Towards Model-Based Automatic Testing of Attack Scenarios

M. Zulkernine[1], M.F. Raihan[1], and M.G. Uddin[2]

[1]School of Computing, [2]Department of Electrical and Computer Engineering
Queen's University, Kingston, Ontario, Canada K7L 3N6
{mzulker,raihan,gias}@cs.queensu.ca

Abstract. Model-based testing techniques play a vital role in producing quality software. However, compared to the testing of functional requirements, these techniques are not prevalent that much in testing software security. This paper presents a model-based approach to automatic testing of attack scenarios. An attack testing framework is proposed to model attack scenarios and test the system with respect to the modeled attack scenarios. The techniques adopted in the framework are applicable in general to the systems, where the potential attack scenarios can be modeled in a formalism based on extended abstract state machines. The attack events, i.e., attack test vectors chosen from the attacks happening in real-world are converted to the test driver specific events ready to be tested against the attack signatures. The proposed framework is implemented and evaluated using the most common attack scenarios. The framework is useful to test software with respect to potential attacks which can significantly reduce the risk of security vulnerabilities.

1 Introduction

A software vulnerable to different attacks can lead to catastrophic failure which can range from hindering normal service quality to causing dangers to human life. Therefore, software systems should be tested whether they exhibit any attack behavior when they are under potential attacks[1]. A software system under security testing is tested for security vulnerabilities with respect to specific security requirements. Model-based testing approaches provide techniques for testing system behavioral conformance to specific functional requirements [1,2]. A model-based approach to security testing involves developing models of security requirements and then testing security properties of the modeled system by automatically generating test vectors [3,4]. Testing attack behavior of a system involves modeling of attack scenarios and verifying the modeled attack scenarios against automatically generated system events. Modeling attack scenarios requires incorporating attack system attributes to the model which might not be present in a traditional modeling language. Moreover, specific testing techniques have to be developed to test the system attack behavior with respect to the modeled attack scenarios.

[1] For brevity, the behavior exhibited by a system under attack is called the attack behavior of the system throughout the paper.

B. Buth, G. Rabe, T. Seyfarth (Eds.): SAFECOMP 2009, LNCS 5775, pp. 229–242, 2009.
© Springer-Verlag Berlin Heidelberg 2009

In this paper, a framework is presented for automatic model-based testing of a system with respect to potential attacks, where the attack behavior is assumed to be modeled using formalisms based on extended abstract state machines [6,8,9]. Attack scenarios are modeled to represent system attack behavior representing states, conditions, and transitions required to characterize the attacks. The attack scenarios are made executable by developing a suitable attack signature generator. An attack signature includes necessary specifications using states and transitions which are directly executable against the system events for a particular attack. The framework provides an attack test driver which generates attack signatures and tests system attack specific behavior with respect to the modeled attack scenarios. The attack test driver automatically generates attack test vectors, *i.e.*, system events. The system events are converted to attack test driver specific events before being tested against the attack scenarios. The attack test driver uses an attack testing engine which employs a generic attack testing algorithm applicable for various target systems. The framework is evaluated, and experimental results show the efficacy of the framework in testing wide range of attacks.

The overview of the attack testing framework is provided in the next section. The details of the attack testing process is presented in Section 3. Section 4 presents the implementation and experiments. The related work are discussed in Section 5. Section 6 summarizes this work and future research directions.

2 Attack Testing Framework Overview

Figure 1 presents the proposed model-based attack scenario testing framework. Attack scenarios are modeled in extended abstract machines (ASMs), where states are instrumented with specific attack attributes. ASMs incorporate attack variables in the state machines [6,8,9]. The attack variables allow more specific descriptions of system attributes corresponding to different attacks. An attack is modeled as a set of states and transitions. States represent a snapshot of different system attributes during the course of attacks. The transitions are labeled with system events that cause changes from one state to another. A state transition can take place only if certain conditions associated with the transition are satisfied. The system events need to take place in certain order to make an attack successful. Once the system reaches a state under attack, an attack report is generated (see example in Section 3.3).

The rest of the framework consists of three major modules: signature-base module, sensor module, and main module (see Section 3.1). The three modules form the architecture of the attack test driver. Signature-base module provides the executable attack test scenarios called attack signatures that are ready to be used for testing by the attack test driver. The attack signature generator is used to produce attack signatures from the modeled attack scenarios.

The sensor module generates system events for testing those against the modeled attack scenarios. The attack test vectors are generated automatically from the system events using the event generator. An attack scenario can have different representation formats based on the target system environment. Therefore, the system events have to be captured first in an appropriate format so that they can be tested against the modeled attack scenarios. The task of the attack schemas is to read system events and provide a

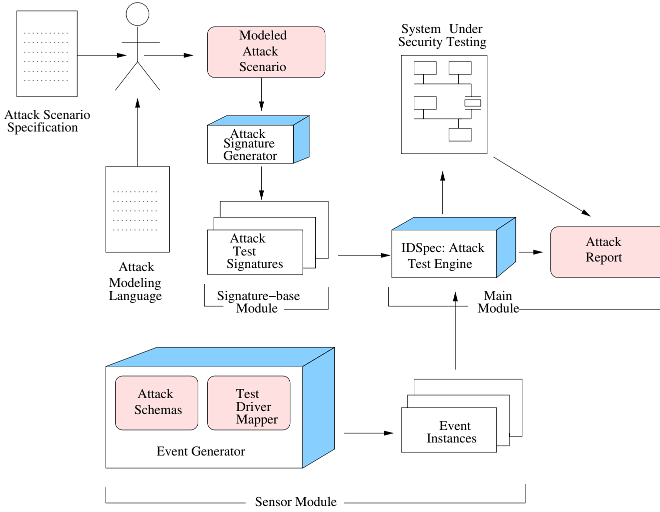

Fig. 1. Attack testing framework

way how they can used for testing. The test driver mapper converts the system events to the attack test driver specific events.

The main module contains an attack testing engine (called IDSpec) that in general requires two types of parameters: attack signatures and test driver specific system events. IDSpec tests the system based on the modeled attack scenarios and generates a report when an attack is found.

3 Testing Attack Scenarios

In this section, the testing process is described in detail following the proposed framework. The attack test driver architecture is described in Section 3.1. The attack testing engine of the architecture employs the CAAT (Context-Aware Attack Testing) algorithm (see Section 3.2). The testing process is further illustrated using the DosNuke attack in Section 3.3.

3.1 Attack Test Driver Architecture

The attack test driver consists of three principal modules (see Figure 2): signature-base, sensor, and main. The modules are discussed in the following paragraphs.

Signature-Base Module. This module contains executable attack signatures that are used by IDSpec to match the captured events with the signatures and to test potential attacks. Based on the security requirements, high-level descriptions of attack scenarios are developed. The attack scenarios are then modeled in ASMs. The attack signature generator implemented within this framework produces executable attack signature plug-ins from the modeled attack scenarios. During the course of execution, the plug-ins are loaded in the knowledge base of the attack test driver.

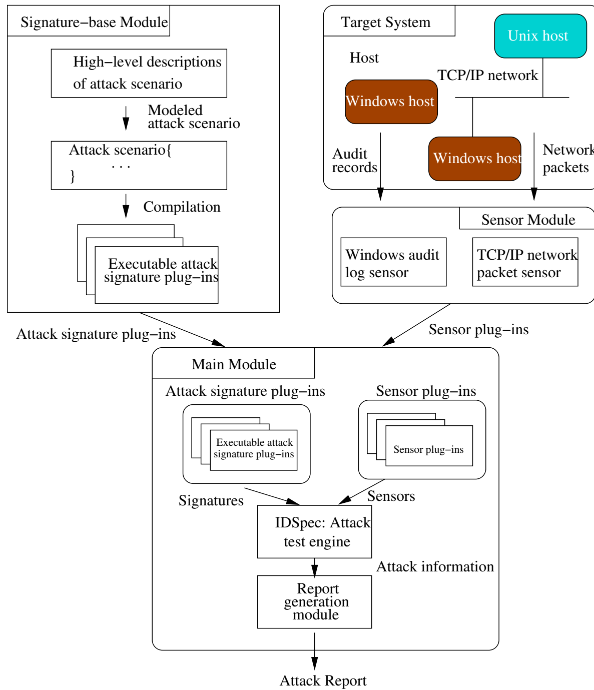

Fig. 2. Attack test driver architecture

Sensor Module. The attack test driver analyzes the events that take place in the system and identify ongoing attacks. It is assumed that attacks will leave a trace in the system activity logs. The attack signatures are written based on these events. Each log has its own format (like Windows security log and tcpdump log files). Therefore, the primary task is to read data from the event sources and convert those to the test driver specific form that can be easily analyzed by IDSpec. Figure 2 shows the target system considered in the testing process. The events from windows host are considered as audit records, while the events from TCP/IP network are regarded as network packets. However, the framework is designed in such a way so that it can incorporate other types of data sources (like Solaris BSM audit data) in its sensor module.

Main Module. The attack test driver collects events representing ongoing system activities from the sensor module. IDSpec analyzes the event streams and identifies whether there is an attack in progress. For this purpose, IDSpec matches the description of an executable attack scenario against the stream of events. Once an attack has been detected, the report generation module notifies the administrator. The notification consists of information having the time and date of an attack, the source of the attack, detailed testing information regarding the attack, and the effects it has on the system under test. IDSpec uses a generic attack testing algorithm, CAAT, presented in the following section.

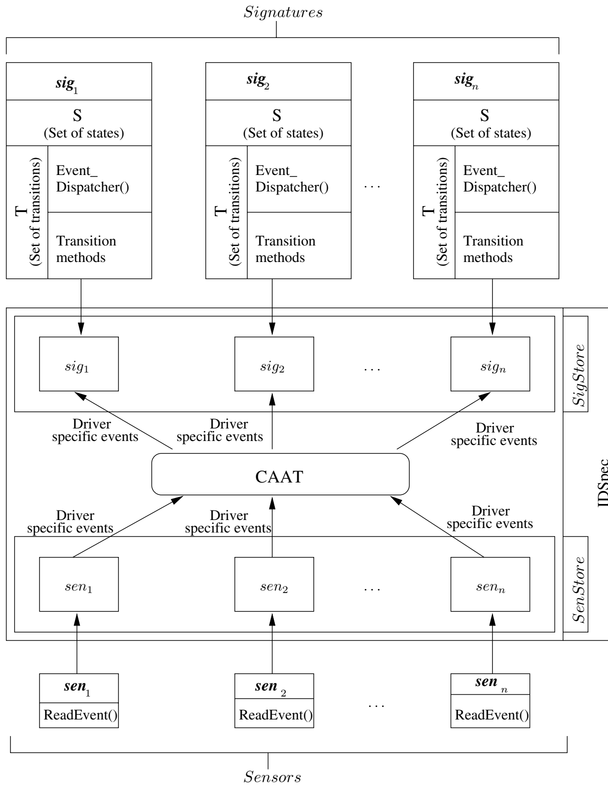

Fig. 3. Attack test driver

3.2 Context-Aware Attack Testing

The CAAT algorithm is provided in Listing 1. The algorithm takes as input a set of *n* attack signatures defined by *Signatures*= $\{sig_1, sig_2, \ldots, sig_n\}$. The signature plug-ins are provided by the signature-base module. Here, each sig_i represents a particular attack signature. During the course of execution of the attack test driver, each of the attack signature plug-ins are loaded in a global storage space denoted by *SigStore* located inside IDSpec. The second parameter of the algorithm is a set of *m* system events, *E*= $\{e_1, e_2, \ldots, e_m\}$, which are collected by the sensor modules. Here, each e_i represents a particular system event. Let *Sensors* be the set of *p* sensor plug-ins, *Sensors*= $\{sen_1, sen_2, \ldots, sen_p\}$, which capture events from the target system and convert them to test driver specific event format as expected by IDSpec. This set forms the third parameter of the CAAT algorithm. *SenStore* is a global storage space, where all the sensor plug-ins from the set *Sensors* are instantiated and loaded during the initialization phase of the attack test driver. IDSpec employs the algorithm, CAAT, matches the signatures from $SigStore$ against the driver specific system events from $SenStore$ to test any ongoing attack in the system. $SigStore$, $SenStore$, and the algorithm execution body of CAAT form IDSpec.

The following paragraphs provide the details of the algorithm by referring to the line numbers of Listing 1, while Figure 3 demonstrates the functionality of the test driver. The algorithm first initializes the signature storage *SigStore* and the test driver specific system event storage, *SenStore*. In the beginning, both sets are empty (Lines 01-02), and then *SigStore* is initialized by loading each of the attack signatures from the set *Signatures* (Lines 03-05), and *SenStore* is initialized by loading the sensor plug-ins from the set *Sensors* (Lines 06-08).

Listing 1. CAAT: Attack testing algorithm

Input: A set of *n* attack signature plug-ins (*Signatures*), a set of *m* events (*E*), and a set of *p* sensors (*Sensors*)
Output: Tests whether the events from *E* takes the system from a safe state to a state under attack by matching the events in the attack steps defined in an attack signature. (*T* is set of transitions, *F* is state transition function, *C* is set of conditions, and *X* is set of actions).

```
00. CAAT (Signatures, E, Sensors)
01.     SigStore:= ∅
02.     SenStore:= ∅
03.     FOR EACH attack signature plug-in a ∈ Signatures DO
04.         SigStore:= SigStore ∪ a
05.     END FOR
06.     FOR EACH sensor plug-in s ∈ Sensors DO
07.         SenStore:= SenStore ∪ s
08.     END FOR
09.     WHILE TRUE DO
10.         FOR EACH sensor plug-in s ∈ SenStore DO
11.             EventInstance E_x := s.ReadEvent()
12.             IF E_x = NULL THEN CONTINUE
13.             FOR EACH signature plug-ins a ∈ SigStore
14.                 a.EventDispatcher (E_x)
15.             END FOR
16.         END FOR
17.     END WHILE
18.     EventDispatcher (EventInstance E_x)
19.     FOR EACH transition t ∈ this.T
20.         IF (Satisfies (E_x, t))
21.             FOR EACH action statement x ∈ t.X
22.                 Execute (x)
23.             END FOR
24.         END IF
25.     END FOR
26.     Satisfies (EventInstane E_x, Transition t) returns Boolean
27.     Boolean bResult:= FALSE
28.     State X:= GetSourceState (t.F)
29.     FOR EACH state instance x ∈ X DO
30.         IF t.C holds for E_x bResult:= TRUE
31.     END FOR
32.     RETURN bResult
```

The next part of the algorithm is responsible for collecting events and performing analysis on the event stream to test any potential attack attempts. Each sensor plug-in provides an interfacing method `ReadEvent()`. This method captures data from the data sources (*e*.g., Windows audit logs or TCP/IP networks), formats the data into test driver specific events, and returns the events to their callers. The `EventDispatcher()` method (Lines 13-14) matches each event in $SenStore$ against each of the signatures of $SigStore$. The details of the `EventDispatcher()` method are provided in Lines 18-25. When the `EventDispatcher()` method receives an event, it checks

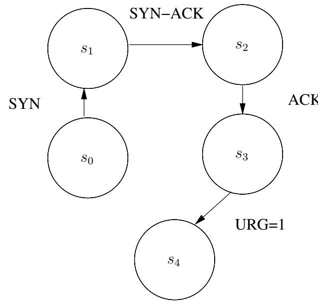

Fig. 4. State transition diagram of the DoSNuke attack

all the possible transitions of current signature instance that could be fired by the event. An event could fire a transition only if it satisfies the condition set of that transition (Lines 19-20). This is checked by the `Satisfies()` method shown in Lines 26-32. First, the method retrieves the source state of the transition by calling the method `GetSourceState()` with the state transition function *F* as a parameter. Then, the method checks if there exists any state variable instance in the source state that matches with event attributes specified in the condition constraint set for that transition (Lines 29-31). Depending on the positive outcome of the decision, the signature plug-in executes the set of actions associated with the transition (Lines 21-22). The set of action statements include updating state variables, making a transition to the new state, or generating an attack report in case of reaching the "state under attack". Otherwise, the current state remains unchanged.

3.3 The Testing Process Illustrated

We illustrate the testing process using the DosNuke attack. DoSNuke is a Denial of Service (DoS) attack that exploits a bug in the Windows NT operating system of a victim machine. At first, the attacker establishes a TCP connection to NETBIOS port (port number 139) and then sends a series of packets with URG bit set. The URG bit is set to represent out-of-band data (called "urgent data" in TCP) in a data stream. Figure 4 shows the state machine for the DoSNuke attack. Receiving a connection request packet (SYN packet) from the attacker changes system state from s_0 to s_1. When the receiving machine acknowledges the request with a SYN-ACK packet, the state changes from s_1 to s_2. Receiving acknowledgement from the attacker (ACK packet) establishes a TCP connection between the victim and the attacker and causes the state to transit from state s_2 to s_3. When the victim receives a TCP packet, destined to port 139, with URG bit set, it takes the system to a compromised state, (*i.e.,* s_4).

While translating the modeled attack scenario to executable attack signature plug-ins, the model is instrumented with necessary data structures as shown in Figure 5. In this figure, the generic attack scenario model *A* has three states: S_0, S_1, and S_2 with state variables *SourceIP*, *SourcePort*, *AttackerIP*, and *AttackerPort*. Moreover, *A* defines three transitions T_1, T_2, and T_3, each having the form of <*F,C,X*>. Each state is implemented as a list, storing attack scenario instances, to facilitate the testing of same type of attack taking place concurrently. Different values for state variables

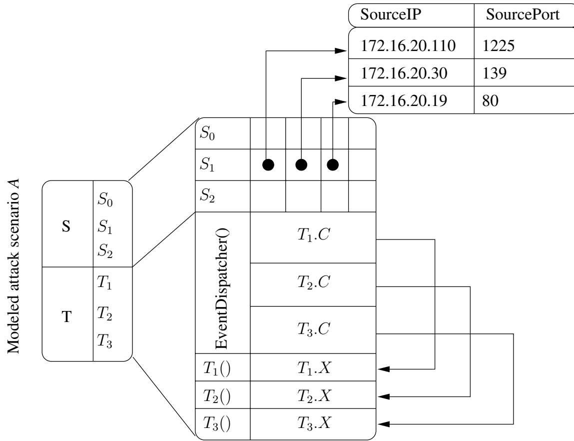

Fig. 5. Signature data structure

are stored in the list representing attack instances. For example, the three entries for S_1 (<172.16.20.110:1225>, <172.16.20.30:139>, and <172.16.20.19:80>) represent that three instances of attack type A are in progress.

Similarly, each transition presented in the modeled attack scenario is mapped to executable instructions in the signature plug-in. The condition part of each transition executes the `EventDispatcher()` method in attack signatures. As mentioned before, this method decides whether a captured system event is able to make changes in system states. Figure 5 shows that the condition parts of T_1 ($T_1.C$), T_2 ($T_2.C$), and T_3 ($T_3.C$) are merged in the `EventDispatcher()` method. The action part of each transition is mapped to a set of functions that are called by `EventDispatcher()` upon satisfying the condition set for that transition. For example, if an event satisfies $T_1.C$, then the function T_1 is called that executes the action statements corresponding to that transition (*i.e.,* $T_1.X$). Therefore, CAAT provides the flexibility to test for multiple attacks of the same kind executed at the same time by providing event matching capability to every attack signature in IDSpec. Upon the arrival of a particular system event specific to an attack scenario, the corresponding attack signature is executed by IDSpec. With the completion of an attack testing process, an attack report is generated. The attack test driver keeps track of different attack instances as it analyzes each system event with respect to the modeled attack scenarios. Figure 6 shows the DosNuke attack testing process by providing a simulation of two simultaneous DosNuke attacks against a victim machine.

Let the victim machine has IP address $P = 172.16.20.100$, and the attacker machines have IP addresses $X = 172.16.115.234$ and $Y = 172.16.115.20$. Let X and Y attempt to carry out the DosNuke attack against host P. The TCP/IP packets that are exchanged between these hosts are denoted as a tuple of the form <*SourceIP, SourcePort, Flag, DestIP, DestPort*>, where *SourceIP* and *SourcePort* denote the sender's IP address and port number respectively, while *DestIP* and *DestPort* denote the receiver's IP address and port number respectively. *Flag* represents the type of the network packet.

Attack states

Time

Event	S_0	S_1	S_2	S_3	S_4
<X, 1216, SYN, P, 139>		<X, 1216, P, 139>			
<P, 139, SYN−ACK X, 1216>			<X, 1216, P, 139>		
<Y, 1510, SYN, P, 139>		<Y, 1510, P, 139>	<X, 1216, P, 139>		
<P, 139, SYN−ACK, Y, 1510>			<X, 1216, P, 139> <Y, 1510, P, 139>		
<X, 1216, ACK, P, 139>			<Y, 1510, P, 139>	<X, 1216, P, 139>	
<Y, 1510, ACK P, 139>				<Y, 1510, P, 139> <X, 1216, P, 139>	
<X, 1216, URG bit set, P, 139>				<Y, 1510,	<X, 1216, P, 139> DosNuke Attack
<Y, 1510, URG bit set, P, 139>					<Y, 1510, P, 139> DosNuke Attack

Fig. 6. Testing for the DosNuke attack using CAAT

The DosNuke attack signature is executed when the corresponding system event is generated by the attack test driver. The first column of the table in Figure 6 represents system events related to DosNuke attack scenario. The rest of the columns simulates the different testing stages of the DosNuke attack showing successive states of the DosNuke attack signature. The different states represent different attack instances of the DosNuke attack. Moving from left to right of the table needs transitions from one state to the next state. A transition is fired upon the arrival of a corresponding system event necessary to satisfy the condition. System events in the upper rows are generated before the system events in the lower rows. For example, with the arrival of a SYN packet, transition from states s_0 to s_1 is performed by the `EventDispatcher()` method. A transition from states s_1 to s_2 is performed when the packet with SYN-ACK flag is generated. The system attributes are updated according to every state transition. The columns representing different states store respective system attributes related to the DosNuke attack scenario.

4 Implementation and Experiments

The three modules of the attack test driver (signature-base, sensor, and main) are implemented using C#.NET programming language. To specify attack scenarios, for the sake of widespread applications and the execution capability, a security extension of AsmL (Abstract State Machine Language) [8] called AsmLSec (Abstract State Machine Language for Security) [9] is used in this work. The attack signature generator

Table 1. Attack scenarios used in evaluating the framework

Attack Type	Attack Name	Short description
DoS	Land	Using network packets with same source and destination address
	DoSNuke	Using network packets with TCP URG bit set
	Teardrop	Using mis-fragmented UDP packets
	CrashIIS	Malformed HTTP request causes IIS server to crash
Probe	Queso	Using seven network packets with odd combination of TCP flags
R2L	Netcat	Using a trojan to create backdoor on victim machine
U2R	Sechole	Using DLL to add the user to administrator group
	Yaga	Hacking the registry adds the user to administrator group
	Anypw	Allows the attacker to logon to the system without a password
Data	NTFSDos	Allows the attacker access to NT partitions without authentication

implemented in this framework is an AsmLSec compiler. Flex [23] is used to generate the lexical analyzer unit, while Bison is used for generating the parser of the AsmLSec compiler. The output from the two phases are compiled and linked together using a C compiler. The compiler produces the AsmL representation from the modeled AsmLSec attack scenarios. The AsmL specification of the modeled attack scenarios is compiled using the AsmL compiler to generate the signature plug-ins in the form of a Dynamic Link Library (DLL).

Each event-capturing module for the sensor module is implemented as a shared library (Dynamic Link Libraries) written in *C#.NET* language. Two DLLs are implemented for the two event sensors: *WinLogPlugin.dll* for capturing Windows audit log events and *TCPIP.dll* for network packets. During the initialization phase of the attack test driver, it loads these plug-ins dynamically thus having the flexibility to add a new plug-in for another type of data source in future. Each plug-in provides a method, `ReadEvent()` that is invoked to fetch a captured event from the event generator according to the test driver specific event format. In case of *WinLogPlugin.dll*, the function returns a Windows audit log entry, *WinLogRecord*. Similarly, *TCPIP.dll* captures TCP/IP network packets and returns a record of type *FrameHeader* representing the captured ethernet frame.

The framework is evaluated for by modeling the following five most common categories of attack scenarios: *Denial of Service attacks (DoS)* are designed to disrupt a host or network service; *Remote to Local attacks (R2L)* let an attacker gain local access to a machine even though he or she does not have an account on that machine; *User to Root attacks (U2R)* allow a local user on a machine to gain administrative privileges; *Probe attacks* scan a network of hosts to discover information such as IP addresses, ports, and host operating system types; and *Data attacks* access to restricted files [7]. Table 1 presents the the attacks that are used to evaluate the framework. Experimental results show the effectiveness of the framework in testing those attacks against the target system.

5 Related Work

Blackburn *et al.* [4] propose a model-based approach to automate software security testing. The generated test vectors from the security specifications can be executed against Oracle and Interbase database servers. The security specification is written in SCR

(Software Cost Reduction) with SCRtool. SCR test specification is converted to T-VEC test specification using an SCR-to-T-VEC translator. A T-VEC tool is used to generate test vectors from T-VEC test specifications. Chandramouli and Blackburn [3,5] continue this model-based security testing approach by combining the security behavioral model and the test vectors with product interface specifications. The interface specification is provided using an object mapping file which maps between the behavioral model variables and the interface elements. The model-based testing approach in this paper tests a system attack behavior against a state-based formalism of the modeled attack scenarios. While their security testing processes use the SCR-to-T-VEC translator to translate the SCR specification into T-VEC test specification, the attack testing process of this work generates different system events and automatically converts them into attack-driver specific test vectors, *i.e.*, attack events.

Potter and McGraw [10] argue in favor of risk-based security testing which should be employed while the software is still under development. Software penetration testing technique plays a vital role in security testing, where the software is tested against all kinds of possible attacks and probing. Arkin *et al.* [11] propose that a penetration test must be structured according to perceived risk. Stytz and Banks [12] suggest an intelligent system that can test a software system while it is still in the development phase by presenting the basic concept of dynamic security testing. They argue that a software under development should be tested against all kinds of attacks. The risk-based testing, penetration-based testing, and dynamic security testing approaches have influenced the development of the attack testing framework provided in this paper. The framework can be employed early in the software development life cycle to test a system under development.

A security-critical system designed in UMLsec (Unified Modeling Language extension for security) can be tested for flaws automatically using effective tool support [13]. The UMLsec models have to be imported in an internal repository which is an XMI-specific data-binding library for the XML representation of an UML diagram. The access to this repository is provided by JMI (Java Metadata Interface) which can be used for static and dynamic checking of the model. For the dynamic analysis part, the UMLsec diagrams are translated into first-order logic formulas. Jürjens [14] provides a list of tools supporting model-based testing where the security properties are specified using UMLsec, and the model is verified automatically by a Prolog-based attack generator against the system. The modeled attack behavior in this work is tested against automatically generated system events. In this work, an automatic attack testing framework is provided where attack scenarios are modeled in state-based formalism. Executable attack signatures are generated from the modeled attack scenarios, and then they are tested against automatically generated system events.

Allen *et al.* [18] propose an architecture for testing the security of network protocol implementations. A protocol specification is converted into a finite state diagram. A valid state sequence is called a test template. Each test template accompanied with valid data is termed as a test case or message. Valid messages are separated into relevant blocks supported by protocol specifications and fuzzed to generate corrupted inputs to reveal vulnerabilities in applications. In contrast, our work uses attack signatures and matches attacks with incoming network packet sequences. Kosuga *et al.* [19] propose

an SQL (Standard Query Language) injection attack (SQLIA) testing framework named Sania for the application development and debugging phase. Their approach initially constructs parse trees of intended SQL queries written by developers. Terminal leafs of parse trees typically represent vulnerable spots, which are filled with possible attack strings. The difference between the initial parse tree and the modified parse tree generated from user supplied attack string results in warnings of SQLIAs. Salas *et al.* [20] generate test cases that reveal security bugs of functional specification written in Object Constrained Language (OCL). They perform testing of SQL injection attacks based on the specification of login functionalities for web applications by injecting faults in specifications. In contrast, our work tests attacks through AsmL specification. Similarly, Wimmel *et al.* [21] generate test cases by mutating specification of cryptographic protocol. The modification includes confusion of keys or secrets, missing or wrongly implemented verification of authentication codes, etc. The implementation of the protocol is tested based on the mutated specification. Jayaram [22] proposes testing the security of cryptographic protocol specified with UML state charts. The method generates initial test data sets that are adequate for control and data flow coverage criteria. The resultant test set is measured for adequacy with respect to security mutants which must be nullified by the generated test cases.

Tal *et al.* [15] propose vulnerability testing of frame-based network protocol implementation, where the structure of a protocol data unit (PDU) is specified in a frame. Their approach captures PDUs from client machines, mutates data fields of PDUs, sends them back to the server, and observes whether the protocol daemon running in the server crashes due to segmentation violation. Ghosh *et al.* [16] mutate the internal states of program to detect vulnerabilities at runtime. They develop Fault Injection Security Tool (FIST) which injects various types of faults such as corruption of boolean, integer, and string variables, overwriting the return addresses of stacks. Du *et al.* [17] perform vulnerability testing of applications by perturbing environment variables during runtime from initialization processes, file system inputs, network packets, etc. They propose fault coverage-based test adequacy criteria. Ideally, the higher the fault coverage, the more secure the application is.

6 Conclusions and Future Work

In this work, a framework is proposed which can test software for possible attacks with respect to modeled attack scenarios. The architecture of the attack test driver is presented by describing its different modules and their interactions. A generic attack testing algorithm called CAAT (Context-aware Attack Testing) is presented. The algorithm is employed by the attack test driver to test the target system with respect to the modeled attack scenarios. The modeling and testing of attack scenarios are explained using the DosNuke attack scenario as an example. The framework is evaluated by using the five categories of attacks: DoS, R2L, U2R, probe, and data attacks. The attack testing engine compares the attack signature plug-ins against automatically generated attack test vectors, *i.e.*, system events.

This work contributes to the automatic testing of attack behavior of a system, where the attack scenarios are modeled in a formalism based on extended abstract state

machines. The proposed attack testing framework can also be used to test the software under development with respect to potential attacks for discovering vulnerabilities early in the software development life cycle. The framework is applicable for various types of target systems and the most common attack scenarios. The attack testing algorithm, CAAT, provides a generalized approach to testing which greatly improves the applicability of the framework.

Attacks are of varying nature, and it is almost impossible to model and test all the attacks against a particular system using any attack modeling language and a framework. Most of the limitations and future research of this work are related to the current implementation of the attack test driver and the expressive power of the attack scenario modeling language. We will extend our work to cover more attack scenarios that the current implementation of the attack test driver fails to test. Some attacks may be carried out spanning over several login sessions or may be carried out after weeks. The attack test driver cannot keep track of such attacks and therefore fails to test system penetrations due to those attacks. Another type of attack that the driver cannot test is when the same attacker logs in with a different username and each time carries out one step of an attack. In future, AsmLSec grammar can be modified to express the varying nature of many attack scenarios. Because of the variations of the attacks in different systems and operating environments, it is not easy to measure the attack test coverage of the proposed attack testing framework. However, the framework can be extended to test more attacks.

Acknowledgment

This research work is partially funded by the Natural Sciences and Engineering Research Council (NSERC) of Canada. We would also like to thank Hossain Shahriar of Queen's University, Canada for his helpful comments to improve this paper.

References

1. Dalal, S., Jain, A., Karunanithi, N., Leaton, J., Lott, C., Patton, G., Horowitz, B.: Model-based testing in practice. In: Proc. of the Intl. Conf. on Software Engineering, USA, May 1999, pp. 285–294 (1999)
2. Rosaria, S., Robinson, H.: Applying models in your testing process. Information and Software technology 42(12), 815–824 (2000)
3. Chandramouli, R., Blackburn, M.: Automated testing of security functions using a combined model and interface-driven approach. In: Proc. of the 37th Annual Hawaii International Conference, Hawaii, USA (January 2004)
4. Blackburn, M., Busser, R., Nauman, A., Chandramouli, R.: Model-based approach to security test automation. In: Proc. of the 14th International Software and Internet Quality Week Conference, San Francisco, USA (June 2001)
5. Chandramouli, R., Blackburn, M.: Security functional testing using an interface-driven model-based test automation approach. In: Proc. of the 18th Computer Security Applications Conference, Las Vegas, USA (December 2002)

6. Barnett, M., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., Veanes, M.: Towards a tool environment for model-based testing with AsmL. In: Proc. of the 3rd International Workshop on Formal Approaches to Testing of Software, pp. 252–266. Springer, Heidelberg (2003)
7. MIT Lincoln Laboratory. DARPA Intrusion Detection Evaluation (2006), `http://www.ll.mit.edu/ist/ideval` (accessed in April 2006)
8. Barnett, M., Schulte, W.: The ABCs of specification: AsmL, behavior, and components. Informatic (Slovania) 25(4), 517–526 (2001)
9. Raihan, M., Zulkernine, M.: AsmLSec: An extension of abstract state machine language for attack scenario specification. In: Proc. of the 2nd International Conf. on Availability, Reliability and Security, Vienna, Austria (April 2007)
10. Potter, B., McGraw, G.: Software security testing. IEEE Software Security & Privacy Magazine 2(5), 81–85 (2004)
11. Arkin, B., Stender, S., McGraw, G.: Software penetration testing. IEEE Software Security & Privacy Magazine 3(1), 84–87 (2005)
12. Stytz, M., Banks, S.: Dynamic software security testing. IEEE Software Security & Privacy Magazine 4(3), 77–79 (2006)
13. Jürjens, J.: Sound methods and effective tools for model-based security engineering with UML. In: Proc. of the 27th International Conference on Software Engineering, St. Louis, USA, May 2005, pp. 322–331 (2005)
14. Jürjens, J., Fox, J.: Tools for model-based security engineering. In: Proc. of the 28th international conference on Software engineering, Shanghai, China, May 2006, pp. 819–822 (2006)
15. Tal, O., Knight, S., Dean, T.R.: Syntax-based Vulnerabilities Testing of Frame-based Network Protocols. In: Proc. of the 2nd Annual Conference on Privacy, Security and Trust, Fredericton, Canada, October 2004, pp. 155–160 (2004)
16. Ghosh, A.K., O'Connor, T., McGraw, G.: An automated approach for identifying potential vulnerabilities in software. In: IEEE Symp. on Security and Privacy, USA, pp. 104–114 (1998)
17. Du, W., Mathur, A.: Testing for software vulnerabilities using environment perturbation. In: Intl. Conf. on Dependable Systems and Networks, New York, USA, June 2000, pp. 603–612 (2000)
18. Allen, W., Chin, D., Marin, G.: A Model-based Approach to the Security Testing of Network Protocol Implementations. In: Proc. of the 31st IEEE Conference on Local Computer Networks, November 2006, pp. 1008–1015 (2006)
19. Kosuga, Y., Kono, K., Hanaoka, M., Hishiyama, M., Takahama, Y.: Sania: Syntactic and Semantic Analysis for Automated Testing against SQL Injection. In: Proc. of the 23rd Annual Computer Security Applications Conference, Miami, December 2007, pp. 107–117 (2007)
20. Salas, P., Krishnan, P., Ross, K.J.: Model-Based Security Vulnerability Testing. In: Proc. of Australian Software Engineering Conference, Melbourne, Australia, pp. 284–296 (2007)
21. Wimmel, G., Jürjens, J.: Specification-based Test Generation for Security-Critical Systems Using Mutations. In: George, C.W., Miao, H. (eds.) ICFEM 2002. LNCS, vol. 2495, pp. 471–482. Springer, Heidelberg (2002)
22. Jayaram, K.R.: Identifying and Testing for Insecure Paths in Cryptographic Protocol Implementations. In: Proc. of the 30th Annual International Computer Software and Applications Conference, Chicago, USA, September 2006, pp. 368–369 (2006)
23. Aaby, A.: Compiler Construction using Flex and Bison, `http://cs.wwc.edu/` (Accessed, April 2006)