USA (2004) 112-121.
18. Jackson, M.: Problem *Frames: Analysing and structuring software development problems*, Addison-Wesley, Reading, Washington, USA (2001).
19. He, Q. and Anton, A. I.: Deriving access control policies from requirements specifications and database design, North Carolina State University CS Technical Report. TR-2004-24, (2004).
20. Mouratidis, H.,Giorgini, P. and Manson, G.A.: Using security attach scenarios to analyse security during information systems Design. In *Proceedings of the 2nd International Workshop on Security in Information Systems at ICEIS 2004*, Porto, Portugal, April 2004 (2004) 10-17.
21. van Lamsweerde, A.: Elaborating security requirements by construction of intentional anti-models. In *Proceedings of the 26th International Conference on Software Engineering* (ICSE'04), Edinburgh, UK, 23-28 May 2004, IEEE Computer Society Press, Los Alamitos, California, USA (2004)148-157.
22. Huang, S. and Tilley, A.: Workshop on Graphical Documentation for Programmers: Assessing the Efficacy of UML Diagrams for Program Understanding. Held in conjunction with *The 11th International Workshop on Program Comprehension*, IWPC 2003, 10 May 2003, Portland, Oregon, USA, IEEE Computer Society Press, Los Alamitos, California, USA (2003) 281-282.
23. Tilley, S., and Huang, S.: A qualitative assessment of the efficacy of UML diagrams as a form of graphical documentation in aiding program understanding. In *Proceedings of the 21st ACM Annual International Conference on Design of Communication* (SIGDOC 2003: 12-15 October 2003; San Francisco, California, USA, ACM Press: New York, New York, USA (2003) 184-191.