COVERAGE-BASED TEST GENERATION FOR XACML POLICIES

by

Ning Shen

A project

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

Spring 2015

Introduction

Access Control is one of the most important and widely used security mechanisms in the information security field. In this field, ABAC(Attribute Based Access Control) is a new method which enable fine-grained access control. ABAC defines attributes as subjects, resources, actions and environments and combines various attributes into access control decisions. XACML, which stands for eXtensible Access Control Markup Language, is an OASIS standard for specifying ABAC policies in the XML format. Compared with other access control methods, ABAC has two remarkable advantages. First, though ABAC is new, it can be backward compatible with other traditional access control methods, such as Role-Based Access Control(RBAC), Mandatory Access Control(MAC) and Discretionary Access Control(DAC). Second, within these Access control methods, ABAC is the most flexible and meaningful one so it can be used within a large enterprise or across multiple organization.

In XACML, there are three core structural elements, which are PolicySet, Policy and Rule. The PolicySet can contains one or more other PolicySet or Policy. The Policy consists of one or more Rules, and the Rule is the basic element in a XACML policy. Because PolicySets can contain one or more Policies or PolicySets, it is necessary to provide a way to evaluate the decision of each sub-decision and return a final decision. For example:

Rule 1: Customers can buy up to 20 items.

Rule 2: Customers can not make a purchase over $500.

There is potential conflict between these two rules. XACML handle this situation by using

the concept called Combining Algorithm. Combining Algorithm specifies which Rules/Policies has the higher level to make the final decision. For instance, “Deny Override” is one of the Combining Algorithms in XACML, which means if one Rule/Policy is evaluated as Deny, no matter how many other Permits we get, the final result should be Deny. XACML defines a number of Combining Algorithm that can be identified in Policy or PolicySet. They are:

- Deny-overrides: if any rule / policy returns Deny then the evaluation stops and the overall result is Deny
- Permit-overrides: if any rule / policy returns Permit then the evaluation stops and the overall result is Permit
- Deny-unless-permit: will always return Deny decision unless there is at least one permit decision(Indeterminate or NotApplicable will never be the result)
- Permit-unless-Deny: will always return Permit decision unless there is at least one permit decision(Indeterminate or NotApplicable will never be the result)
- First-Applicable: rules are evaluated in the order in which they are listed. If a rule’s target and condition evaluate to “true”, then return its effect. If a rule’s target or condition evaluates to “false”, then move to next rule. If no further rule exists, then return “NotApplicable”. If an error happens, then return “Indeterminate”.

In the real world, the consequences of such mistakes might grant unauthorized access and deny legitimate access. Finding such mistakes might be time consuming because the user might have to review the whole XACML policies to locate a fault.

Besides combining algorithms, there are several other mistakes in XACML that might lead to security faults and conflicts. Potential security faults and conflicts might include missing

Rules/Targets/Conditions, missing attributes, Rule/Policy with wrong effects, and coding errors.

To test XACML policies, adequate test coverage is important because a defect in a policy element will not be revealed unless that policy element is exercised by some test case. In this project, we generate the test suites based on different coverage criteria, and we demonstrate that coverage-based testing is very useful for validation of XACML policies.

Project Statement

In this project, we define some new coverage criteria for XACML 3.0 policies and then develop efficient methods for automatically generating test cases to meet these coverage criteria. Also, we will use some examples to evaluate the cost effectiveness of our test generation methods. We hope these test cases can detect a majority of faults and security flaws in XACML policies.

Motivation

There are several methods to generate test cases for XACML 1.0 or 2.0 policies. However, they have two problems. First, they are not cost-effective because they either are incapable of detecting many defects or produce a large number of test cases. Second, the test cases generated by these methods do not achieve adequate coverage of the XACML policy under test. There are three contribution in this project. First, we define a new set of coverage criteria for XACML 3.0 policies, including rule coverage, decision coverage, MC/DC coverage, and rule pair coverage. These criteria are defined based on the elements of XACML specifications. Second, by using the constraint solver Z3-str, we develop an efficient method for automatically generating test cases to meet the coverage criteria' requirements. Third, we will conducte empirical studies to evaluate the cost-effectiveness of the test generation methods for the coverage criteria. We collect some real world XACML policies then create mutants based on them, and executed the mutants with the generated test cases. We call a mutant is killed if there is at least one test case that reports a failure. All of these attempts, if succeed, may become a powerful and efficient way to detect faults and security flaws in XACML policies.

Methods

Coverage-Based Testing in XACML

Before generating the test cases, we first introduce some new coverage criteria that we used in this project.

- Rule Coverage

  In this method, a test suite TS for a policy P is said to satisfy rule coverage of P if, for each rule R in P, there is at least one test case q in TS that evaluate R to its specified effect.

- Decision Coverage

  In XACML, a policy has targets and individual rules and each rule may also have individual target and condition. A test suite TS for a policy P is said to satisfy decision coverage(DC) of PS if:

  - (1) TS has three tests to make policy target pt evaluate to true, false and error, respectively,
  - (2) for each rule in Policy P, TS has three tests to make rule target evaluated to true, false, and error,
  - (3) and for each rule in P, TS has three tests to make rule condition evaluated to true, false, and error

- MC/DC Coverage

  MC/DC stands for the modified condition/decision coverage, which is "the primary means used by aviation software developers to obtain Federal Aviation Administration(FAA) approval of ariborne computer software"[1]. Here we call a test suite TS satisfies MC/DC coverage of Policy P if:

  - (1) TS satisfies MC/DC of policy target PT, and has a test to make PT evaluate to

error,

- (2) for each rule in P, TS achieves MC/DC of rule's Target and has a test to make rule's Target to error,
- (3) and for each rule in P, TS achieves MC/DC of rule's Condition and has a test to make rule's Condition to error

- Rule Pair Coverage

  A test suite TS for a policy P is said to achieve rule pair coverage if TS, for each pair of rules within each policy P, TS has a possible test to make both rules evaluate to their specified effects.

From the four definitions, we will generate four groups of test cases and each of group will satisfied one of the four coverages.

Query Generation by Constraint Solver Z3-str

In query generation, making a target/condition evaluate to true or false requires us to make the target/condition's constraints to be satisfied or dissatisfied. These constraints may be boolean constraints, string constraints, numeric constraints,etc. So we need a general constraint solver to derive these values.

Z3 is an efficient SMT(Satisfiability Modulo Theories) Solver from Microsoft Research. Z3 supports basic data types(e.g., int and boolean). However, Z3 doesn't support strings. To solve

this problem, we found another tool called Z3-str, which extends Z3 by treating string as a primitive type and supporting common string operations.

In order to use Z3-str, we also need to convert the constraint from XACML to Z3-str inputs. These conversions will be introduced in the project report. The last step is to invoke the Z3-str to solve and constraint and convert the result back into a query.

Reachability

In XACML, if we want to generate test cases for a policy/rule, we must make sure that policy/rule is reachable. A policy/rule is reachable if there is a request such that the decision is made based on this policy/rule. In a large set of policies/rules, there might have some policies/ rules which are not reachable. For example, in Deny-Overrides, if one Deny rule evaluates to true, then the evaluation process stops all the rules behind it are not reachable. In project, we define the reachability of rule R as:

The policy target evaluates to true and no rule before R in policy terminates the evaluation of that policy. That is:

i. For Deny-overrides and Deny-unless-permit, rule R is reachable only if no other Deny rule before R evaluates to NotApplicable/ID.
ii. For Permit-override and Permit-unless-deny, rule R is reachable only if no other Permit rule before R evaluates to NotApplicable/IP.

iii. For First-Applicable, rule R is reachable only if all the rules before R return NotApplicable.

Test Generation For Different Coverage Criteria

i. Rule Coverage

Test generation of rule coverage aims to generate a query to cover each rule. We first need to compose the reachability conditions, including obtaining the reachability of the policy and the reachability of each rule within a policy. Then we add the rule's constraint to the whole reachability conditions and pass them to the Z3-str. If Z3-str can derive corresponding values then we say that we generate a test case that covers the targeted rule.

ii. MC/DC and Decision Coverage

For each constraint, we use a truth table to manage its MC/DC coverage. Each entry consists of three parts: a truth value for each of the basic conditions that comprise the constraint, decision of the constraint, and whether the entry is covered by some existing test. For example, the following table shows a sample truth table for MC/DC coverage of resource-id = Liquor resource-id = Medicine.

Basic Conditions	Decision	Covered

resource-id = Liquor	resource-id = Medicine	resource-id = Liquor resource-id = Medicine	
T	F	T	F
F	T	T	F
F	F	F	F
		E	F

(T: True; F: False; E:Error)

When generating queries, we deal with three levels of MC/DC coverage truth tables for policy target, rule target and rule condition. Rule target is reachable only when policy target is true. Rule condition is reachable when policy target and rule target are both true. That is, we explore the next level only when the decision of current level is true. If the decision of current entry is false and not covered yet, we use Z3-str to generate a query that make it evaluated to Not-Applicable. If the current entry is error and not covered yet, we generate an error query for it. After a query is generated, we update the coverage information in all MC/DC truth tables. The Test generation of Decision coverage is a special case of test generation for MC/DC coverage, which entries in MC/DC coverage are generated by an MC/DC tool while entries in decision coverage are generated by hard code.

Mutant Generation

(Coded by Jimmy Wang)

Since the purpose of testing is to find faults, a fault model is usually useful for effective testing. A fault model represents certain types of faults that occur in XACML. Each mutant is a variation of the original policy, where one fault of a particular type is injected. A mutant is said to be killed by a test suite if execution of the test suite reports a failure. The fault detection capability of the test suite can be evaluated by the number of killed mutants divided by the total number of mutants.

In this project, we include 8 types of mutants that are used by previous researches very often, they are:

i. Policy Target True(PTT): Remove the Target of each Policy ensuring that the Policy is applied to all requests.
ii. Policy Target False(PTF): Modify the Target of each Policy ensuring that the Policy is never applied to a request.
iii. Change Rule Combining Algorithm(CRC): Replaces the existing rule combining algorithm with another rule combining algorithm. The set of considered rule combining algorithms is {deny-overrides, permit-overrides, first-applicable}.
iv. Change Rule Effect(CRE): Changes the rule effect by replacing Permit with Deny or Deny with Permit.
v. Rule Target True(RTT): Remove the Target of each rule ensuring that the Rule is applied to all requests.
vi. Rule Target False(RTF): Modify the Target of each rule such that the Rule is never applied to a request.
vii. Rule Condition True(RCT): Removes the condition of each Rule ensuring that the

Condition always evaluates to True.

viii. Rule Condition False(RCF): Manipulates the Condition values or the Condition functions ensuring that the Condition always evaluates to False.

Also, we introduce some other kinds of mutants, they are:

ix. Remove Rule(RER): Chooses one rule and removes it.

x. First Permit Rule(FPR): It moves in each policy the rules having a *Permit* effect before those ones having a *Deny* effect. The evaluation priority among the rules having a *Permit* effect and that among the rules having a *Deny* effect is maintained.

xi. First Deny Rule(FDR): It moves in each policy the rules having a *Deny* effect before those ones having a *Permit* effect. The evaluation priority of the rules having a *Permit* effect and that of the rules having a *Deny* effect is maintained.

xii. Add NotFunction (ANF): It adds the *Not* function as first function of each Condition element.

xiii. Remove NotFunction (RNF): It deletes the *Not* function defined in the condition.

Experiment

■ Subject Policies

To verify our work, we collect five XACML policies from Internet as our examples. All these policies are either used by previous researches or used by some companies as a demo. These five XACML policies are: K-market, itrust, pluto, conference, and fedora. Among them, itrust has the largest size, which has 64 rules in it. We notice that even 64 is a relatively small size, so we also create another large sample so that it will become closer to real world cases.

■ Expected Results

For each coverage criterion, we expect a complete test suite be generated and meet the criterion's requirements. We will use the policies and their mutants to test our test suites. From design aspect, the fault detection ratio for each criterion, from high to low, should be:

i. MC/DC coverage

ii. Decision coverage

iii. Rule coverage

Project schedule

Oral Presentation: Feb 19th, 2015

Translate XACML constraints to Z3-str: Done

Algorithm of test generation: Done

Gui: Feb 30th, 2015

Testing : March 5th, 2015

Project report: May 15th, 2015

Final oral Defense: June 1st, 2015

Bibliography

i. Hayhurst, Kelly; Veerhusen, Dan; Chilenski, John; Rierson, Leanna. “A practical tutorial on modified condition/decision coverage,” NASA. May 2001.

ii. Balana. Open source XACML 3.0 implementation. http://xacmlinfo.org/2012/08/16/balana-the-open-source-xacml-3-0-implementation/

iii. eXtensible Access Control Markup Language (XACML) Version 3.0 http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-cs-01-en.pdf

iv. Bertolino A., Daoudagh S., Lonetti F., Marchetti E. “Automatic XACML requests generation for policy testing”, The Third International Workshop on Security Testing (SecTest 2012), pp. 842 - 849. Montreal, QC, Canada, April 2012.

v. Bertolino, A, Le Traon Y., Lonetti, F., Marchetti, E., Mouelhi, T. “Coverage-based test cases selection for XACML policies,” 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp.12-21.

Artifacts

i. Code of reading and acquiring elements in XACML policies.

ii. Code of translating XACML constraints to Z3-str.

iii. Code of generating different groups of test suites.

iv. Code of a simple user interface.

v. A brief user guide.