A Threat Model Driven Approach for Security Testing [*]

Linzhang Wang
Department of Computer Science
Nanjing University
Nanjing, Jiangsu 210093, P.R.China
lzwang@nju.edu.cn

Eric Wong
Department of Computer Science
University of Texas at Dallas
Richardson, TX 75083, USA
ewong@utdallas.edu

Dianxiang Xu
Department of Computer Science
North Dakota State University
Fargo, ND 58105, USA
dianxiang.xu@ndsu.edu

Abstract

In this paper, we propose a novel threat model-driven security testing approach for detecting undesirable threat behavior at runtime. Threats to security policies are modelled with UML (Unified Modeling Language) sequence diagrams. From a design-level threat model we extract a set of threat traces, each of which is an event sequence that should not occur during the system execution. The same threat model is also used to decide what kind of information should be collected at runtime and to guide the code instrumentation. The instrumented code is recompiled and executed using test cases randomly generated. The execution traces are collected and analyzed to verify whether the aforementioned undesirable threat traces are matched. If an execution trace is an instance of a threat trace, security violations are reported and actions should be taken to mitigate the threat in the system. Thus the linkage between models, code implementations, and security testing are extended to form a systematic methodology that can test certain security policies.

Keywords: *threat model, UML sequence diagram, security testing*

1 Introduction

In the current software development process, as the non-functional requirements of the system under construction, software security is implemented and tested after the functional features are implemented. The delay results in continually revising the code in the late phase of software development, which will do harm to the efficiency and quality of software as well as add the cost. System-level penetration testing is a traditional security testing method, which tries to simulate an adversary's attempts to achieve malicious goals in the system. Although similar to software testing which aims to prove that a product works as it should, penetration testing concentrates on exploring software security flaws. In other words, software testing usually verifies that products work as expected in certain scenarios or conditions. This is known as positive testing. Penetration testing, on the other hand, must probe directly and deeply into security risks driven by threats to security policies. Therefore, it is considered negative testing.

Recent research and practices of model-driven engineering and security engineering have advocated negative design of software security that models security threats from the adversary's perspective. Security threats are identified to explain exactly where and how the attackers violate security policies. Published literature[1, 2] already gives a list of observed threats corresponding to certain security policies, we can verify the final system to ensure that violations of these security properties have not been reintroduced into it. We cannot ensure the software is totally correct and meet the security requirement, but we can ensure the software is immune to the exploited threats.

Threat modeling usually describes threats with attack tree or informal description at code-level[1]. It can also be extended to model security threats with UML in the design phase. UML is widely used to model software systems by both academia and industry[3]. UML-based mod-

[*] Supported by the National Natural Science Foundation of China (No.60603036) and the National Grand Fundamental Research 973 Program of China (No.2002CB312001).

eling shifts software development from a code-centric activity to a model-centric activity gradually, and models described in UML and UML profiles are ordinary artifacts employed to specify the expected structure and behavior of the software under construction[4]. The code is still implemented by the programmers based on the design model, and is not automatically generated from the design model. Model-driven testing approaches reuse the design models as the source of test generation and the rational oracle of the test execution[5, 6]. UML is a proper formalism for threat modelling to specify the undesirable behavior. Whether the software meets the security requirement can be determined by threat model driven security testing in a negative way.

This paper propose a threat model-driven security testing method. Threat behaviors are modelled with UML sequence diagram. Then, the threat models are used to driven the security testing of final system. The rest of the paper is organized as follows. Section 2 introduces the threat modeling approach. The systematic approach of threat model-driven security testing is presented in section 3. Related work is presented in section 4, and some conclusions and future work are discussed in the last section.

2 Threat modeling with UML sequence diagrams

A software system consists of different granularity units, such as classes, components, and subsystems, which are realized individually and integrated to constitute the whole system. The system behavior is realized by interactions between the involved units, and the interactions are focused on during the design modeling phase. A threat scenario is an undesirable behavior of the system, and its occurrence would violate security goals. It can be identified by reviewing the key use cases. A UML sequence diagram is an attractive formalism for specifying requirements related to interaction behavior scenarios of software[3]. As security requirements often crosscut functional requirements, threat scenarios can be modelled by sequence diagrams at the design phase. A sequence diagram describes an interaction by focusing on the sequence of messages that are exchanged, along with their corresponding occurrence specifications on the lifelines. It can be used to model the behavior of the system by representing the realization of a use case scenario. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects in order to realize the behavior of the scenario.

To model the threat scenario, the first step we need to take is determine the security policy and potential threats to the system. A security policy says what is allowed and what is not. A threat is a condition that enables someone (i.e., the attacker) to violate the security policy. Threat behaviors are negative scenarios with hostile intent, appear to be new avenues to compensate security requirements. We treat identification of security threats as part of the requirements analysis and model them with UML sequence diagrams. Threats are behaviors that an attacker may pose to the system to violate security properties, such as authentication,authorization,confidentiality, privacy, and availability. They essentially reflect various ways of violating desirable security properties.

UML sequence diagrams are exploited to describe the interactions an attacker would go through to compromise the system, resulting in a threat. The threat scenario is represented by a sequence of message exchanges. Actually, a message sequence is a path from the first message to the end message in the sequence diagram. We can apply certain coverage criteria to the sequence diagram to get the paths, each of which can represent a independent threat scenario. In this paper, we focus on message exchanges via procedure calls.

A critical issue is how security requirements specification can further facilitate the design, implementation, and testing of software systems in which security is a major concern. The threat models in the design phase can be used to find the solution to mitigate the threat, and they also can be used during the security testing for validating the security policy by applying a scenario specification-based method.

In the rest of the paper, a threat model represented by a sequence diagram (SD_{TM}) is viewed as a tuple of (O, M, E) where

- $O = \{o_1, o_2, \cdots, o_m\}$ is a finite set of objects. For any $o_i \in O$, let $f_{OC}(o_i)$ represent the belonging class of object o_i;

- $E = \{e_1, e_2, \cdots, e_k\}$ is a finite set of events. Let f_{eo} be a function from E to O, $f_{eo}(e) = o_i \in O$ means e is the occurrence of the corresponding message in the lifeline of o_i;

- $M = \{m_1, m_2, \cdots, m_n\}$ is a finite set of labelled messages. $\forall m_k \in M$, let $!m_k$ and $?m_k$ represent the send and the receive of m_k, respectively, and $\exists e_i, e_j \in E$ the corresponding sending and receiving event, and let $f_{em}(e_i) = !m_k$ and $f_{em}(e_j) = ?m_k$;

A threat behavior is a scenario of the misuse case and also a sequence of object interaction. For example, an unauthorized user with the wrong password can access the system and an authorized user cannot access the system. This scenario violates the security policy,authentication, and should be prohibited in the implementation. In a UML-based software development process, the threat scenarios

Third International Workshop on Software Engineering for Secure Systems (SESS'07)
0-7695-2952-6/07 $20.00 © 2007 IEEE

are modelled with a sequence diagram in the design phase. Researchers and practitioners of testing focus on testing the code implementation against the model. There may be alternatives and loops in a sequence diagram. So by traversing the possible paths of the sequence diagram, we can get the message sequences. Each message sequence represents a single scenario of the threat model. In the threat model, the messages may be conditioned or iterated, as denoted by the guard conditions in the label of the messages. We can derive alternative message sequences from the complicated sequence diagrams by applying gray-box testing coverage criteria[5]. For a threat model $SD_{TM} = (O, M, E)$, msq is a message sequence in the form $(m_1 \rightarrow m_2 \rightarrow ... \rightarrow m_j \rightarrow m_k \rightarrow ... \rightarrow m_n), \forall i, 1 < i < n, m_i \in M, m_i \rightarrow m_j$ (i.e., m_j is the direct successor of m_i,we use $\rightarrow$ to denote the precede relationship between two elements).

Because the runtime behavior of software is a event sequence of method calling and method execution, to verify whether the threat behavior described by the threat model was implemented, the sequence of events should be derived from the sequence diagram which represents the trace of the threat behavior. The set of messages and the sequence of the messages can be generated by analyzing the sequence diagram. A pair of message sending event and message receiving event can be determined from each message. If the sequence between any two events can be determined, the message sequence can be refined into an event sequence. Thus the threat trace of event sequences can be generated from the threat model. The semantic of a message is simply a trace of a message sending event and message receiving event, because at runtime each message is realized by a message sending event and a message receiving event. Hence, a system behavior that is represented by a message sequence can also be represented at a finer granularity in terms of a sequence of sending and receiving events.

In a UML sequence diagram, message passing can be synchronous or asynchronous. In the case of the former, sending and receiving of the message is said to take place in a sequential order, whereas the asynchronous case treats the sending and receiving of message as distinct events taking place at different times. As for the asynchronous message passing, because of the infinite transmit time of messages and multi-threaded nature of object-oriented software, an object can only control the time of the message sending event and not the message receiving event. So we can only determine the message sending event prior to the corresponding message receiving event, and the sequence between two sending events. The time sequence between the receiving events of different messages cannot be determined until runtime, so any sequence of these events is possible. Based on previous work[7], the time sequence between any two events of the sequence diagram can be determined with the help of message sequence. Since the message sequence can be directly extracted from the sequence diagram, so the possible event sequences can be derived to represent specific behaviors more precisely.

If the time sequence between any two events of the sequence diagram was determined, the sequence of events which represents the threat modelled in the sequence diagram specification has been determined. It is a run of a sequence diagram in a certain scenario, we name it as threat trace, denoted by $ttrace$. Let $SD_{TM} = (O, E, M)$ be a threat model, $\forall e_i \in E, 1 < i < n$, and an event sequence $e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow e_n$ is an $ttrace$, which represents one behavior scenario of SD_{TM}, where $e_i = (eid, etype, sender, receiver, method), \exists m \in M, f_{eo}(e_i) =!m$ or $?m$.

1. eid is a serial number, and uniquely denotes an event.

2. $etype = iif(f_{em}(e_i) =!m, 's', 'r')$ is the type of event, $'s'$ represents the message sending event, and $'r'$ represents the message receiving and executing event;

3. $sender, receiver$ represent the message sending object and the message receiving object of the event,

   (a) if $f_{em}(e_i) =!m$, then $sender = f_{eo}(e_i)$, $receiver = f_{eo}(?m)$

   (b) if $f_{em}(e_i) =?m$, then $sender = f_{eo}(!m)$, $receiver = f_{eo}(e_i)$

4. $method$ is the method called by the message m.

The traces of the threat model may be implemented by the trace of the program at runtime. So it is used to represent the threat behavior. Based on the threat model, we can systematically introduce the threat model-based security testing method.

3 Threat model driven security testing

The operation of the software system may suffer from security failures which should not occur if the corresponding threats are carefully modelled, analyzed, and tested. The threat model can help programmers take necessary actions to prevent the undesirable threats from being implemented into the system, and can also help testers verify whether the final system is free from the undesirable threat behavior.

Security testing is used to uncover software security threats before release so as to prevent attacks after deployment. However, possible violations of the security properties can be previewed by threat modelling in design phase. We can test the final code against the threat model to ensure that violations of these security properties have not been reintroduced into it. Penetration testing is a traditional security testing method. Just like the black-box and white-box testing method, penetration testing can either be blind or informed. In blind penetration tests, the test team is given no inside knowledge, such as design documentation or source code. With informed penetration testing, the testers are given access to some or all of this information. Given the threat models, we can track the runtime behavior of the corresponding modeled threats in the execution of the code.

We extend our previous research on scenario specification based runtime verification[8] and model-driven testing [5] to the threat specification based negative testing. The exploited threats of specific security policies are expressed as a set of UML sequence diagrams, each of which denotes a certain threat scenario. We can use these threat models to drive the security testing process in the following steps.

First, we derive message sequences from the threat models, and derive the threat traces from each message sequence. Second, we want to track the threat scenario represented in the threat model in a running program. When the code becomes baseline, it can be instrumented using the threat model as a guide, so as to record the trace of threat scenario related method calling and method execution at runtime and also to reduce the cost of irrelevant information collection. Third, in order to exercise the program under test more thoroughly, the random testing method is applied. After we get the input parameters and their corresponding domain, a large amount of test cases can be generated, and arranged into a data pool. Fourth, we recompile the instrumented code, and design a test driver to feed test execution with random test data. The runtime execution traces are recorded into a trace file. Last, the runtime execution traces are matched with the threat traces in order to determine whether the threat is still existing.

The key technique in this paper is to verify the consistency between the threat specification, which is described in a UML sequence diagram, and the code implementation. It views the execution of the program as a event sequence of method calling and method executions. If any of the event sequences recall the possible event sequence in the threat model, then the threat in the model is still present in the code, and an error message will be produced to report the failure. The specification of threat scenarios can be derived by threat modeling, introduced in section 2, and is the basis of this method. The two artifacts in the software development process, the threat model and the program, are input by the proposed approach. The details are explained in this section.

3.1 An Example

The example employed in this paper is based on the project ”Simulation of an Automated Teller Machine” (see also http://www.math-cs.gordon.edu/local/courses/cs211/ATMExample/Links.html). It was implemented in Java. To demonstrate the method in this paper, we simplify both the design model and the code. The software under test will control a simulated automated teller machine (ATM) having a magnetic stripe reader for reading an ATM card, a customer console (keyboard and display) for interaction with the customer, a slot for depositing envelopes, a dispenser for cash , a printer for printing customer receipts, and a key-operated switch to allow an operator to start or stop the machine. The ATM will communicate with the bank's computer over an appropriate communication link. (The software on the latter is not part of the requirements for this problem.) The ATM will service one customer at a time. A customer will be required to insert an ATM card and enter a personal identification number (PIN), and both the card number and the PIN will be sent to the bank for validation as part of each transaction. The authorized customer will then be able to perform one or more transactions. In our case study, we are concerned with the violation of the authorization policy.

3.2 Modeling threat scenarios with UML sequence diagrams

Based on the experience of predictable undesirable threats and the specification of use cases and misuse cases of the system under test, we can derive a specification of threat scenarios from the misuse cases. A threat behavior is a scenario of a misuse case and also a sequence of object interaction. For example, an unauthorized user accessing the service provided by the bank is one threat to the ATM system. This scenario violates the security policy of authorization, and should be prohibited in the implementation. It is undesirable with respect to both the users and the developers. We can model one threat scenario of an unauthorized user accessing the service provided by the bank with a UML sequence diagram, which is depicted in figure 1.

3.3 Deriving threat traces from the threat model

Since the threat model is a simple sequence diagram, we derive a message sequence from the model. Regarding to the complicate sequence diagram, we can derive the set of message sequences by applying gray-box coverage criteria[5]. We use the sequential number of the label of the message represent message respectively, such as $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$. Based on the description in section 2, we can derive an event trace from the message sequence, !1 →?1 →!2 →?2 →!3 →?3 →!4 →?4. Then, the message sending event and the message receiving event of the trace can be replaced by method calling and method execution by corresponding objects respectively. This is an instance-level threat trace, which may be implemented by

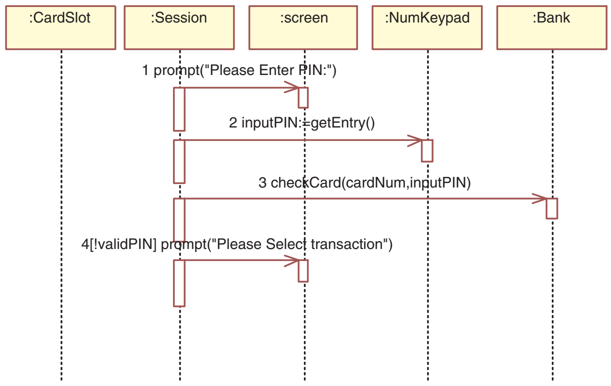
:CardSlot
:Session
:screen
:NumKeypad
:Bank
1 prompt("Please Enter PIN:")
2 inputPIN:=getEntry()
3 checkCard(cardNum,inputPIN)
4[!validPIN] prompt("Please Select transaction")

Figure 1. The sequence diagram for the threat model of access control scenario

the trace of the program at runtime. So it is used to represent the threat behavior and is also the basis of the track.

3.4 Instrumenting the source code

Because we want to track the behavior modeled in the threat model, in order to reduce the cost of irrelevant information instrumentation, only the essential information described in the threat model is focused on. In this paper, our objective is to collect runtime information, such as the method name and the class name of the method calling and the method execution. The insert position in the code is determined by the feature of the information to be collected. Traditional instrumentation is only concerned with the method execution[9]. We want to track the runtime behavior in finer granularity. The probe statement of method calling is inserted before the calling statement of method calling, and the probe statement of method execution is inserted before the first statement of the method definition of the method executing. Thus we can monitor method calling and method execution event pair at runtime execution. First, the source code is scanned for parsing tokens. Once a related method calling and method definition m is found, if it is a definition of method m, then we revise the formal parameter list by adding a formal parameter mid and insert the code segment before the first statement in the method definition for gathering the information about the receiver and the class it belongs to. If it is a calling for method m, then we revise the actual parameter list according to the formal parameter mid and insert the code segment before the method calling for gathering the information about the sender and the class it belongs to. The parameter mid is used to pair a method calling and its corresponding method execution. Thus the caller and callee can be related. The dynamic behavior, such as method calling and method execution, can be profiled by the execution of the instrumented statement and recorded into a trace file. The method calling and the method executing in the source code are instrumented to record the method name and occurring sequence. The detail algorithm is not listed due to the limited place.

3.5 Monitoring the test execution driven by random test cases

Penetration testing is used to exercise the software under test to detect the specific threat traces. Test case generation is one of the important concerns in software testing. Generating test cases based on the design model is not a new topic. Most published literatures introduce techniques for generating test cases from UML models, such as sequence diagrams or activity diagrams, and so on[5, 6]. Any diagram-based test method is based on path traversing[9]. A run driven by one test case may not detect the modeled threats, so various runs taking different paths may be necessary to find a path which can activate the threat behavior. So we randomly generate a number of test cases automatically which can be used to drive the program execution to collect the traces which represent the behavior of the system and can be used to match the threat traces. Test case generation is customized by the user interactively. Given the input parameters and their corresponding domain, specified size of test cases are randomly generated to construct a test suite. The test data is sequentially organized in a data pool (i.e., a .txt file). Each line of the file is a test case. The instrumented program is then recompiled and driven by the randomly generated test cases without user interference. When the execution needs input, test data is fed from the data pool. When the instrumented statements in the source code are executed, the method calling and method execution will be reported to the trace file (also a .txt file). A trace segment corresponds to a random test case. And flags are set between any two traces so as to conveniently differentiate them. The trace file is the operational profile of the runtime behavior, which can be visualized as the runtime execution trace. The tester then can conduct the comparing operation.

3.6 Matching the execution traces with the threat traces

In this section, we check the program execution traces for the threat model by matching the execution traces with the threat traces. Each runtime execution trace is a sequence of method call and method execution, which correspond to the message sends and receives in the threat model because of the guided instrumentation. From the trace file,

Third International Workshop on Software Engineering for Secure Systems (SESS'07)
0-7695-2952-6/07 $20.00 © 2007 IEEE

we find the execution trace, denoted as $etrace$, which is a sequence of method calling and method execution events at runtime, such as $re_1 \rightarrow re_2 \rightarrow \cdots \rightarrow re_m$, where $re_i = (mid, etype, o_s, o_t, method)$, mid is the unique flag of the message, $etype$ could be s or r representing the method calling event or method execution event, respectively, o_t is the name of the target object of method call, o_s is the name of the source object which issued the request of the method call, and $method$ is the name of currently calling or executing method. Thus both $ttrace$ and $etrace$ are sequences of method calling events and method executing events. They can be compared directly, while the unexpected threat behavior and the runtime behavior can not. If an $etrace$ matches a $ttrace$, we can conclude that an undesirable threat behavior is detected, and some measures should be taken to mitigate the threat.

4 Related work

Secure software design and security testing is focused on by more and more professionals. As UML is one of the most popular modeling languages, some of the security research applies UML as the modeling language in security design. Most of them demonstrate how to describe the expected security by different methods to design secure software with UML[10, 11]. Few of them are concerned about the negative design from the adversary's perspective, such as security threats, so as to let the programmer know what's undesirable and should be absent from the system under development[1, 12, 16]. Threat modeling is a sound approach to address software security from the attacker's viewpoint at the design level, but most of them are modelled by attack trees or just text format[1]. Pauli and Xu [16] introduced an approach to model the threats with UML sequence diagrams. But they only highlighted and mitigated security violations, without considering the security testing.

Although UML and model-driven engineering are popular in the industry, to our knowledge, code is still implemented based on the design model, and is not generated automatically from the model. So the design model meeting the security requirement does not assure that the final implementation satisfies the security requirements. In this circumstance, we should verify whether the implementation meets the security design and is absent of the modelled threats. Penetration testing is a traditional security testing method. Blind penetration testing method can be easily applied in an ad hoc manner. But it can hardly be automated, and thus is a time-consuming, labor intensive technique and requires the users to have a strong security background. So far, informed penetration only accesses the code information, without considering the design information[1, 13]. Swinderski and Snyder[1] also presents the idea of threat model based penetration testing, but also manually. Different from previous works, we use UML sequence diagrams to model the threats. Both the threat model and the code are combined and used in the security testing. A runtime verification technique[8, 17, 18] is applied in the testing process for monitoring the execution trace of the software. Specification-based testing method is usually used to generate test cases[5, 6, 9]. However in this paper, it is applied to derive threat traces from threat models. We use a random method to generate a large number of test data, and we also design a test harness to drive the test execution, to feed the test data. The approach proposed in this paper is highly automated, and can be easily deployed. It is more usable especially in the industry for software engineers without detailed knowledge of formal methods and security.

5 Conclusion and future work

In this paper, a threat model-driven security testing method is proposed. The potential threats which violate the security policies are described in the threat model based on the users' previous experience. We track the runtime behavior based on the threat models. The test process is driven by the threat models, and the test design occurs in parallel with the software development right after the threat models become baseline. The threat traces are derived from the threat model to represent the undesirable threat behavior. The test cases are automatically generated with a random testing method. After the code becomes the baseline, it is instrumented guided by the threat models. The instrumented code is recompiled and executed with the randomly generated test cases. The runtime behavior of the system is monitored, and the execution traces can be recorded into a trace file. Thus we can match the execution traces with the threat traces so as to detect the modelled threats in the program behavior.

The proposed approach extends our scenario specification-based runtime verification and mode-driven testing method to security testing based on negative design specification, i.e., threat model. It is highly automated but a little blind. With the help of our method, the development team and customers can make sure whether the system under test is absent of the modelled threats. Thus they can achieve confidence in their product in regard to security concerns.

The UML-based approach can increase the chance of the work having some impact in industry, notably in widely-used development processes and tools, because in a UML-based software development process, if the testing is also based on UML, the relationship between the testing and the design model is more manifest. In addition, this will make the testing approach more accessible. The approach presented in this paper can be easily applied in practice. As we acquire the knowledge of security threats, the threats can be modelled, which can be used to detect whether the implemented code exercises the modelled threats. Thus software testing engineers can use such approach to detect possible security violations without security expertise. However, the approach still need to be used in more real projects. In the future, we will create a repository of the observed threats with respect to specific security policies guided by the security expert, as well as implement a systematic tool to facilitate the security test. Then security engineers can model the threats from the adversary's perspective. And we can assist the security testers with the threat model-driven security testing tool.

References

[1] F. Swinderski, W. Snyder, Threat modeling, Microsoft Press, 2004.

[2] D. Gilliam, J. Powell, E. Haugh, M. Bishop, Addressing software security and mitigations i n the life cycle, Proceeding of the 28th Annual NASA Goddard Software Engineering Workshop (SEW '03), 2003.

[3] M. Fowler,UML Distilled (3 Edition), Addison Wesley, 2003.

[4] Stuart Kent, Model Driven Engineering, Third International Conference on Integrated Formal Methods (IFM 2002), LNCS 2335, pp. 286-298, 2002.

[5] L. Wang, Research on Model-driven Software Testing, [PH.D Dissertation], 2005 [in Chinese].

[6] L. Wang, J. Yuan, X. Yu, J. Hu, X. Li, G. Zheng, Generating Test cases From UML Activity Diagram based on Gray-box Method, Proceeding of 11th Asia-Pacific Software Engineering Conference(APSEC'04),Busan, Korea,November 30-December 04,2004, pp.284-291.

[7] X. Li and J. Lilius. Timing Analysis of UML Sequence Diagrams, UML'99 , Lecture Notes in Computer Science 1723, Springer, 1999, pp.661-674.

[8] X. Li, L. Wang, X. Qiu, B. Lei, J. Yuan, J. Zhao, and G. Zheng,Runtime Verification of Java Programs for Scenario-Based Specifications. In Proceedings of the 11th International Conference on Reliable Software Technologies (AE2006), Portugal, 2006, Lecture Notes in Computer Science 4006, Springer, pp.94-106.

[9] B. Beizer, Black-box Testing:Techniques for functional testing of software and systems, John Wiley and Sons,Inc, New York 1995.

[10] Jan Jürjens, Secure Systems Development with UML, Springer-Verlag, 2004.

[11] T. Lodderstedt, D. A. Basin, J. Doser, SecureUML: A UML-Based Modeling Language for Model-Driven Security, The proceeding of 5th International Conference on the Unified Modeling Language (UML), 2002, LNCS Volume 2460 426-441.

[12] M. Howard, D. Leblanc, Writing secure code(Second edition), Microsoft Press, 2004.

[13] B. Arkin, S. Stender,G. McGraw, Software penetration testing, Security and Privacy Magazine, IEEE Volume 3,Issue 1, Jan-Feb 2005,pp.84-87.

[14] G. McGraw, Software security, Security and Privacy Magazine, IEEE Volume 2, Issue 2,Mar-Apr 2004,pp.80-83.

[15] B. Potter, G. McGraw, Software security testing, Security and Privacy Magazine, IEEE Volume 2,Issue 5, Sept.-Oct. 2004,pp.81-85.

[16] J. Pauli, D. Xu, Threat-Driven Architectural Design of Secure Information Systems, In Proceeding of First International Workshop on Protection by Adaptation (PBA 2005, In conjunction with the 7th International Conference on Enterprise Information Systems (ICEIS 2005)), Miami, May 2005.

[17] M. Lettrai, J. Klose. Scenario-based monitoring and testing of real-time UML models. In Proceedings of 4th International Conference on Unified Modeling Language (UML2001), LNCS 2185, Springer, 2001.

[18] K. Havelund,G. Rou. Monitoring Java Programs with Java PathExplorer. In Electronic Notes in Theoretical Computer Science, Vol.55, Issue 2, Elsevier, 2001,pp.200-217.