Automated Test Generation for Access Control Policies

Evan Martin
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
eemartin@csc.ncsu.edu

Advisor: Tao Xie
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
xie@csc.ncsu.edu

ABSTRACT

Access control policies are increasingly written in specification languages such as XACML. A dedicated software component called a Policy Decision Point (PDP) receives access requests, evaluates requests against specified policies, and returns responses to inform whether access should be granted. To increase confidence in the correctness of specified policies, policy developers can conduct policy testing to probe the PDP with some typical test inputs (in the form of requests) and check test outputs (in the form of responses) against expected ones. Unfortunately, manual test generation is tedious and manually generated tests are often not sufficient to exercise various policy behaviors.

In this paper we present an efficient test generation approach and its supporting tool called Targen. We model the policy under test as a tree, perform a depth-first traversal specifically targeting each rule, and generate a set of requests for each rule by satisfying the necessary constraints to cover the rule. We further reduce the number of requests for inspection based on structural coverage information. Because some rules may be redundant being unreachable, we also develop an novel approach for redundant-rule detection and its supporting tool called Cirg, developed based on a change-impact analysis tool. We have evaluated Targen on policies collected from various sources, some of which are complex policies being used in real systems. Our results show that Targen can effectively generate tests to achieve high structural coverage of policies and outperforms the existing random test generation in terms of structural coverage and fault-detection capability. Cirg can identify a large number of redundant rules among rules defined in a complex, real policy.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging—Debugging aids, Testing tools

Keywords

XACML, Test Generation, Access control policy

Supported in part by an IBM Ph.D. Fellowship.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA ©06 Portland, Oregon, USA
Copyright 2006 ACM X-XXXXXX-XX-X/XX/XX ...$5.00.

1. INTRODUCTION

Software testing aims at efficiently detecting errors in software through dynamic execution. Errors in policy specifications may also be discovered by leveraging existing techniques for software testing and applying them to policy testing. In policy testing, test inputs are access requests and test outputs are access responses. The execution of test inputs occurs as requests are evaluated by the PDP against the access control policies under test. Policy authors can inspect request-response pairs to check whether they are expected. Access control policies are often tested with manually defined access requests so that policy authors may check the PDP’s responses against expected ones [3]. Unfortunately, current policy testing practice tends to be a manual, ad hoc process. With such a process, it is questionable that high confidence can be gained in the correctness of access control policies.

2. APPROACH

The eXtensible Access Control Markup Language (XACML) is a language specification standard designed by OASIS. Developers can use XACML to express domain-specific access control policies, access requests, and access responses. It offers a large set of built-in functions, data types, and combining logic. It also provides standard extension interfaces for defining application specific features. XACML has increasingly attracted attention from both academia and industry.

We have defined three types of policy structural coverage corresponding to three major types of elements in XACML policies: policies, rules for each policy, and a condition for each rule. Like program coverage [7] for software testing, policy structural coverage can help determine whether the policy has been fully tested and which parts of policy have not been tested. To automatically measure policy structural coverage, we have developed a policy-structural-coverage measurement tool written in Java based on Sun’s open source XACML implementation [1]. We use this tool to reduce request sets such that the original and reduced sets achieve the same policy coverage.

Because manually generating requests for testing policies is tedious, we developed a random test generation tool for policies. The tool analyzes the policy under test and constructs a request factory that provides requests on demand. The factory generates requests by randomly selecting requests from the set of all combinations of attribute id-value pairs found in the policy. The tool represents the form of requests) and check test outputs (in the form of responses) against expected ones [3]. Unfortunately, current policy testing practice tends to be a manual, ad hoc process. With such a process, it is questionable that high confidence can be gained in the correctness of access control policies.

Because manually generating requests for testing policies is tedious, we developed a random test generation tool for policies. The tool analyzes the policy under test and constructs a request factory that provides requests on demand. The factory generates requests by randomly selecting requests from the set of all combinations of attribute id-value pairs found in the policy. The tool represents the form of requests) and check test outputs (in the form of responses) against expected ones [3]. Unfortunately, current policy testing practice tends to be a manual, ad hoc process. With such a process, it is questionable that high confidence can be gained in the correctness of access control policies.

Because manually generating requests for testing policies is tedious, we developed a random test generation tool for policies. The tool analyzes the policy under test and constructs a request factory that provides requests on demand. The factory generates requests by randomly selecting requests from the set of all combinations of attribute id-value pairs found in the policy. The tool represents the form of requests) and check test outputs (in the form of responses) against expected ones [3]. Unfortunately, current policy testing practice tends to be a manual, ad hoc process. With such a process, it is questionable that high confidence can be gained in the correctness of access control policies.
where $n$ is the number of attribute values found in the policy. The tool randomly picks an integer $i$ from 0 to $2^n$, convert it to a vector, and generate tests based on the vector. In particular, each request is generated by setting each bit in the vector to 0 or 1 with probability 0.5. The number of randomly generated requests can be configured by the user and the configured number can be considerably smaller than the total number of combinations.

To automatically generate high-coverage tests for access control policies, we develop a target driven approach that considers each rule in isolation and attempts to satisfy the constraints required for that rule to be applied in various ways. The policy under test is modeled as a hierarchy or tree structure. Each leaf in the tree represents a rule and each path from the root to the leaf contains a series of constraints found in the $<$Target$>$ tag. In particular, each target may have some number of attribute id-value pairs found in the subject, resource, and action section of the target. We collect these attribute id-value pairs in three sets; one for each section of the target. Once a leaf is reached, we use these sets to form a predicate $p$ out of $s + r + a$ independent clauses where $s$, $r$, and $a$ correspond to the number of id-value pairs in the subject, resource, and action set, respectively. Each id-value pair maps to a specific clause in the predicate $p$. Furthermore, the clauses within sets are or'ed and each of the predicates formed by the sets are and'ed. For example, let the subject, resource, and action set for a particular rule be denoted by $S = \{s_1, s_2\}$, $R = \{r_1, r_2\}$, and $A = \{a_1, a_2\}$. The predicate corresponding to this rule is $p = (s_1 \lor s_2 \lor s_3) \land (r_1 \lor r_2 \lor r_3) \land (a_1 \lor a_2 \lor a_3)$. A request set is generated that satisfies all possible combinations of truth values for each independent clause. Therefore, a predicate with $n$ independent clauses has $2^n$ possible assignments and so at most $2^n$ requests are generated for each rule.

Margrave [5] is a software tool suite written in PLT Scheme for analyzing access control policies written in XACML. More specifically, it provides mechanisms to verify properties against a policy, perform change-impact analysis between two policies, and counter-example generation. Our redundant-rule detection approach uses this generic and powerful Margrave API as a redundant-rule detection and request generation engine to efficiently detect redundant rules by exploiting its ability to perform change-impact analysis.

We use policy mutation testing to measure the fault-detection capability of a request set in the evaluation of our new test generation approach. Mutation testing [4] has historically been applied to general purpose programming languages. The program under test is iteratively mutated to produce numerous mutants, each containing one fault. A test input is independently executed on the original program and each mutant program. If the output of a test executed on a mutant differs from the output of the same test executed on the original program, then the fault is detected and the mutant is said to be killed. The fundamental premise of mutation testing as stated by Geist et al. [6] is that, in practice, if the software contains a fault, there will usually be a set of mutants that can only be killed by a test that also detects that fault. In other words, the ability to detect small, minor faults such as mutants implies the ability to detect complex faults. Because fault detection is the central focus of any testing process, mutation testing provides an external measure of the effectiveness of that process. The higher the percentage of killed mutants, the more effective the test set is at fault detection. We developed an automated policy mutation testing approach; in their approach, the program under test, test inputs, and test outputs correspond to the policy, requests, and responses, respectively. Given a policy, a mutator generates a number of mutant policies. Given a request set, their approach evaluates each request in the request set on both the original policy and a mutant policy. The request evaluation produces two responses for the request based on the original policy and the mutant policy, respectively. If these two responses are different, then their approach determines that the mutant policy is killed by the request; otherwise, the mutant policy is not killed.

3. CONCLUSION

To automatically generate high-coverage tests for complex real-world policies, we have developed an automatic test generation approach and its supporting tool called Targen based on combinatorial coverage [2] of independent clauses that determine whether a rule is applied to a given request. Because some rules may be redundant, being unreachable by any request, we have developed a novel approach and its supporting tool called Cirg that automatically detects redundant rules based on a change-impact analysis tool such as Margrave [5]. Because the number of generated tests is often high for complex policies, we reduce the size of the generated test set based on policy structural coverage so that developers can inspect the tests with reasonable efforts. We have evaluated Targen on a set of policies collected from various sources, most of which are complex ones used in real systems. The results show that Targen can effectively generate tests for complex policies to achieve high policy structural coverage. The results also show that Cirg can generate tests that have a higher policy structural coverage and higher fault-detection capability than the existing random test generation. We have also applied Cirg on a complex real-world policy and detected a high number of redundant rules among the rules in the policy.

4. REFERENCES