Security Test Generation using Threat Trees

Aaron Marback, Hyunsook Do, Ke He, Samuel Kondamarri, Dianxiang Xu

Department of Computer Science
North Dakota State University
Fargo, ND 58108, USA
{aaron.marback, hyunsook.do, ke.he, samuel.kondamarri, dianxiang.xu}@ndsu.edu

Abstract

Software security issues have been a major concern to the cyberspace community, so a great deal of research on security testing has been performed, and various security testing techniques have been developed. Most of these techniques, however, have focused on testing software systems after their implementation is completed. To build secure and dependable software systems in a cost-effective way, however, it is necessary to put more effort upfront during the software development life cycle. In this paper, we provided a security testing approach that derives test cases from design-level artifacts. The security testing approach we consider consists of four activities: building threat trees from threat modeling; generating security tests from threat trees; generating test inputs including valid and invalid inputs; and assigning input values to parameters. We also conducted an empirical study to show feasibility of our approach.

1 Introduction

Software security issues have been a major concern to the cyberspace community. These issues include denial of service attacks, attacks that corrupt data, and attacks that lead to disclosure of confidential information, such as sensitive financial and medical data. While the importance of trustworthy software systems has been well recognized and tremendous effort has been devoted to enhancing cyber security, companies have still suffered from various cyber crimes [17]. For instance, one recent study performed by the WebCohort's Application Defense Center from 2000 to 2004 reported that at least 92% of web applications are vulnerable to some form of hacker attacks [22].

It has been well-recognized that security issues should be considered as early as possible during the software development life cycle. In particular, secure design is critical to the success of secure software development because design-level vulnerabilities are among the hardest defects to handle [9, 16]. Design for security (e.g., design-level threat modeling) has recently become a widely accepted practice for building dependable software systems in a cost-effective way. However, security design does not guarantee secure implementation as vulnerabilities can be introduced in the implementation process. Using the artifacts of security design (e.g., threat models) is highly desirable for testing the resultant implementation.

To derive security tests from design-level artifacts, there are two challenges involved. The first challenge involves automated test generation from secure design. Security attacks typically result from malicious intentions or invalid inputs deliberately chosen by attackers. The complexities of input space and program structure make the testing of a program against all invalid inputs extremely difficult. Therefore, it is important to automate or partially automate security testing.

The second challenge involves generating executable test code. Because software design is independent of implementation decisions, security tests generated from design-level artifacts are not immediately executable. To transform design/model-level security tests to implementation-level test code, several issues need to be addressed.

First, model-level elements (e.g., events, states, and constraints in threat trees, and security tests) must be mapped to implementation-level constructs (e.g., classes and methods in an object-oriented program) to make security tests executable. Second, after the events in model-level security tests are mapped to method invocations, specific values need to be assigned to the arguments of each method invocation. Third, the expected results of each model-level security test also need to be converted into a sequence of

AST'09, May 18–19, 2009, Vancouver, Canada
978-1-4244-3711-5/09/$25.00 © 2009 IEEE

executable assertions at the implementation level.

To address these challenges and demonstrate feasibility of our approach, we developed security testing techniques and performed a case study using a real-world web application. The security testing techniques we consider in this paper consist of the following activities: (1) building threat trees; (2) generating security tests from threat trees; (3) generating test inputs including valid and invalid inputs; and (4) assigning input values to parameters.

The results of our experiment showed that our approach is effective in exposing vulnerabilities that have been introduced to the resultant implementation. Our results also showed feasibility of our approach to automatic security test generation.

The rest of the paper is organized as follows. Section 2 discusses background information and related work relevant to security testing techniques. Section 3 presents an overview of threat modeling. Sections 4 presents test generation techniques and their implementations. Section 5 presents our study design, results, and discussion of our results. Section 6 presents conclusions and discusses possible future work.

2 Background and Related Work

Software testing for security aims at finding security vulnerabilities by executing the software with test cases [20]. While both valid inputs and invalid inputs are required for testing any software, invalid inputs are more important to security testing because by nature they verify whether or not the system is misused (e.g., whether access is granted to unauthorized users), or provides unexpected functionalities. This feature makes security testing more difficult because there are too many possible invalid inputs. We also need to test the "presence of an intelligent adversary bent on breaking the system" [16].

Threat modeling has emerged as a viable practice for secure software development. Threat modeling is the process of producing a simplified, abstract description of how an adversary would perform potential attacks or pose security threats to the system. It can be conducted at various levels of abstraction and granularity or in different development phases, such as requirements analysis, design, implementation, and even testing [23].

Several notations have been proposed for threat modeling, such as threat trees (a variation of fault trees for safety analysis) [19], threat nets (based on Petri nets) [15, 24], misuse cases (based on use case modeling) [4, 18]. Obviously, threat models can be used to generate security tests for exercising whether the implementation is resistant from the identified security threats.

For example, Wang et al. [21] have proposed a threat model-driven security testing approach for detecting undesirable threat behavior at runtime. In their approach, threats to security policies are modeled with UML sequence diagrams. A set of threat traces is extracted from a design-level threat model. Each threat trace is an event sequence that should not occur during the system execution; otherwise it becomes a security attack. Different from this work, this paper is based on Microsoft's threat modeling approach [19], where an application is decomposed with data flow diagrams and security threats to the application are modeled with threat trees. The goal is to automatically generate security tests from threat trees.

Martin and Xie [10, 12, 11] have been investigating techniques for test generation from access control policy specifications written in XACML (OASIS eXtensible Access Control Marckup Language). They have defined policy coverage criteria [10] and a mutation testing framework for XACML access control policies [12]. To generate tests from policy specifications, they synthesize inputs to a change-impact analysis tool [11]. Masood et al. [13, 14] have investigated a state-based approach to test generation for role based access control (RBAC) policy. Their approach first constructs a finite state model of the RBAC policy and then drives tests from the state model. In contrast, in this paper, we investigate test generation from threat trees.

3 Secure Design using Threat Modeling

In this section, we describe a general approach to threat modeling to provide an overview of how threat modeling works, and then describe threat trees.

In general, threat modeling [8, 19] consists of the following four steps. First, we decompose the application under development in terms of data flow diagrams (DFDs) and identify the components of the application. An application can be decomposed into subsystems, and subsystems can further be decomposed into lower-level subsystems. Note that application decomposition is not to determine how everything works, but rather to identify the system boundaries, components, or assets, of the application and how data flow between the components. Second, we use the components/assets identified in the decomposition process as threat targets, identify the threats according to the STRIDE category [8], and model the threats using threat trees. STRIDE stands for Spoofing identity, Tampering with data, Repudiation, Information disclosure, Denial of service, and Elevation of privilege. The threat tree describes the decision-making process an attacker would go through to compromise the component. Third, we evaluate the risks of the threats using the DREAD method [8] and rank the threats by descending security risks. The DREAD method calculates the security risk of a threat as the average of the numeric values assigned to each of the following five categories: Damage potential, Reproducibility, Exploitabil-

ity, Affected users and Discoverability. Finally, we determine how to respond to the threats and choose appropriate techniques to mitigate the threats.

Based on the threat modeling process that we just described, we followed the first two steps to build threat trees, which serve as inputs for security test generation. Figure 1 shows a portion of the DFD diagram for the osCommerce system (version 2.2ms2) [1] used in our empirical study. More information about osCommerce can be found in section 5. The customer is the main external entity who provides inputs to the system. For the *Login* process, the input is username and password and the output is the acknowledgment that takes the customer to the home page. The *Shopping cart* process holds all items together until the checkout process is ready. Thus, the inputs include addition/deletion of items, and also updating the contents of the shopping cart whenever changes are made to it. The output is the contents of the cart. The *Checkout* process has the cart contents as its input, and the total price including taxes and shipping charges as its output. The *Payment* process can be done in two ways: (1) paying by credit cards; (2) paying cash on delivery. Its input includes the total price and credit card information and the output is detailed information on the requested order including the shipping/billing addresses, charges, and the cart contents. The credit card details are then forwarded to the credit card company or a third party for verification and then the approval statement is sent to the shopping cart company. The *Confirmation* process sends an email to the customer soon after the order is confirmed.

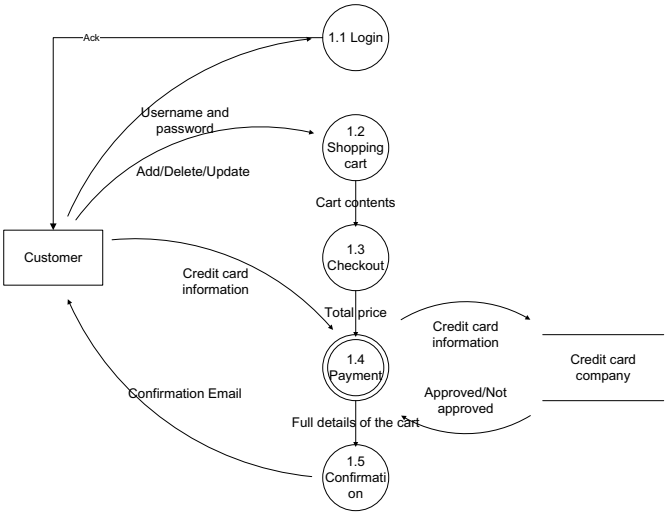

Figure 1. A Portion of the Data Flow Diagram for osCommerce

After building DFD diagrams, we proceed to build threat trees using STRIDE method as we described in the second step of threat modeling. Threat trees [6, 8, 19] have been used as the notation of threat models in order to decompose a high level threat into intermediate objectives and finally to individual attack actions. Threat trees are tree structures with child nodes having AND or OR relationships. The root node of a threat tree is the threat goal. The root node is then decomposed into sub-goals and the sub-goals are further decomposed until leaf nodes representing the individual attack actions are determined. A node of threat trees can be decomposed either as (1) a set of sub-goals, all of which must be achieved for the parent goal to succeed, that are represented as an AND-relationship, or (2) a set of sub-goals, any one of which must be achieved for the parent goal to succeed, that are represented as an OR-relationship. Each threat tree enumerates and elaborates the ways that an attacker could compromise the software. Each path through a threat tree represents a unique attack approach on the software. Different paths form different sequences of attack action that the attacker could achieve the threat goal of a threat tree. These paths are the templates of our test sequences.

Through the threat modeling process, we identified five threat trees for the osCommerce application. Among these, we present three threat trees that show existing threats in the osCommerce application. The first one is cross-site scripting. The input passed to the "page" parameter in multiple files and to the "zpage" parameter in geo_zones.php file of admin directory is not properly sanitized before being returned to the user. This can be exploited to execute arbitrary HTML and script code in an administrator's browser session in context of an affected site.The threat tree with four attack approaches to this threat is shown in Figure 2. Attackers try to obtain a user's *Session ID* by script code of *Hidden Image*, *Form*, or *Float DIV or IFrame*.

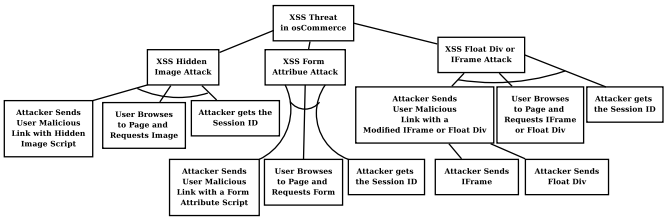

Figure 2. Threat tree of exploiting cross site scripting threat

The second threat is an SQL injection problem. The variables of $option and $value are taken from $this→contents in the shopping cart class via the $cart→get_products() function call in the script shopping_cart.php. Unfortunately these shopping cart values are taken from session data that is not properly escaped and can be controlled by an attacker via the id[] array when adding a product to the cart. The threat tree with three attack approaches to this threat is shown in Figure 3. Attackers try to obtain user's *Credit*

Card Information, *Personal Information*, or *Login Information* by injecting corresponding SQL scripts into the *id array*.

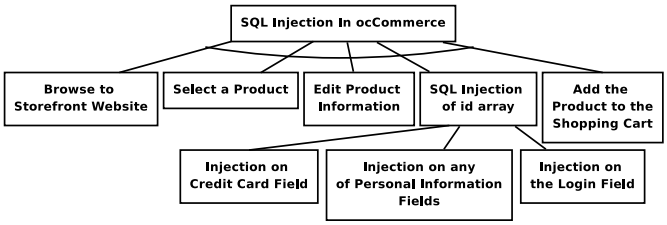

Figure 3. Threat tree of exploiting SQL injection threat

The third threat is an information disclosure issue.Lack of limiting the number of attempts of user name and password in the login process, attackers can guess user name and password, and then login and change user's account. The threat tree with four attack approaches to this vulnerability is shown in Figure 4. Attackers try to obtain *User Name* or *Password* by *Brute Force Attack* or *Dictionary Attack*.

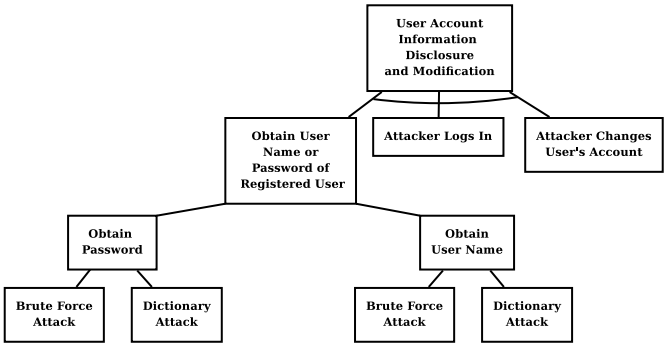

Figure 4. Threat tree of exploiting information disclosure threat

4 Security Test Generation

Our approach tries to bridge the gap between threat trees and executable test cases. To achieve this, we have developed a technique that: (1) analyzes threat trees created by the Microsoft Threat Modeling Tool [19]; (2) places the relevant data into a tree data structure; (3) walks this tree data structure to produce valid test sequences; (4) provides a mechanism to map actions in the test sequences to the applicable tests; (5) produces an executable test script for every sequence; (6) provides an input specification for every test script; and (7) generates test inputs including valid and invalid inputs.

Figure 5 shows how these activities are related. First, we generate security test sequences from threat trees(Section 4.1) and test parameters from input syntax(Section 4.2), respectively. We then integrate test parameters into the test sequences to create complete model-level security test cases. Based on the implementation knowledge, we further convert the model-level security tests into implementation-level security tests. The implementation-level tests can be executed together with the system under test. To illustrate our approach to test generation that is described in this section and the following section, we use the SQL Injection threat example (Figure 3) shown in Section 3.

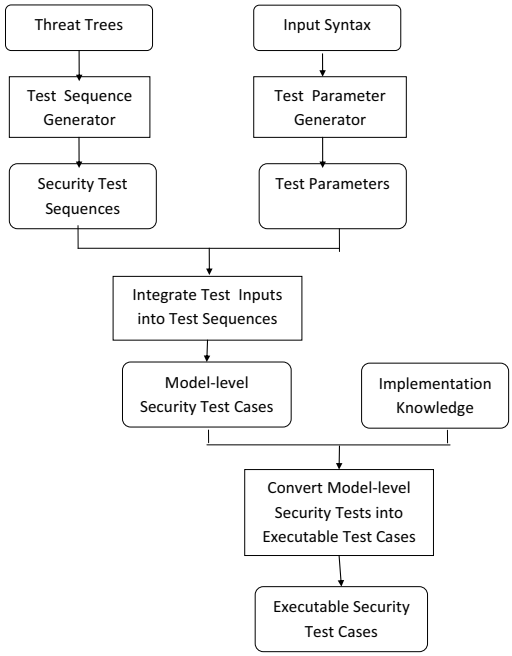

Figure 5. The Process of Security Test Generation

4.1 Security Test Sequence Generation from Threat Trees

For the first two tasks described above, we analyzed the ".tmd" files produced by the Microsoft Threat Model Tool. Since the .tmd files use XML version 1.0 syntax, they can be easily parsed using readily publicly available libraries. Using the Java DocumentBuilderFactory class, we created a series of and/or tree based data structures, and inserted the threat tree data from the threat trees into these trees; our implementation provides a way to add parameter information to the applicable leaf nodes in threat trees. For example, in our SQL Injection example, the "product_id" is required by the "Select a Product" node.

Once the relevant threat tree data is placed in our and/or tree data structure, we can generate test sequences by analyzing this tree. We used a depth-first approach for test sequence generations. The main goal of this step is to gen-

erate every sequence of events that may exploit the vulnerabilities of a system. To obtain these threat sequences, we used a node replacement algorithm where parent nodes are replaced by the sequence of their children. In the case where the child nodes are related by an AND relationship, the parent node is replaced by a sequence of the child nodes. The order of the nodes in this sequence is determined from the child nodes' left-to-right placement in the threat-tree. In the case where the child nodes are related by an OR relationship, we make n-1 (n is the number of child nodes) copies of the sequence containing the parent node. Then we replace the parent node in the original and copy sequences with each of the child nodes.

This algorithm generates three test sequences from our SQL Injection example (Figure3): (1) Browse to Storefront Website → Select a Product → Edit Product Information → *Injection on the Credit Card Field* → Add the Product to the Shopping Cart; (2) Browse to Storefront Website → Select a Product → Edit Product Information → *Injection on any of the Personal Information Fields* → Add the Product to the Shopping Cart; and (3) Browse to Storefront Website → Select a Product → Edit Product Information → *Injection on the Login Field* → Add the Product to the Shopping Cart.

Since the test sequences have been generated from design-level threat trees, we need to transform the test sequences into executable tests. In a typical threat tree, there may be actions that can be incredibly difficult or simply impossible to transform into executable test cases, such as social engineering attacks, hardware tampering, or distributed attacks. To resolve this problem, our technique allows a test case developer to bind only applicable and necessary events in the sequences to the executable test code. In our SQL Injection example, the event "Browse to the Storefront Website" is an unnecessary action because we can select the product directly using the product_id (http://example.com/product_info.php?products_id=2).
Also the event "Edit Product Information" cannot be performed separately from the "SQL Injection of the ID array" event with our test language. Such cases are not required to be transformed into the executable test code.

As mentioned earlier, many actions can be associated with parameters. Since a test sequence can have many events each with many parameters, keeping track of the order of all of the parameters associated with all of the actions in a sequence can be challenging. To address this challenge, we are required to follow several steps. First, we specify parameters in the Microsoft Threat Modeling Tool. These parameters are imported into to our tree structure during the parsing phase of our technique. Once we have completed the test sequence generation, we create a list of parameters by iterating through the events in a sequence. This list is then generated in the form of a comment line at the beginning of every executable test script. These parameters must then be provided to the test case in the form of command line parameters. This forces the test developer to take precautions when developing tests because these tests must contain the correct number of parameters. In our SQL injection example, the "Select a Product" event contains two parameters (product_id and options_id).

4.2 Syntax-based Test Input Generation

Once we have obtained test cases with parameters, we need actual inputs to complete test cases. To generate test inputs, we adopt a syntax-based testing approach, which is a grammar-based and input data-driven testing [5, 7].

As an illustration, consider the SQL Injection example that requires a product_id. The simplest syntax for the product_id is the regular expression a^* where a is any printable ASCII character. The cardinality of this set is enormous; therefore the syntax cannot be used in this form. Length constraints can be applied to the inputs to reduce number of total input. The total number of inputs after specifying a length constraint may still be too large. In this case, a length constraint of L will produce 95^L (where 95 is the number of printable ASCII characters) test inputs. To reduce the number of inputs down even further we use equivalence partitioning and boundary value analysis to group our inputs. In the most basic of cases inputs will partitioned into four categories:

1. Valid Inputs: These inputs meet all the design requirements and allow the program to function properly.
2. Handled Invalid Inputs: These inputs are invalid but the program should use some type of error handling to handle them.
3. Unhandled Invalid Inputs that do not pose a security risk: These are inputs of interest because they do reveal a flaw in the software, but are not the primary focus of this research.
4. Unhandled Invalid Inputs that do pose a security threat: These may cause security vulnerability. This is the primary focus of the input generation portion of this research.

To perform the boundary value analysis we must review design documentation for the application under test to retrieve the boundaries of the partitions. In our SQL Injection example, a review of the design documentation can further limit the product_id's to signed integers that are four bytes in length.

The last and final technique that is used to develop inputs is the use of a database to store common valid or invalid strings. A database also gives us the ability to combine certain parameters may be dependent on others. In the case of product_id, we already have a list of which inputs should be accepted by the application. Some product_id also have

a corresponding option_id's that are specific to that product_id (e.g. A video card could be equipped with 128MB or 256MB of RAM). A list of acceptable parameters for the video card (product_id = 2) example with multiple RAM configurations (128MB:options_id = 0; 256MB:options_id = 1) would simply be the set {(2,0) (2,1)}.

For testing purpose, we will execute every relevant test case with all of the valid product_ids and their corresponding option_ids combined with an invalid SQL injection string. The benefit of using a database is that it can be used to store common invalid data strings such as SQL injection strings and cross site script injection strings.

4.3 Implementation

To provide techniques that we described in Section 4, we have developed tools that allow us to analyze threats trees and generates security test cases. The following subsections provide further details on each of these tools.

Test Generation Tools

We have developed a prototype tool that implements our threat tree driven testing technique. We have implemented the following three modules using the Java programming language.

Parser and Sequence Generator: This module opens a .tmd file, parses it and places the data obtained from it into a tree structure. The tool then generates all of the event sequences that can be derived from the tree structure.

Test Assignment: This module allows a user to assign tests to all of the events in the every sequence. To achieve this we have implemented a graphical interface that displays the events to a tester. The tester can then use a file browser to assign specific tests to every event. We have also allowed the user to save and reload a test assignment session because this can be a very time consuming process.

Executable Test Generation: This module combines the assigned test files into executable test cases that correspond to our event sequences.

Test Cases

In this study we use the Selenium IDE Firefox extension and the Selenium Remote Control[3] testing environment to create and execute the majority of tests. For tests that were too complicated to be executed with selenium, we used the WWW::Mechanize and LWP [2] Perl libraries. All tests in this study were written in the Perl programming language.

For example, consider the first sequence in a shopping cart application that tries to expose SQL Injection vulnerabilities. We would need our test case to: (1) Select the Product, (2) Perform SQL Injection Credit Card field, and (3)Add the product to the shopping cart (Note that the "Browse to Storefront" and "Edit Product Information" events were not included because we consider them unnecessary.).

To accomplish this we must develop a test for each of these events. The test code for the Select the Product, Perform SQL Injection Credit Card field, and Add the product to the shopping cart events are shown in Figures 6, 7, and 8 respectively. For the final executable test sequence, we just combine all of these tests as shown in Figure 9.

```
use WWW::Mechanize;
use HTTP::Cookies;
use WWW::Mechanize::FormFiller;
use URI::URL;
my $agent = WWW::Mechanize->new(autocheck => 1);
$agent->cookie_jar(HTTP::Cookies->new);
$myargs = 0;
$agent->get('http://example.com/product_info.php?
products_id='."$ARGV[$myargs]");
$myargs = $myargs + 1;
```

Figure 6. Select Product Test Segment

```
$agent->field("$ARGV[$myargs]", '99\' UNION SELECT null,CONCAT('cc_number'), null,null FROM customers');
$myargs = $myargs + 1;
```

Figure 7. Perform SQL Injection Credit Card Field Test Segment

```
$agent->click();
```

Figure 8. Add the Product to the Shopping Cart Test Segment

Test Inputs

As shown in figure 9, we provide inputs to our executable test case in the form of command line arguments. Since this test case requires the user to select a product and the product's options , we can only use valid product-option combinations. This data can be obtained by viewing the options in the application's database. For all tests in our study, we were able to develop a relevant input list using the design documentation and our database.

5 Empirical Study

To demonstrate feasibility of our approach and to see our techniques are effective in exposing security vulnerabilities, we have designed and implemented an experiment. The following subsections present our object of analysis, experiment setup, and experimental results.

```
#usr/bin/perl -w
#p0:username p1:product_id p2:options_id
use WWW::Mechanize;
use HTTP::Cookies;
use WWW::Mechanize::FormFiller;
use URI::URL;
my $agent = WWW::Mechanize->new(autocheck => 1);
$agent->cookie_jar(HTTP::Cookies->new);
$myargs = 0;
$agent->get('http://example.com/product_info.php?
products_id='."$ARGV[$myargs]");
$myargs = $myargs + 1;
$agent->field("$ARGV[$myargs]", '99\' UNION SE-
LECT null,CONCAT('cc_number'), null,null FROM cus-
tomers');
$myargs = $myargs + 1; $agent->click();
```

Figure 9. Combined Executable Test Case

Object of Analysis

For this experiment, we used a web application, osCommerce [1]. OsCommerce is a web based storefront and shopping cart application. The application is written in the php programming language and uses the MySQL database system. We chose osCommerce for two reasons: (1) it is an open source; (2) it is a popular choice for retailers that want to deploy a shopping cart on their own server. We have developed a threat model that takes into account five different threats: (1) a data modification threat; (2) a user credential disclosure threat; (3) an SQL injection threat; (4) a cross-site scripting threat; (5) a multiple login attempts threat. Three of these (2, 3, and 4) are described in Section 3 in detail. Using our tool, we generated 38 test sequences and 2006 test cases equipped with test inputs.

Experimental Environment

We performed our experiment using two different virtual machines (one as the client, one as the server) on the same host. The client operating system was Xubuntu Linux version 8.04.1 and the server operating system was Suse Linux version 9.1. The server ran Apache as its HTTP server and MySQL as its database backend. One important thing to note is that the network connection has a direct impact on the execution time required for testing. In this experiment virtual machines on the same host are used to provide a nearly optimal network connection.

Data and Analysis

Our goal is to show that test cases generated from our approach would either expose a vulnerability or lack thereof. Table 1 shows the results of our test case execution. The table lists, for each of the five threats ("Threats"), the number of test sequences ("Sequences"), the number of tests executed for each sequence ("Tests per Sequences"), the total number of tests for each threat ("Tot. Tests"), and the number of tests that exposed vulnerabilities ("Tests Exposed Vulner."). To address the issue of false positives (tests that reveal non-existent vulnerabilities), we developed test sequences from threats that we knew the application would properly handle. Out of the five threats that we tested, Threats 1 and 2 were designed from attack patterns in the application that had been properly handled and threats 3, 4, and 5 were designed from known vulnerabilities. Thus, in the cases of Threats 1 and 2, the results showed that no tests exposed vulnerabilities (the fifth column shows "0" for these cases which indicates the absence of false positives). In the cases of threats 3, 4, and 5, however, the results showed that all tests exposed vulnerabilities, which means that vulnerabilities existed and were discovered through security testing.

Table 1. Effectiveness of Test Results

Threats	Sequences	Tests per Sequences	Tot. Tests	Tests Exposed Vulner.
1	18	56	1008	0
2	9	56	504	0
3	4	33	132	132
4	3	46	138	138
5	4	56	224	224
Totals	38	247	2006	494

We also measured the time required to develop and execute these test cases and the number of executable steps in every sequence in order to gauge how time efficient this testing technique was at completing test cases. There were a total of 256 executable events that were mapped to test segments in our study. It took a total 7379 seconds to map the executable events to the test segments. On average it took 1475.8 seconds to map each threat, 194.18 seconds to map each test sequence, and 28.82 seconds to map each event. It took a total time of 2032 seconds to run our 2006 individual tests. On average each test took .97 seconds to execute and .15 seconds per test segment.

Discussion

The results from our study show that this can be an effective test case development technique for discovering unmitigated threats. The lack of false positives and negatives also shows that this can be an effective security testing and verification technique. In particular, through the threat trees-based approach, we could identify the threat goals that attackers want to realize, and these threat goals are the targets of our security testing. We need to test whether these threat

goals will be realized or not. We also identified how attackers realize the threat goals. Different sequences of attack actions can be derived from the threat trees. These sequences of attack actions represent the ways in which attackers realize the threat goals and thus provide a basis for security test sequences.

Although the results for this empirical study are promising, there are some limitations in our study. One limitation involves how the threat trees were developed. The lack of comprehensive design documentation for our object program forced us to create threat trees from known vulnerabilities instead of design documents. Since this technique requires developers to build threat trees and the accompanying tests, the effectiveness may be directly affected by the ability of developers in developing tests and threat trees. Another limitation of our study is the lack of variety in object programs. We tried to minimize this limitation by performing realistic attacks against a large, readily deployed web application.

6 Conclusions and Future Work

In this paper we have developed threat model based security test generation techniques and conducted a study to assess our approach. Our results showed that our approach is effective in exposing vulnerabilities that have been introduced into the software system. In addition to these promising results, our approach also provides several significant advantages for security testing: (1) we could identify threat goals that attackers want to realize using threat trees; (2) we generated executable security tests automatically from threat trees considering actual inputs; (3) we could confirm that our object program was safe from some vulnerabilities - this indicates that these vulnerabilities were considered and thus handled in design-level.

For future work, first, we intend to perform additional studies using several web applications. Second, in this study we utilized small number of threat trees, so we intend to identify additional threat trees. Third, we have not considered domain constraints of input parameters, so we will investigate how domain constraints should be handled.

References

[1] oscommerce website. http://www.oscommerce.com/.

[2] Perl cpan modules. http://www.cpan.org/modules/index.html.

[3] Selenium web page. http://seleniumhq.org/projects/.

[4] I Alexander. Misuse cases: Use cases with hostile intent. *IEEE Softw.*, 20(1):58–66, 2003.

[5] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[6] E.G. Amoroso. *Fundamentals of Computer Security Technology*. Prentice Hall PTR, Englewood Cliffs, NJ, 1994.

[7] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, NY, 1990.

[8] M. Howard and D. LeBlanc. *Writing Secure Code*. Microsoft Press, 2nd edition, 2003.

[9] M. Howard, D. LeBlanc, and J. Viega. *19 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. McGraw-Hill, 2005.

[10] E. Martin and T. Xie. Defining and measuring policy coverage in testing access control policies. In *Proceedings of the 8th International Conference on Information and Communications Security*, pages 139–158, December 2006.

[11] E. Martin and T. Xie. Automated test generation for access control policies via change-impact analysis. In *The 3rd International Workshop on Software Engineering for Secure Systems*, May 2007.

[12] E. Martin and T. Xie. A fault model and mutation testing of access control policies. In *Proceedings of the 16th international conference on World Wide Web*, pages 667–676, May 2007.

[13] A. Masood, R. Bhatti1, A. Ghafoor, and A. Mathur. Model-based testing of access control systems that employ RBAC policies. Technical Report SERC-TR-277, Purdue University, September 2005.

[14] A. Masood, A. Ghafoor, and A. Mathur. Scalable and effective test generation for access control systems that employ RBAC policies. Technical Report SERC-TR-285, Purdue University, September 2006.

[15] J. McDermott. Abuse-case-based assurance arguments. In *ACSAC '01: Proceedings of the 17th Annual Computer Security Applications Conference*, page 366, Washington, DC, USA, 2001. IEEE Computer Society.

[16] B. Potter, B. Allen, and G. Mcgraw. Software security testing. In *IEEE Security & Privacy*, pages 32–36, September 2004.

[17] R. Richardson. CSI Survey: Computer crime and security survey. *Computer Security Institute*, 2007.

[18] G. Sindre and A Opdahl. Eliciting security requirements with misuse cases. In *Proceedings of TOOLS Pacific*, pages 120–131, 2000.

[19] F. Swiderski and W. Snyder. *Threat Modeling*. Microsoft Press, 2004.

[20] H. Thompson and J. Whittaker. Testing for software security. *Dr. Dobb's Journal*, pages 24–34, November 2002.

[21] L. Wang, W. Wong, and D. Xu. A threat model driven approach for security testing. In *The 3rd International Workshop on Software Engineering for Secure Systems*, May 2007.

[22] Inc. WebCohort. Only 10% of web applications are secured against common hacking techniques. 2004.

[23] D. Xu. *Software security*. Wiley Encyclopedia of Computer Science and Engineering, B. W. Wah (Editor-In-Chief), Volume 5, pages 2703-2716, John Wiley & Sons, Inc., Inc., Hoboken, NJ, January 2009.

[24] D. Xu and K. Nygard. Threat-driven modeling and verification of secure software using aspect-oriented petri nets. *IEEE Transactions on Software Engineering*, 32(4):265–278, 2006.