SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2013; **43**:241–258
Published online 26 January 2012 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2111

A threat model-based approach to security testing

Aaron Marback[1], Hyunsook Do[1,*,†], Ke He[2], Samuel Kondamarri[1] and Dianxiang Xu[2]

[1]*Computer Science, North Dakota State University, Fargo, ND, USA*
[2]*Computer Science, Dakota State University, Madison, SD, USA*

SUMMARY

Software security issues have been a major concern in the cyberspace community, so a great deal of research on security testing has been performed, and various security testing techniques have been developed. Threat modeling provides a systematic way to identify threats that might compromise security, and it has been a well-accepted practice by the industry, but test case generation from threat models has not been addressed yet. Thus, in this paper, we propose a threat model-based security testing approach that automatically generates security test sequences from threat trees and transforms them into executable tests. The security testing approach we consider consists of three activities in large: building threat models with threat trees; generating security test sequences from threat trees; and creating executable test cases by considering valid and invalid inputs. To support our approach, we implemented security test generation techniques, and we also conducted an empirical study to assess the effectiveness of our approach. The results of our study show that our threat tree-based approach is effective in exposing vulnerabilities. Copyright © 2012 John Wiley & Sons, Ltd.

Received 14 January 2011; Revised 1 October 2011; Accepted 3 January 2012

KEY WORDS: threat modeling; threat trees; security testing; software testing

1. INTRODUCTION

Software security issues have been a major concern in the cyberspace community. These issues include attacks that deny service, corrupt data, and disclose confidential information. Although 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 [1, 2]. For instance, one 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 [3]. A more recent survey performed by Cenzic [1] reported that the number of attacks continues to increase and that hackers become more sophisticated and more organized.

One approach to improving software security involves software testing. To date, researchers have developed various security testing techniques. These include techniques that generate test cases or identify vulnerabilities focusing on specific attacks, such as SQL injection or cross-site scripting (XSS) [4–7]; generate test cases using model-based approaches, such as threat modeling or use case modeling [8–12]; and generate test cases from control policy specifications [13, 14] (Section 2 provides details).

Although each of these approaches has strengths and weaknesses for different types of security issues, threat modeling particularly provides a systematic way to identify threats that might compromise security. Further, this approach can be utilized not only during software testing phase but also during software design and development phases for secure design and risk mitigation. Owing to these merits, a threat modeling approach has been a well-accepted practice by the industry [15, 16],

*Correspondence to: Hyunsook Do, Computer Science, North Dakota State University, Fargo, ND, USA.
†E-mail: hyunsook.do@ndsu.edu

and recently, researchers have begun to investigate a threat modeling approach to security testing [11, 12]. Although their work has made some progress in this area, an important open issue is how test cases can be generated automatically from threat models.

Thus, in this paper, we propose a threat model-based security testing approach that generates security test cases from threat trees. This approach consists of two steps. The first step involves automated test sequence generation from threat trees. The second step involves transforming the security tests into an executable form by considering valid and invalid inputs. Security attacks typically result from malicious actions 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.

To demonstrate the feasibility of our approach, we have developed security test generation techniques that require the two aforementioned steps and performed an empirical study using real-world Web applications. The results of our study showed that our testing approach is effective in exposing vulnerabilities that are present in the Web applications that we examined. Through our study, we evaluated the feasibility of our approach that generates executable security test cases from threat trees.

The rest of the paper is organized as follows. Section 2 discusses background information and related work. Section 3 presents threat modeling for Web applications under test. Section 4 presents test generation techniques and their implementations. Section 5 presents our study design and results. Section 6 presents conclusions and discusses future work.

2. BACKGROUND AND RELATED WORK

Threat modeling has emerged as a viable practice for secure software development, which involves identifying, specifying, evaluating, and counter-measuring potential security attacks (i.e., threats). It can be conducted at various levels of abstraction and granularity or in different development phases, such as requirements analysis, design, and even testing [17]. Although existing methods have used threat trees for risk-based security testing [16, 18], they do not generate security tests systematically from threat trees.

Several notations have been proposed for threat modeling, such as threat trees (a variation of fault trees for safety analysis) [15], threat nets [9, 12], and misuse cases (based on use case modeling) [8, 10]. Threat nets are based on Petri nets, a mathematically based formalism for modeling and verifying distributed systems. Building threat models with threat nets requires expertise in formal methods. Misuse case modeling specifies misuse cases as security threats to use cases and suggests mitigation use cases. It is used for the elicitation and analysis of security requirements. Because threat trees are intuitive and easy to use, they have been widely adopted by the software industry. Therefore, this paper focuses on security testing with threat trees. Wang *et al.* [11] have proposed a threat model-driven security testing approach for detecting undesirable threat behaviors at runtime. 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. In comparison, this paper is based on Microsoft's threat modeling approach, where an application is decomposed using data flow diagrams and security threats to the application are modeled with threat trees. Our goal is to systematically generate security tests from threat trees.

One approach to evaluating the effectiveness of security testing is mutation analysis, which creates mutants by injecting vulnerabilities deliberately and determines how many of the injected vulnerabilities can be revealed by security tests. The existing work on security mutation falls into three categories: (1) implementation-level security mutation, (2) specification-level security mutation, and (3) mutation of access control policies. Implementation-level security mutation is to inject vulnerability into implementation and to evaluate how many of the injected vulnerabilities can be revealed by security tests. Shahriar and Zulkernine have developed the MUTEC and MUSIC tools for mutation testing of XSS and SQL injection attacks against Web applications [19, 20]. MUTEC provides five and six mutation operators for modifying JavaScript code and PHP code, respectively [20]. MUSIC provides four operators for injecting faults into where conditions of SQL queries and five operators for injecting faults in database API method calls [19]. Fonseca *et al.* [21] assessed the effectiveness of Web scanning tools by injecting XSS and SQL injection faults in Web applications. Tuya *et al.* [22] developed the SQLMutation tool that provides four categories of mutation operators for SQL SELECT queries. They include SQL clause operators, operator replacement mutation operators, NULL mutation operators, and identifier replacement. The aforementioned work focuses on particular types of vulnerabilities (SQL injection and XSS). Unlike these approaches, in this work, our approach is evaluated using various types of real vulnerabilities not injected vulnerabilities.

Specification-level security mutation is to introduce errors in the specification of security-related behaviors so as to generate threat scenarios. Wimmel and Jürjens used such mutation for security test generation from structured system diagrams in AutoFocus, a computer-aided software engineering tool for graphically specifying distributed systems [23, 24]. Abstract test sequences are determined with respect to the system's required security properties using mutations of the system specification and threat scenarios. These abstract tests are then converted into executable tests. In [23], security testing focuses on the fail-safety principle based on the common electronic purse specifications. In the aforementioned work, threat scenarios are generated from the specification according to security properties. Our work is based on the explicit specifications of various threats, such as spoofing, tampering with data, and denial of service.

For mutation analysis of access control policies, there are two main systems used to write access control policies. One uses organization-based access control or role-based access control (RBAC) as a modeling scheme to abstractly design a security policy without the implementation [25, 26]. The other uses eXtensible Access Control Markup Language (XACML) to create the actual implementation of a security policy [13, 27, 28]. Le Traon *et al.* have investigated various issues of testing access control policies, such as test criteria of access control policies [25], test generation from access control models [29], and selection and transformation of functional tests for policy testing [30]. Martin and Xie [13, 27, 28] have been investigating techniques for test generation from access control policy specifications written in XACML (OASIS XACML). They have defined policy coverage criteria [27] and a mutation testing framework for XACML access control policies [13]. To generate tests from policy specifications, they synthesized inputs to a change-impact analysis tool [28]. Masood *et al.* [14, 31] have investigated a state-based approach to test generation for RBAC policies. This approach first constructs a finite state model of the RBAC policy and then drives tests from the state model. The aforementioned work focuses on testing whether the intended access control policies are enforced, but not from the standpoint of how the system can be attacked.

Blackburn *et al.* [32] have developed an approach to automated generation of functional security tests. In this approach, security properties are modeled in Software Cost Reduction (SCR). An SCR-to-Test Vector (T-VEC) translator is used to translate the SCR model to a T-VEC test specification. T-VEC tools are then used on the T-VEC representation of the security properties to automatically generate test vectors (i.e., test cases with test input values, expected output values, and traceability information) and requirement-to-test coverage metrics. Jürjens [33] has developed an approach for testing security-critical systems based on UMLsec models. Test sequences for security properties are generated from UMLsec models to test the implementation for vulnerabilities. Different from this work, our work generates security tests solely from threat models.

Another challenge related to transforming the test sequences generated from threat trees to executable test code is identifying feasible input data. Because the primary concern with software security testing is finding security vulnerabilities [34], although both valid inputs and invalid inputs are required for testing any software system, invalid inputs are more important to security testing because, by nature, they verify whether 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 far more invalid inputs than valid inputs. We also need to test the 'presence of an intelligent adversary bent on breaking the system' [35]. Several security testing techniques that address invalid input generation have been proposed. Fuzz testing [36] provides negative random inputs to the program. Although this approach is easy to apply, it tends to only find simple faults. Other approaches that target specific types of flaws that could cause security breaches, such as buffer overflows, command/SQL injection, or XSS vulnerabilities [4–7, 37]. In this paper, we provide an input data generation approach that uses the design information of an application and threat models as the basis for input categorization and generation instead of targeting specific types of security flaws.

3. THREAT MODELING

Threat modeling is a systematic process of identifying, analyzing, documenting, and mitigating security threats to a software system [15]. This approach allows development teams to understand a system's threat profile by examining the application through the eyes of a potential attacker and helps them determine potential security risks posed to a system. The main steps of threat modeling are the following: (1) identifying the assets of an application, (2) determining the threats to an application, (3) ranking the threats, and (4) mitigating the threats.

In this paper, we used the first two steps to identify threats relevant to the applications under test. In the first step, we decompose an application under development using data flow diagrams to identify the assets of an 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 flows between the components. Confidential and sensitive data to the application, such as credit card numbers and social security numbers, should be identified during this step.

In the second step, we use the identified components as the threat targets for the threat model, categorize the threats by *STRIDE* model, and model the threat using threat trees. *STRIDE* stands for *S*poofing identity, *T*ampering with data, *R*epudiation, *I*nformation disclosure, *D*enial of service, and *E*levation of privilege. STRIDE model provides a simple way to categorize security threats and helps testers determine the resources required to expose security vulnerabilities. The threat tree describes the decision-making process an attacker would go through to compromise the component.

After identifying potential threats, we model them with threat trees [15] and generate test cases from threat trees. The following subsection describes how threats are modeled with threat trees.

3.1. Modeling threats with threat trees

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 subgoals, and the subgoals are further decomposed until leaf nodes representing the individual attack actions are determined. A node of threat trees can be decomposed as either (1) a set of subgoals, all of which must be achieved for the parent goal to succeed (the relationship between subgoals is represented as an AND relationship), or (2) a set of subgoals, any one of which must be achieved for the parent goal to succeed (the relationship between sub-goals is represented as an OR relationship). Threat trees describe the decision-making process that an attacker would go through to compromise the software. Typically, software has a set of threat trees that are relevant to its operation. The root of each threat tree represents a potential threat goal that could violate the security goal or policy of the software. Each threat tree enumerates and elaborates the ways that an attacker could cause events to occur. Each sequence of events in a threat tree represents a unique attack on the software. Different paths form different sequences of attack actions that an attacker could use to achieve the threat goal of a threat tree.

We illustrate the threat tree using an example. Suppose that we have a Web application system that provides services to multiple users with different privileges to access the system. To identify possible threats to this system, we examine categories defined by the STRIDE method and decide the threat that the system may be vulnerable to if an attacker runs unauthorized commands (*elevation of privilege* in STRIDE). Once we identify the threat, we define specific goals and events that can compromise the system using design documentation.

To gain access to unauthorized administration commands, an attacker needs to perform the following three basic tasks in order: (1) gain access to the system, (2) log into the system, and (3) run system commands. Each task can have subgoals or events as we explained earlier. For instance, the first task can be achieved by either physical access to the system or remote access to the system. To achieve the third task, the attacker could try two techniques: a buffer overflow attack, which can be attempted on any system that does not have safety features in place that forbid users for overwriting data on the runtime stack; and a brute force attack, which can be attempted against any system that does not limit the number of times that the attacker can enter credentials or commands.

Figure 1 shows the entire threat tree of this threat. In the threat tree, AND relationships are represented by edges from a parent node to the child nodes that are connected by an AND conditional connector. For instance, in the figure, the `Attacker executes unauthorized system command` has three children that have an AND relationship. To execute this attack, the attacker must execute all of the child nodes that appear from left to right (i.e., `Attacker accesses the system` followed by `Attacker logs into the system` then `Attacker executes system commands`). OR relationships are represented by edges from a parent node to child nodes that do not contain an AND conditional connector. For instance, the two child nodes of (`Attacker accesses the system`) have an OR relationship. The attacker can execute any of the child nodes (i.e., `Attacker has physical access` or `Attacker has remote access`) to realize that goal. In the following section, we describe the process of generating event sequences from threat trees.

4. TEST GENERATION

4.1. Security test sequence generation from threat trees

Having developed threat trees, we now need to generate executable test cases from them. To do this, we implemented a test generation technique that (1) imports threat trees created using the Microsoft Threat Modeling Tool [15] and creates test sequences; (2) provides a graphical interface so that testers map events (gathered from the threat trees) in the test sequences to the applicable unit tests; (3) adds input parameters to test sequences; (4) generates test inputs including valid and invalid inputs; and (5) generates test scripts that execute all test sequences with assigned input values.

We will provide more detailed descriptions for each of these activities in this section and in Section 4.2. We refer to these activities as ‘Test Gen Activities’.

Figure 2 shows how these activities are related to each other. First, we generate security test sequences from threat trees and test parameters from input syntaxes. We then integrate test parameters into the test sequences to create complete model-level security test cases. With the implementation knowledge, we further convert the model-level security tests into implementation-level security tests. To illustrate our approach to test generation that is described in this section and the following section, we use the threat tree described in Section 3.1, *Adversary executes unauthorized system commands* (Figure 1), as an example.

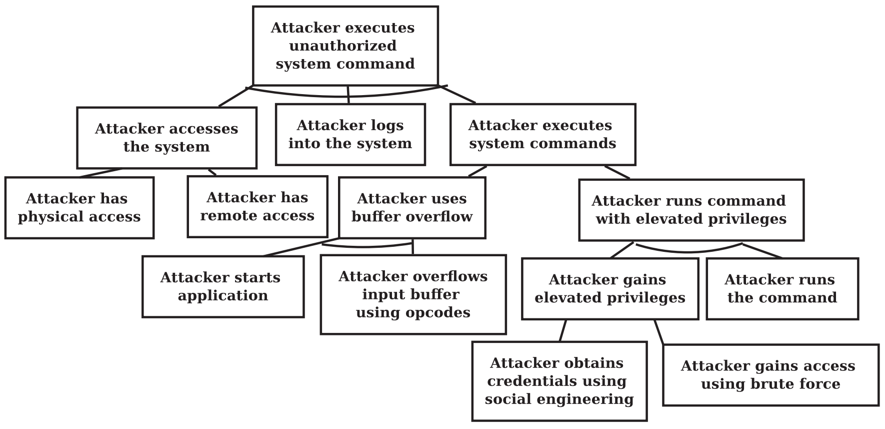
Attacker executes unauthorized system command
Attacker accesses the system
Attacker logs into the system
Attacker executes system commands
Attacker has physical access
Attacker has remote access
Attacker uses buffer overflow
Attacker runs command with elevated privileges
Attacker starts application
Attacker overflows input buffer using opcodes
Attacker gains elevated privileges
Attacker runs the command
Attacker obtains credentials using social engineering
Attacker gains access using brute force

Figure 1. Threat tree showing an attacker executing unauthorized system commands.

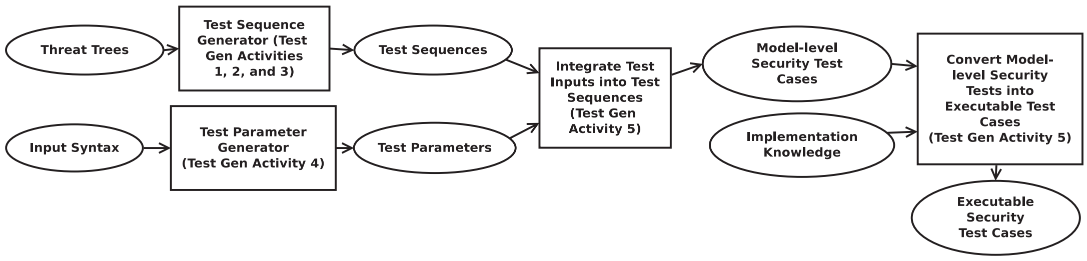
Threat Trees
Test Sequence Generator (Test Gen Activities 1, 2, and 3)
Test Sequences
Input Syntax
Test Parameter Generator (Test Gen Activity 4)
Test Parameters
Integrate Test Inputs into Test Sequences (Test Gen Activity 5)
Model-level Security Test Cases
Implementation Knowledge
Convert Model-level Security Tests into Executable Test Cases (Test Gen Activity 5)
Executable Security Test Cases

Figure 2. The process of security test generation.

For the first activity described previously (Test Gen Activity 1), we analyzed the '.tmd' files produced by the Microsoft Threat Model Tool. Because the '.tmd' files use XML version 1.0 syntax, they can be easily parsed using publicly available libraries. Using the Java Document-BuilderFactory class, we created a series of and/or tree-based data structures and inserted the threat tree data into our Java data structure; during this step, we added parameter information to the applicable leaf nodes in threat trees. For instance, in our example, a parameter `overflowed input` is required by the `Adversary overflows input buffer using opcodes` node, so we added this parameter to the `Adversary overflows input buffer using opcodes` node.

Once threat tree data has been placed in our and/or tree data structure, we generated test sequences by analyzing this tree (Test Gen Activity 2). We used a top–down approach for test sequence generation. The main goal of this step is to generate 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 as shown in Algorithm 1.

4.1.1. Test sequence generation algorithm. The test sequence generation algorithm (Algorithm 1) takes a threat tree as an input and generates a set of test sequences as an output. The algorithm analyzes a threat tree by using a breadth-first search to walk the tree and generate test sequences. For simplicity, we make two assumptions. First, the sibling nodes with an 'AND' or 'OR' relation are ordered—they are treated in the sequential order when test sequences are generated. Second, the same group of sibling nodes has a single AND or OR relation, but not a mixture. This does not lose generality because the mixed AND and OR relations can be eliminated by using more intermediate nodes when threat trees are built.

The algorithm starts by creating a test sequence that contains a root node. This test sequence is then added to the set of test sequences (line 4). The root node is also added to the queue of unvisited nodes (line 5). Every node in the queue of unvisited nodes is then processed until it is empty (line 6). The node at the front of the queue is dequeued (*CurrentNode*), and any children that it may have are enqueued into the unvisited nodes queue (lines 7–11). The algorithm then analyzes the nodes in the set of test sequences until every node that matches *CurrentNode* in every sequence is replaced with a leaf node by using the following replacement method. (1) If *CurrentNode* is in a test sequence that has children with an AND relationship, replace *CurrentNode* with these children in that sequence in the left-to-right order in which they appear in the tree (lines 12–22). (2) If *CurrentNode* is in a test sequence that has children with an OR relationship, we create the same number of sequences as children of the *CurrentNode* and replace *CurrentNode* in each of these sequences with one of the children of the OR relationship. Then, delete the original sequence containing *CurrentNode* (lines 24–35). The set of test sequences that remains after the substitutions is the final set of test sequences used for test generation.

We illustrate this algorithm by using our example (Figure 1). We begin by analyzing the root node. A set of test sequences containing only a sequence with the root node (`Adversary executes unauthorized system commands`) is created. This node is then added to the unvisited nodes queue and is then dequeued. The child nodes of the root node are added to

```
Input   : $TT$ - A Threat Tree, a threat tree is composed of nodes that have an ordered set (left to right) of child nodes. Each node
          also has an AND or OR relationship for the set of child nodes
Output  : $TestSequences[]$ - The set of all test sequences.
Declare : $CurrrentNode$ - The node currently being examined
          $UnVisitedNodes[]$ -The queue of nodes that have not been visited
          $TestSequences[]$ - The set of all test sequences.
          $TempSequence[]$ - A Temporary sequence of nodes
$TestSequencs \longleftarrow \emptyset$;
$UnVisitedNodes[] \longleftarrow \emptyset$;
$TempSequence \longleftarrow \emptyset$;
Add $\{TT$.root$\}$ to $TestSequencs$;
$UnVisitedNodes$.enqueue($TT$.root);
while $UnVisitedNodes[] \neq \emptyset$ do
    $CurrrentNode \leftarrow UnVisitedNodes$.dequeue() ;
    if $CurrrentNode.Children \neq \emptyset$ then
        foreach $x \in CurrrentNode.Children$ do
            $UnVisitedNodes$.enqueue($x$) ;
        endforeach
        if $CurrrentNode.RelationshipOfChildren = AND$ then
            foreach $seq \in TestSequences$ do
                $TempSeqeunce \leftarrow seq$ ;
                foreach $node \in TempSeqeunce$ do
                    if $CurrentNode = node$ then
                        Replace $node$ in $TempSeqeunce$ with all elements $node$.Children in the order that they appear;
                        $TestSequences \longleftarrow TestSequences \cup \{TempSequence\}$;
                        $TestSequences \longleftarrow TestSequences \setminus seq$;
                    endif
                endforeach
            endforeach
        else
            //CurrentNode.RelationshipOfChildren = OR
            foreach $seq \in TestSequences$ do
                $TempSeqeunce \leftarrow seq$ ;
                foreach $node \in TempSeqeunce$ do
                    if $CurrentNode = node$ then
                        foreach $y \in node.Children$ do
                            Replace $node$ in $TempSequence$ with $y$ ;
                            $TestSequences \longleftarrow TestSequences \cup \{TempSequence\}$;
                        endforeach
                        $TestSequences \longleftarrow TestSequences \setminus seq$;
                    endif
                endforeach
            endforeach
        endif
    endif
endwhile
```

Algorithm 1. Breadth-first search test sequence generation.

the unvisited nodes queue. With the use of the substitution algorithm described previously, the sequence containing only the root node in the set of test sequences is replaced by the following sequences: `Adversary accesses the system` → `Adversary logs into the system` → `Adversary executes system commands`. We then dequeue the first child node, `Adversary accesses the system`. Because this node has two children that have an OR relationship, we create two new sequences with each child (`Adversary has physical access`; `Adversary has remote access`) and replace `Adversary accesses the system` with these new sequences.

We proceed to the next node in the queue, `Adversary logs into the system`. Because it does not have any children, we proceed to the next node. We then dequeue the next node in the queue, `Adversary executes system commands`. Because this node has two children that have an OR relationship, we create two new sequences with each child (`Adversary uses buffer overflow`; `User runs command with elevated privileges`) and replace `User runs command with elevated privileges` with these new sequences. We continue this process until all sequences have been generated and the unvisited node queue is empty.

When we have completed analyzing the threat tree, this algorithm generates six test sequences from our example:

1. `Adversary has physical access` → `Adversary logs into the system` → `Adversary starts application` → `Adversary overflows input buffer using opcodes`
2. `Adversary has physical access` → `Adversary logs into the system` → `Adversary obtains credentials using social engineering` → `Adversary runs the command`
3. `Adversary has physical access` → `Adversary logs into the system` → `Adversary gains access using brute force` → `Adversary runs the command`
4. `Adversary has remote access` → `Adversary logs into the system` → `Adversary starts application` → `Adversary overflows input buffer using opcodes`
5. `Adversary has remote access` → `Adversary logs into the system` → `Adversary obtains credentials using social engineering` → `Adversary runs the command`
6. `Adversary has remote access` → `Adversary logs into the system` → `Adversary gains access using brute force` → `Adversary runs the command`

Because the test sequences have been generated from design-level threat trees, we need to transform the test sequences into executable tests (Test Gen Activity 2). 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. For instance, in the case of social engineering attacks, we are forced to assume that an attacker has access to some data that are typically accessible only to users of the application. In the case of distributed attacks, we were able to create test cases for the application. However, these test cases had to be executed on multiple computers simultaneously, which involved writing a test harness manually.

As mentioned earlier, many events can be associated with parameters (Test Gen Activity 2). Because a test sequence can consist of many events each with multiple parameters, keeping track of the order of all of the parameters associated with all of the events in a sequence can be challenging. To address this issue, we needed to take the following steps. First, we specified parameters in the Microsoft Threat Modeling Tool. These parameters are imported into our tree structure during the parsing phase. Once test sequence generation has been completed, we then generate a list of parameters by iterating through the events in a sequence. The generated list is helpful for test developers because it specifies which inputs are needed for a test case, the order in which they occur, and which inputs are represented by which arguments. For instance, in our example, the `Adversary overflows input buffer using opcodes` event would probably contain one parameter (`overflowed input`) that tells that testers need to provide one input for that event.

4.2. Syntax-based test input generation

Once we have obtained test cases with parameters, we need actual inputs to execute the test cases (Test Gen Activity 4). To generate test inputs, we used a syntax-based testing approach, which is syntax-based and input data-driven testing [38, 39] considering valid inputs and invalid inputs that could pose security threats. To achieve this, we have developed a technique that (1) retrieves parameter information (the number of parameters, data type and format, constraints on values) from the generated test sequences; (2) creates a syntax by using the data type, format information, and constraints given by the specification in the threat trees; (3) constrains the values produced by this syntax even more by using implementation information and common malicious strings (in the case of invalid inputs); (4) generates values from this syntax and all constraints.

To illustrate these steps, we consider our prior example that requires `overflowed input` to successfully execute the system commands. First, we retrieve the parameter information (`overflowed input`) from the threat tree. Second, we constrain our malicious input by specifying the data type, in this case, it is a string (we need a text input combined with an opcode and a return address). We then define a syntax for the overflow string. The simplest syntax for the overflow string is the regular expression a^* where a is any printable ASCII character. After we have a format for the regular expression, length constraints can be also applied to the inputs to reduce the total number of inputs. The total number of inputs after specifying a length constraint can still be too large to feasibly handle. In this case, a length constraint of L will produce 95^L (where 95 is the number of printable ASCII characters) test inputs.

To further reduce the number of inputs and identify malicious input values (step 3), we use source code and design documentation of the application under test to retrieve information related to inputs, such as valid values and dependency constraints. In this case, let us assume that the input is stored in a character array with a length of 10.

Now, with this information, we can create malicious inputs that cause security attacks. For instance, we can create test cases that realize an *Adversary overflows input buffer using opcodes* attack on this system by using fields that accept strings as input (in this example, `overflowed input`).

We utilize the design documentation of the system to determine which opcodes are valid for the system. From this design information, we define a syntax for the input generation. In this particular example, let us assume that the adversary wants to assign a different value to some variable in the runtime stack of the application. One common approach for this attack is to overwrite the return address of a function. This program will then jump to a section of the program that the adversary has specified. We then generate a syntax that can be used to create malicious strings for the attack. In this example, the overflow string consists of characters to fill the buffer, a collection of instructions that will execute without crashing the program (this is typically a copy of the original instruction in the program that is between the input buffer and the return call), a jump command, and a malicious return address. Figure 3 shows the syntax definition for this parameter.

Once we define a syntax for inputs, we need to create actual values for them (step 4). Consider the *Buffer Overflow* part of our example. To realize the *Buffer Overflow* attack, we need to create invalid inputs for the `overflowed input`. To do so, we can parse the syntax that we defined in the previous step (Figure 3) and automatically generate a pattern of malicious *Buffer Overflow* stings. One string that can generated from this syntax is the combination of **0123456789** for the input buffer (a 10-character hexadecimal string), **8800000087654321** for the valid instructions(a hexadecimal string whose length is application specific), **EB** for the jump opcode, and **1234567890ABCDEF** for the return address (a 16-character hexadecimal string). The end result of the syntax-based input generation would yield **01234567898800000087654321EB1234567890ABCDEF**.

Having generated all test sequences and required test inputs, we then create test scripts that execute all test sequences with assigned input values (Test Gen Activity 5). Once we have generated a list of inputs associated with a test case, we can easily create a test harness script that passes these inputs to the executable test script as command line arguments. Figure 4 shows a complete example of an executable test case with `$myagrgs` as the command line argument. In this example test case, the adversary is trying to overflow the `ISBN field` in a Web application by using the string generated by the syntax in Figure 3 (**01234567898800000087654321EB1234567890ABCDEF**) as a command line argument.

⟨Overflowed Input⟩→⟨Input Buffer Chars⟩⟨Valid Instructions⟩⟨Jump Opcode⟩⟨Malicious Return Address⟩
⟨Input Buffer Chars⟩→[0-F]^{10}
⟨Valid Instructions⟩→[0-F]*
⟨Jump Opcode⟩→ **EB**
⟨Malicious Return Address⟩→[0-F]^{16}

Figure 3. An example of buffer overflow input syntax for a 16-bit x86 computer.

```
#/usr/bin/perl-w
#p0:overflowedInput
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;
$agent-> get('http://example.com/product_info.php?ISBN='.“$ARGV[$myargs]”);
$agent-> click();
```

Figure 4. Combined executable test case.

5. EMPIRICAL STUDY

As stated in Section 1, we wished to assess our threat model-based security testing approach considering the following research question:

> Can test cases generated from STRIDE-based threat trees be effective in detecting potential vulnerabilities in a given target system?

To investigate this research question, we performed an empirical study. The following subsections present, for this study, our objects of analysis, study setup, and data and analysis.

5.1. *Objects of analysis*

For this study, we used two open source Web applications as objects of analysis: *osCommerce* [40] and *Drupal* [41]. OsCommerce is a Web-based storefront and shopping cart application. Drupal is a content management system that is specifically suited for blogging. These applications are written in the PHP programming language and use the MySQL database system. They are real, nontrivial Web applications that have been used by a large number of users.

Table I lists, for each of our objects, its associated 'Lines of code', 'Threat trees' (the total number of threat trees), 'Test sequences' (the total number of test sequences generated from threat trees), and 'Test cases' (the total number of test cases generated from threat tree-based test sequences). The following subsections describe details about these artifacts' construction.

5.1.1. Threat trees. In building threat trees, we initially started with osCommerce and identified 13 threats. Then, we examined the threats to see whether they can be applied to Drupal because some functionalities are common to the Web applications, such as user authentication. We were able to reuse four of them with minor modifications and found two new threats for Drupal. Owing to the space limit, we describe six threat trees that represent each category from *STRIDE* model. (see Table II in Section 5 for the STRIDE category of the identified threat trees). The first three threat trees present existing vulnerabilities in osCommerce.

5.1.1.1. Hijacking a user's session ID. This threat can be realized by XSS. *XSS* is one of the most commonly used attack methods and is responsible for large portions of all security vulnerabilities [42]. XSS attacks allows an attacker to embed malicious code into Web pages. By doing so, the attacker can steal users' confidential information or can change user settings [43]. For instance, in

Table I. Objects of analysis.

Application	Lines of code	Threat trees	Test sequences	Test cases
osCommerce	29,320	13	63	259
Drupal	27,771	6	18	106

Table II. Threat trees for osCommerce.

Name of the threat tree	Threat type	Threat description
Hijacking a user's session ID	Spoofing	An attacker can execute arbitrary HTML and script code using an administrator's browser session.
Stealing user information using SQL injection	Tampering	An attacker can obtain user information using SQL injection by entering malicious inputs.
Multiple login attempts	Elevation of privileges	An attacker can login as a registered user.
Data modification	Repudiation	An attacker can access and modify restricted files within the application or the underlying system.
Denial of service using local resources	Denial of service	An attacker can perform a denial of service by depleting various system resource of the Web application's host system.
User data disclosure	Information disclosure	This threat can be realized by directly attacking an SQL server associated with the application
Login information vulnerability	Spoofing	This threat makes use of the steal session information vulnerability to change the username and the password
Session information disclosure	Information disclosure	Access to the referral log file, and session fixation are the two ways of achieving the goal of stealing the session information
Session ID exhaustion	Denial of service	If an attacker can get the session IDs exhaust, his or her goal of denial of service to the genuine users will be achieved
Transaction resource	Denial of service	Credit card authorization process is attacked to keep the system busy from doing the actual user request.
User data tampering	Tampering	An attacker tampers with the user data in the database with the help of SQL injection and session information.
Data tampering using access to admin files	Tampering	Because of unprotected admin files, an attacker is able to find the admin files.
User credential disclosure	Information disclosure	An attacker can query the database for user credentials.

the case of osCommerce, user-generated input that is passed to the applications is not properly sanitized, allowing the attacker to execute arbitrary HTML and script code using an administrator's browser session (see Figure 5).

5.1.1.2. Stealing user information with SQL injection. SQL injection is also a very common method used by attackers. In osCommerce, there are a few places where an SQL code can potentially be injected by an attacker. During the process of adding items to the shopping cart, the attacker is

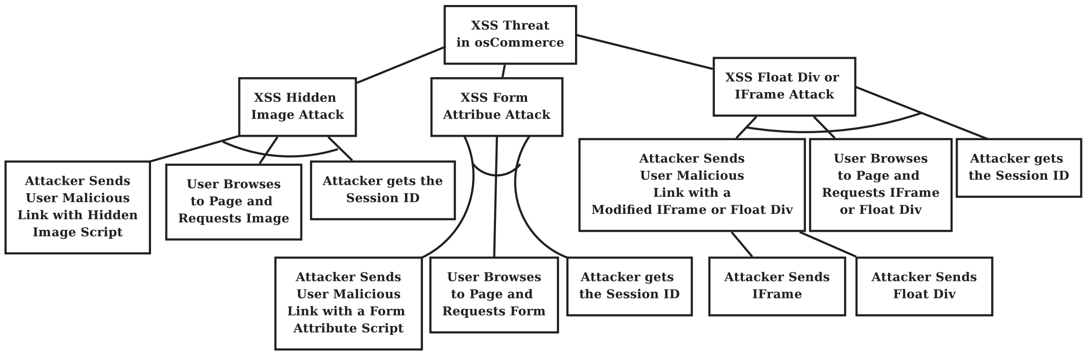
XSS Threat in osCommerce
XSS Hidden Image Attack
XSS Form Attribue Attack
XSS Float Div or IFrame Attack
Attacker Sends User Malicious Link with Hidden Image Script
User Browses to Page and Requests Image
Attacker gets the Session ID
Attacker Sends User Malicious Link with a Form Attribute Script
User Browses to Page and Requests Form
Attacker gets the Session ID
Attacker Sends User Malicious Link with a Modified IFrame or Float Div
User Browses to Page and Requests IFrame or Float Div
Attacker gets the Session ID
Attacker Sends IFrame
Attacker Sends Float Div

Figure 5. Threat tree showing hijacking a user's session ID.

allowed to enter a malicious input into some of the fields related to different product options (e.g., quantity or size). When the attacker submits this datum, the unsanitized and malicious information is injected to the database. We modeled a threat tree with three attack approaches to exploit this threat. As shown in Figure 6, the attacker tries to obtain users' credit card information, personal information, or login information by injecting corresponding SQL statements.

5.1.1.3. Multiple login attempts. By not limiting the number of user login attempts, an attacker can guess the user ID and password, allowing the attacker to login as a registered user. We modeled a threat tree with four attack approaches to exploit this threat. As shown in Figure 7, the attacker tries to guess a user ID or password by using a brute force attack or dictionary attack. Dictionary attacks use a large list of probable inputs to attack an application.

5.1.1.4. Data modification. Data modification is also a common threat that attempts to modify the application source code by combining XSS with the known file structure of the application. This exploit allows an attacker to view the contents of any file on an improperly configured system. The attacker can access and modify restricted files within the application or the underlying system. Figure 8 shows a threat tree for this threat (the whole tree contains more components, but we show

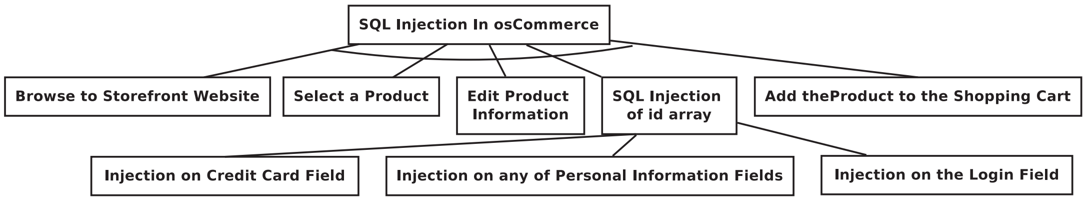
SQL Injection In osCommerce
Browse to Storefront Website
Select a Product
Edit Product Information
SQL Injection of id array
Add theProduct to the Shopping Cart
Injection on Credit Card Field
Injection on any of Personal Information Fields
Injection on the Login Field

Figure 6. Threat tree showing stealing user information with SQL injection.

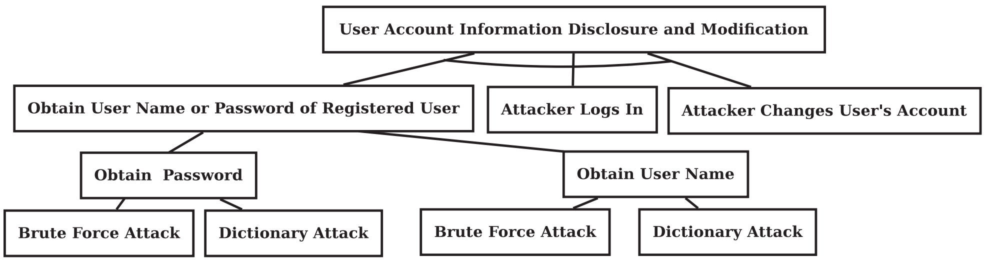
User Account Information Disclosure and Modification
Obtain User Name or Password of Registered User
Attacker Logs In
Attacker Changes User's Account
Obtain Password
Obtain User Name
Brute Force Attack
Dictionary Attack
Brute Force Attack
Dictionary Attack

Figure 7. Threat tree showing multiple login attempts.

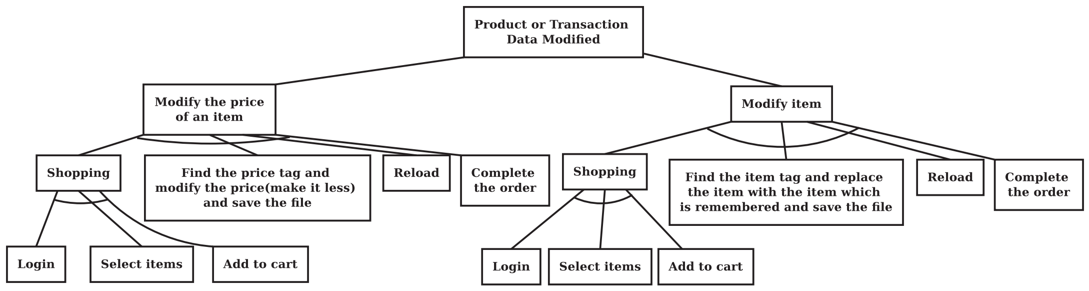
Product or Transaction Data Modified
Modify the price of an item
Modify item
Shopping
Find the price tag and modify the price(make it less) and save the file
Reload
Complete the order
Shopping
Find the item tag and replace the item with the item which is remembered and save the file
Reload
Complete the order
Login
Select items
Add to cart
Login
Select items
Add to cart

Figure 8. Threat tree showing data modification.

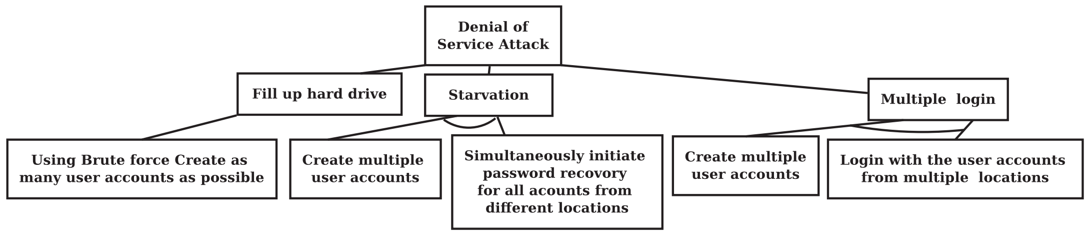
Denial of Service Attack
Fill up hard drive
Starvation
Multiple login
Using Brute force Create as many user accounts as possible
Create multiple user accounts
Simultaneously initiate password recovory for all acounts from different locations
Create multiple user accounts
Login with the user accounts from multiple locations

Figure 9. Threat tree showing denial of service using local resources.

only a partial threat tree because of formatting constraints). In this threat tree, the attacker tries to pass malicious scripts to the different components of the application by using application source code. For instance, the attacker can select an item, try to lower the price of this item in the HTTP POST request, and submit the request.

5.1.1.5. Denial of service using local resources. An attacker can perform a denial of service attack by attempting to deplete various system resources of the Web application's host system. This attack tries to stress the system under test as much as possible. As shown in Figure 9, three attack sequences are included in this threat: attempting to fill the hard drive by repeatedly creating user accounts, executing hundreds of password recoveries by email and creating multiple accounts simultaneously, and performing multiple logins simultaneously.

5.1.1.6. User data disclosure. User data disclosure can be realized by directly attacking an SQL server associated with the application. If an attacker has credentials for the database and the database is not secured properly, the attacker should be able to view any user's data. The attacker can also perform an attack by hijacking a user's session information from the referrer log or by getting a user to disclose their credentials. For instance, the attacker can capture a victim's session information using the referrer log. Once the session information is captured, the attacker has full access to the victim's account. Figure 10 shows this threat.

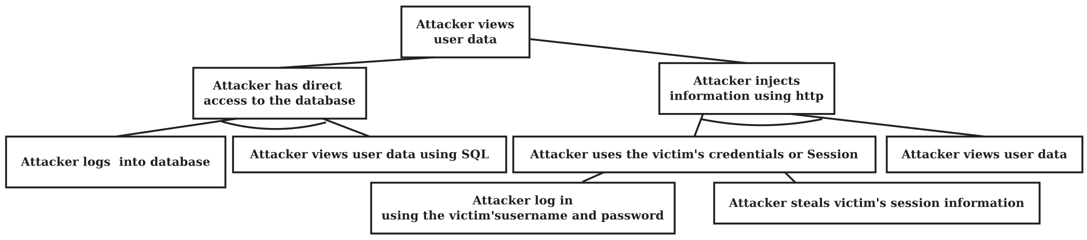
Attacker views user data
Attacker has direct access to the database
Attacker injects information using http
Attacker logs into database
Attacker views user data using SQL
Attacker uses the victim's credentials or Session
Attacker views user data
Attacker log in using the victim'susername and password
Attacker steals victim's session information

Figure 10. Threat tree showing user data disclosure.

All the threat trees we developed for osCommerce are shown in Table II. Having developed the threat trees, we examined them to see whether they can be applied to Drupal. We were able to reuse four of them with minor modifications: *Multiple login attempts*, *Denial of service*, *User data disclosure*, and *User data tampering*. However, we could not reuse the rest of them because of various reasons: some functionalities do not exist in Drupal such as the session ID and credit card processing. We also identified two new threat trees for Drupal; thus, we have six threat trees in total for Drupal as shown in Table III.

5.1.2. Test sequences and test cases. In our study, we used *Selenium* [44] to develop and run test cases. Selenium is a widely used integrated development environment for Web application testing. It provides functions such as record and replay for efficiently developing and running test cases. From threat trees, we generated 63 test sequences for osCommerce and 18 test sequences for Drupal by using our tool described in Section 4.1. For all tests in our study, we developed a relevant input list by using the syntax-based generation technique described in Section 4.2. This technique used design documentation (variable name, database table, and field names), a database containing common malicious strings (SQL, XSS, and command injection string), and implementation information (specific values from the applications database backend) to generate the inputs for the test sequences.

Because assigning all possible values to test sequences is prohibitively expensive and ineffective, we created test inputs following the procedure we explained in Section 4.2. For each of the applicable test sequences, we assigned multiple combinations of valid and invalid inputs depending on the type of test sequences.

5.2. Study setup

We performed our study using two different virtual machines (one as a client, one as a server) on the same host. The client operating system was Ubuntu Linux version 9.04, 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 study, virtual machines on the same host are used to provide a nearly optimal network connection.

In order to execute test cases, we implemented a test assignment and generation module. This module allows a test to assign tests to all of the events in 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 tester to save and reload

Table III. Threat trees for Drupal.

Name of the threat tree	Threat type	Threat description
Multiple login attempts	Elevation of privileges	See Table II
Denial of service using local resources	Denial of service	See Table II
User data disclosure	Information disclosure	See Table II
User data tampering	Tampering	See Table II
Data modification using SQL injection	Tampering	Previews on comments were not passed through normal form validation routines, enabling users with the 'post comments' permission and access to more than one input filter to execute arbitrary code.
Infinite accounts of accounts on from same IP	Denial of service	Allows the same ip address to make an infinite number the website host until hard drive fills

a test assignment session because this can be a very time-consuming process. The module then combines the assigned test files into executable test cases that correspond to our event sequences.

5.3. Data and analysis

Tables IV and V show the results for osCommerce and Drupal, respectively. The tables list, for each of threats, data on its associated number of test sequences (Sequences), the number of test cases executed (Test cases), the number of sequences that exposed vulnerabilities (Test sequences exposed vulner.), and the number of test cases that exposed vulnerabilities (Test cases exposed vulner.). Multiple inputs were used for each sequence as mentioned in Section 5.1.2. For instance, in the case of *Multiple login attempts* in Table IV, we tested nine valid input combinations (a valid username and password) for each of the four different sequences (36 test cases in total). Among these, 28 test cases exposed vulnerabilities. By varying our input for this particular threat, we were able to determine that if only one field (either username or password) was assigned to an invalid input, we could expose the vulnerability.

To address our research question, we need to examine whether test cases generated from our approach would either expose a vulnerability or lack thereof. In the case of osCommerce, we have designed threats by using three different sources: for threats 1, 2, and 3, we used known vulnerabilities; for threats 4 and 13, we used attack patterns that had been properly handled by reading application documentation; for the rest of them, we used common attack patterns without any prior knowledge whether they exposed vulnerabilities or not. As expected, in the first case (threats 1, 2,

Table IV. Test results for osCommerce.

#	Threats	Sequences	Test cases	Test sequences exposed vulner.	Test cases exposed vulner.
1	Hijacking a user's session ID	4	28	4	20
2	Stealing user information using SQL injection	3	24	3	18
3	Multiple login attempts	4	36	4	28
4	Data modification	18	72	0	0
5	Login information vulnerability	1	2	1	2
6	Session information disclosure	5	15	5	15
7	Session ID exhaustion	3	3	1	1
8	Transaction resource	1	1	1	1
9	Denial of service using local resources	6	6	3	3
10	User data disclosure	4	20	3	9
11	User data tampering	3	15	3	15
12	Data tampering using access to admin files	1	1	1	1
13	User credential disclosure	9	36	0	0
Total		62	249	29	113

Table V. Test results for Drupal.

#	Threats	Sequences	Test cases	Test sequences exposed vulner.	Test cases exposed vulner.
1	Multiple login attempts login	4	24	4	20
2	Denial of service	5	5	3	3
3	User data disclosure	3	27	1	7
4	User data tampering	2	26	0	0
5	Data modification using SQL injection	1	5	0	0
6	Infinite accounts from same IP	1	1	1	1
Total		16	88	9	31

and 3), our test cases could expose all known vulnerabilities that are present in osCommerce. As also expected, in the second case (threats 4 and 13), our test cases could not expose any vulnerabilities because these vulnerabilities had been properly handled by developers. This confirms that developers' or designers' security concerns actually have been handled as intended. In the last case (threats 5 through 12), our test cases could expose all vulnerabilities that we have additionally identified.

In the case of Drupal, all threats except for threat 6 were taken from common attack patterns without any prior knowledge of existing system vulnerabilities. Among the unknown five vulnerabilities, our test cases could expose three of them, but the rest of them were not exposed. The reason for this is that Drupal makes extensive use of input sanitization functions built into PHP.

The results from these applications show that test cases generated from threat trees can be effective at exposing vulnerabilities.

5.4. Discussion

Our results show that our threat model-based security testing approach can be effective in discovering unmitigated threats. Through the threat tree-based approach from the adversary's perspective, we could identify the threat goals that an attacker wants to realize, and these threat goals are the targets of our security testing. Different sequences of attack actions have been derived from the threat trees, and these sequences of attack actions represent the ways in which an attacker realizes the threat goals and thus provide a basis for security test sequences.

The results also show that some threat trees that exposed vulnerabilities in one application could be used to expose vulnerabilities in other application after being slightly modified. This could possibly be attributed to a few factors: (1) the programs are both written in PHP, use a MySQL database for data storage, and are deployed on the Apache Web server; (2) both have some similar functionality, especially when it comes to user management authentication; (3) the programs are both Web applications that have an identical high-level architectural design (typical three tier). The threats that were not applicable for both applications were typically due to the areas where the programs differ in functionality or implementation. The denial-of-service threat is a vulnerability that is present in both applications. This is likely due to the similar architectures. The user data tampering threat is an example where osCommerce was vulnerable and Drupal was not. This has to be due to implementation differences in the safeguards the applications use when accessing the database backend. It should be noted that all unknown vulnerabilities discovered during this study have been reported to the developers.

5.4.1. Threats to validity. Although our empirical results are promising, like any empirical study, this study has limitations that must be considered when interpreting its results. One limitation involves how the threat trees have been developed. Owing to a lack of comprehensive design documentation for our object programs, we created some threat trees from known vulnerabilities instead of design documents. Also, because this technique requires developers to build threat trees and the accompanying tests, the effectiveness may be directly affected by the ability of individuals who develop threat trees. Another limitation of our study is a lack of variety in object programs. We tried to minimize this limitation by performing realistic attacks against large, readily deployed Web applications that have been used by a large number of users. Nevertheless, additional studies with other programs are needed to address this limitation. Another limitation comes from the use of STRIDE to construct and categorize threats. STRIDE only covers a subset of all possible threats that a system could possibly be vulnerable to. Our technique is, by no means, an exhaustive approach for discovering all security vulnerabilities, but it does provide a systematic way for identifying possible threats.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have developed threat model-based security testing techniques that systematically generate security test cases from threat trees and conducted a study to assess our approach. These security test cases can verify whether the system under test is vulnerable to the identified security threats. In our study, we have used two real-world Web applications to evaluate the effectiveness of our security testing approach. Our results showed that our threat tree-based testing approach is effective in exposing security vulnerabilities that are present in the software systems. In addition to these promising results, our approach also provides several significant advantages for security testing: (1) we could identify threat goals that an attacker wants to realize using threat trees; and (2) we could generate executable security tests automatically from threat trees considering actual inputs.

For future work, we intend to apply our approach to more applications and perform controlled experiments for evaluating our approach through comprehensive mutation analysis of security vulnerabilities according to Open Web Application Security Project's 10 most critical Web application security risks [45]. We also plan to integrate domain constraints of input parameters into the process of syntax-based test input generation. To do so, we will enhance threat trees with the specification of domain constraints. When actual inputs are to be created for a test sequence that is derived from a threat tree, they need to satisfy all the constraints in the sequence. A potential solution is to enhance our test input generator with a constraint solver. This technique could also be combined with the security policy monitoring technique proposed by Wang *et al.* [11] to provide an oracle framework for the generated tests.

ACKNOWLEDGEMENT

This work was supported in part by NSF under Award CNS-0855106.

REFERENCES

1. Cenzic. Application Security Trends Report Q1 2008. Available from: http://www.cenzic.com [September 2009].
2. Richardson R. CSI Survey: Computer Crime and Security Survey, 2007.
3. WebCohort, Inc. Only 10% of Web applications Are Secured Against Common Hacking Techniques, 2004.
4. Halfond W, Orso A, Manolios P. Using positive tainting and synrax-aware evaluation to counter SQL injection attacks. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, November 2006.
5. Kiezun A, Guo P, Jayaraman K, Ernst M. Automatic creation of SQL injection and cross-site scripting attacks. In *Proceedings of the International Conference on Software Engineering*, May 2009; 199–209.
6. Lam M, Martin M, Livshits B, Whaley J. Securing Web applications with static and dynamic information flow tracking. In *Proceedings of PEPM*, San Francisco, CA, January 2008.
7. Wassermann G, Yu D, Chander A, Dhurjati D, Inamura H, Su Z. Dynamic test input generation for Web applications. In *Proceedings of the International Conference on Software Testing and Analysis*, July 2008.
8. Alexander I. Misuse cases: use cases with hostile intent. *IEEE Softw* 2003; **20**(1):58–66.
9. McDermott J. Abuse-case-based assurance arguments. In *ACSAC '01: Proceedings of the 17th Annual Computer Security Applications Conference*, Washington, DC, USA, 2001; 366. IEEE Computer Society.
10. Sindre G, Opdahl A. Eliciting security requirements with misuse cases. In *Proceedings of TOOLS Pacific*, 2000; 120–131.
11. Wang L, Wong W, Xu D. A threat model driven approach for security testing. In *The 3rd International Workshop on Software Engineering for Secure Systems*, May 2007.
12. Xu D, Nygard K. Threat-driven modeling and verification of secure software using aspect-oriented Petri nets. *IEEE Transactions on Software Engineering* 2006; **32**(4):265–278.
13. Martin E, Xie T. Automated test generation for access control policies via change-impact analysis. In *The 3rd International Workshop on Software Engineering for Secure Systems*, May 2007.
14. Masood A, Ghafoor A, Mathur A. Scalable and effective test generation for role-based access control systems. *IEEE Transactions on Software Engineering* 2009; **35**(5):654–668.
15. Swiderski F, Snyder W. *Threat Modeling*. Microsoft Press: Redmond, WA, 2004.
16. Wysopal C, Nelson L, Zovi DD, Dustin E. *The Art of Software Security Testing: Identifying Software Security Flaws (Symantec Press)*. Addison-Wesley Professional: Boston, MA, 2006.
17. Xu D. Software Security. In *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons, Inc.: Hoboken, NJ, 2008.
18. Howard M, LeBlanc D. *Writing Secure Code*, 2nd ed. Microsoft Press: Redmond, WA, 2003.
19. Shahriar H, Zulkernine M. MUSIC: Mutation-based SQL injection vulnerability checking. In *Proceedings of the 8th International Conference On Quality Software*, Aug 2008; 77–86.
20. Shahriar H, Zulkernine M. MUTEC: Mutation-based testing of cross site scripting. In *Proceedings of the 5th International Workshop on Software Engineering for Secure Systems*, 2009; 47–53.
21. Fonseca J, Vieira M, Madeira H. Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks. In *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, dec 2007; 365–372.

22. Tuya J, Suarez-Cabal M, Riva C. Mutating database queries. *Information and Software Technology* april 2007; **49**(4):398–417.
23. Jürjens J, Wimmel G. Formally testing fail-safety of electronic purse protocols. In *Proceedings of the 16th IEEE international conference on Automated software engineering,* ASE ’01, Washington, DC, USA, 2001; 408. IEEE Computer Society.
24. Wimmel G, Jürjens J. Specification-based test generation for security-critical systems. In *Proceedings of the Fourth International Conference on Formal Engineering Methods*, 2002; 471–482.
25. Le Traon Y, Mouelhi T, Baudry B. Testing security policies: Going beyond functional testing. In *Proceedings of the International Symposium on Software Reliability Engineering*, November 2007.
26. Mouelhi T, Le Traon Y, Baudry B. Mutation analysis for security tests qualification. In *Proceedings of Mutation*, 2007.
27. Martin E, Xie T. Defining and measuring policy coverage in testing access control policies. In *Proceedings of the 8th International Conference on Information and Communications Security*, December 2006; 139–158.
28. Martin E, Xie T. A fault model and mutation testing of access control policies. In *Proceedings of the 16th international conference on World Wide Web*, May 2007; 667–676.
29. Mouelhi T, Fleurey F, Baudry B, Le Traon Y. A model-based framework for security policy specification, deployment and testing. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, October 2008.
30. Mouelhi T, Le Traon Y, Baudry B. Transforming and selecting functional test cases for security policy testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, April 2009.
31. Masood A, Bhatti1 R, Ghafoor A, Mathur A. Model-based testing of access control systems that employ RBAC policies. *Technical Report SERC-TR-277*, Purdue University, September 2005.
32. Blackburn M, Busser R, Nauman A, Chandramouli R. Model-based approach to security test automation. *Quality Week* June 2001.
33. Jürjens J. Model-based security testing using UMLsec. *Electronic Notes in Theoretical Computer Science (ENTCS)* December 2008; **220**(1):93–104.
34. Thompson H, Whittaker J. Testing for software security. *Dr. Dobb’s Journal* November 2002; **27**(11):24–34.
35. Potter B, Allen B, Mcgraw G. Software security testing. In *IEEE Security & Privacy,* September 2004; **2**(5):81–85.
36. Oehlert P. Violating assumptions with fuzzing. *IEEE Security and Privacy* 2005; **3**(2):58–62.
37. Larson E, Austin T. High Coverage Detection of Input-Related Security Faults. In *Proceedings of the USENIX Windows System Symposium*, jul 2000; 1–10.
38. Ammann P, Offutt J. *Introduction to Software Testing*. Cambridge University Press: New York, NY, 2008.
39. Beizer B. *Software Testing Techniques*. Van Nostrand Reinhold: New York, NY, 1990.
40. osCommerce Web page. Available from: http://www.oscommerce.com/ [September 2009].
41. Drupal Web page. Available from: http://www.drupal.org/ [September 2009].
42. Symantec. Symantec Internet Security Threat Report trends for July–December 07.
43. CGI Security. The Cross-Site Scripting FAQ. Available from: http://www.cgisecurity.com/xss-faq.html [September 2009].
44. Selenium Web page. Available from: http://seleniumhq.org/projects/ [September 2009].
45. OWASP Web page. Available from: http://www.owasp.org/ [September 2009].