Test-Driven Assessment of Access Control in Legacy Applications

Yves Le Traon, Tejeddine Mouelhi
IT-TELECOM Bretagne
35576 Cesson Sévigné Cedex,
France
{yves.letraon,tejeddine.mouelhi}
@telecom-bretagne.eu

Alexander Pretschner
ETH Zürich
Switzerland
pretscha@inf.ethz.ch

Benoit Baudry
IRISA/INRIA
35042 Rennes
France
bbaudry@irisa.fr

Abstract

If access control policy decision points are not neatly separated from the business logic of a system, the evolution of a security policy likely leads to the necessity of changing the system's code base. This is often the case with legacy systems. We present a test-driven methodology to assess the flexibility of a system, a property that describes the degree of coupling between the access control logic and the business logic of a system. A low flexibility indicates that a modification of the policy will lead to substantial changes of the code. In this paper, we analyze the notion of flexibility which is related to the presence of hidden and implicit security mechanisms in the business logic. We detail how testing can be used for detecting such mechanisms and how it may drive the incremental evolution of a security policy. We use several case studies to illustrate and validate the methodology.

1. Introduction

Security policies (SPs) embody constraints on the access to data or functions of an organization, both for extra-organizational and intra-organizational subjects. Several access control models have been developed over the past decades, including RBAC [1-3] and Or-BAC [4, 5]. In essence, these models provide means to describe the subjects' permissions and prohibitions to access a resource, for instance, the right to configure a firewall or to access a specific service or record in a database.

From an access control point of view, a software system is composed of three parts: the interface, the business logic and the policy decision point (PDP). Conceptually, the PDP is the decision logic that checks whether or not access to a resource can be granted. Technically, PDPs can be implemented in a multitude of ways, including configurable dedicated components, explicit pieces of code, and by imposing architectural constraints.

In contrast, the business logic consists of the program code which is executed to implement the functional requirements. A request coming from the interface is checked by the PDP which allows or disallows the business logic to be called.

In the last years, researchers [6] have proposed to separate business and access control logics, and to automatically generate respective configuration files as well as application code from a set of related models. This approach, however, only works in model-based development processes and generates applications from scratch. It is thus not applicable to legacy systems. If legacy systems' SPs are modified, the code of these systems often has to be changed as well. However, locating necessary modifications in the code is far from trivial. The reason is that the implementation of PDPs is often interwoven with that of the business logic, and can furthermore be implemented explicitly or implicitly.

Explicit mechanisms are represented by dedicated pieces of code, and can be either visible or hidden. A security mechanism is *visible* in a legacy system if there is a traceability link from (a part of) the security policy to the security mechanism. Otherwise, the mechanism is *hidden*. Due to the lack of documentation and traceability, the location of the code implementing a hidden mechanism may be lost, and so may the knowledge of how it works. This problem especially occurs for large and old legacy systems. Finally, *implicit* mechanisms are a result of technical constraints imposed by the architecture, platform or implementation of the business logic.

While the business logic of a legacy system may implement a given security policy, it may equally restrict the evolution of the security policy. This is a result of hidden and implicit security mechanisms. Given all possible modifications of a security policy, we will use the term *flexibility* to denote the ability of a legacy system to evolve without tedious analysis and modifications of the program (detection and modification of the hidden and implicit security mechanisms, refactorings of the business logic).

Whenever a security policy evolves,

1. the explicit security mechanisms are modified and have to be tested;
2. hidden security mechanisms may be in conflict with the new access control rules: they must be located and adapted; and
3. the business logic may be in conflict with the new security policy: the reasons for this conflict (the implicit mechanisms) must be determined, and the design and the code may have to be modified and possibly refactored to make the legacy system more flexible.

If (black-box) testing legacy applications against new security policies reveals a mismatch between policy and implementation, the determination of the cause is usually difficult. Refactoring the design and reprogramming some parts of the system may be prohibitively expensive. In extreme cases, it might even become impossible to deploy the new security policy.

1.1. Problem Statement

The long-term goal of our research is methodological and technological support for the evolution of legacy systems in the context of changing SPs. In this paper, we take a first step by tackling the following problem. Given a legacy (or newly developed) system and a currently applicable SP, can we assess the coupling between business logic and access control logic? How difficult will it be to implement policy modifications?

1.2. Solution and Contribution

Our solution consists of a test-driven assessment methodology for the flexibility of legacy systems. We use technology described in earlier work [7, 8] to derive tests from policies. We apply small mutations to the original policy – simulating incremental evolutionary steps – and derive tests from each mutated policy. Essentially these tests are requests together with expectations as to whether or not the request is granted. We then disable the *visible security mechanisms* in the application. This is feasible precisely because of the existing traceability. Without hidden and implicit mechanisms, any request should now be granted. If implicit and hidden PDPs are present, in contrast, not all requests will be granted. By applying the generated tests to the system and comparing the actual with the expected access decisions, we extract information on implicit and hidden mechanisms.

Our test-driven approach provides an understanding of the technical reasons that restrict the evolution of the security policy for legacy systems. Because test cases are executable artifacts, they are precious means to help pinpoint those parts of the business logic that are not flexible w.r.t. the evolution of security policies.

The contribution of this paper is the proposed methodology as well as its empirical validation.

1.3. Overview

The remainder of this paper is organized as follows. Section 2 presents the background and the objectives, namely the detection of hidden and implicit security mechanisms. Section 3 presents the methodology, a two-stage test-driven assessment method. In a first step, the current system is analyzed based on exhaustive testing. In a second step, the system is assessed w.r.t. the possible evolutions of the current security policy. Experimental results are presented in Section 4. Section 5 presents the related works and the conclusion.

2. Background

Before detailing the proposed test driven methodology to estimate system's flexibility, we need to provide a few fundamental concepts.

2.1. Access control models

Advanced access control models [4, 6, 9] allow to express rules (a set of rules specifies a SP) that apply only under specific circumstances, called contexts. For instance, in the health care domain, physicians have special permissions in specific contexts such as emergencies. Furthermore, some models provide means to specify different SPs for various parts of an organization (sub-organizations). In this paper, we will consider an access control model in which we can specify permissions and prohibitions as a function of temporal and spatial contexts, roles, activities, and resources.

A rule can be a permission or a prohibition. Each rule of an SP consists of five parameters that we called entities: a *status flag* S indicating permission or prohibition, a *role*, an *activity*, a *view*, and a *context*. Our domain consists of *role names* RN, *activity names* PN, *view names* VN, and *context names* CN. A *security policy* SP is thus a set of rules defined by $SP \subseteq S \times RN \times PN \times VN \times CN$, and we will denote concrete rules by a predicate `Status(Role, Activity, View, Context)`.

As a running example, we will consider a library management system (LMS). Its purpose is to offer services to manage books in a public library.

All users can perform three *activities*, namely borrow, reserve, and return a book. There are also a number of administrative tasks that can be executed, but we do not need them here. In our model, data items are called *views*. Views are the entities that we want to protect from unauthorized access. In our example, there are two views, book and account. All entities can be hierarchical. In the LMS example, there are two types of accounts: borrower and personnel accounts. Finally, *contexts* include temporal contexts such as working day or weekend.

Several access control rules can then be expressed. For example, users are allowed to borrow books only when the library is open. This rule is defined by `Permission(Borrower, BorrowBook, Book, WorkingDays)`. Further examples of rules include the prohibition to borrow a book during holidays, the permission for an administrator to manage the personnel accounts, and the permission for a secretary to consult borrower accounts. These examples can be expressed by `Prohibition(Borrower, BorrowBook, Book, Holidays), Permission(Administrator, ManageAccess, PersonnelAccount, always), and Permission(Secretary, ConsultBorrowerAccount, BorrowerAccount, always).`

For the LMS, the total number of rules is 22. In a large scale system, the access control policy may become very complex, due to the large number of roles, activities and contexts that are involved. Priorities have to be set to specify which rule applies when several rules are in conflicts.

2.2. Visible security mechanisms

Implementing control over the access to resources of a system can be done in a variety of ways. A classical architectural pattern is one component that is dedicated to access control. Typically, it works as a filter between the interface of the system and the control logic functions (services) or data (e.g. access to the database). If it exists, and if it is mentioned in the documentation or located through traceability links, such a component can be easily modified to implement a new security policy. The security mechanism is *explicit* in that it has been created with the objective of controlling access. It is also *visible* since its location in the legacy system is known. For instance, PDPs of this kind can be specified with Sun's XACML. This allows the management of access control independently from the system. All requests pass through the PDP before getting to the system.

In order to illustrate explicit visible mechanisms, we consider that the PDP contains a `SecurityPolicyService` class which is called by the business services classes and returns the decision of the request (allow or deny). Figure 1 presents the point of view of the caller. The business services code of `borrowBook` automatically calls the visible PDP via the `SecurityPolicyService.check` method.

```
public  void borrowBook(User user, Book book)
throws SecuritPolicyViolationException {
// call to the security service
ServiceUtils.checkSecurity(user,
LibrarySecurityModel.BORROWBOOK_METHOD,
LibrarySecurityModel.BOOK_VIEW,
ContextManager.getTemporalContext());
// call to business objects
// borrow the book for the user
book.execute(Book.BORROW, user);
// call the dao class to update the DB
bookDAO.insertBorrow(userDTO, bookDTO);}
```

Figure 1. An explicit and visible security mechanism

2.3. Security policy testing

One aim of security policy testing is to ensure that security mechanisms are exercised in every way that may lead to a failure. A failure of a security mechanism occurs when an access is granted (resp. prohibited) while the security policy stipulates that it should be prohibited (resp. granted).

As an illustration, consider the top part of Figure 2 that displays the validation of a current legacy system against its security policy. Test cases are derived from the security policy SP (1) and the current version of the system is tested w.r.t the access rules.

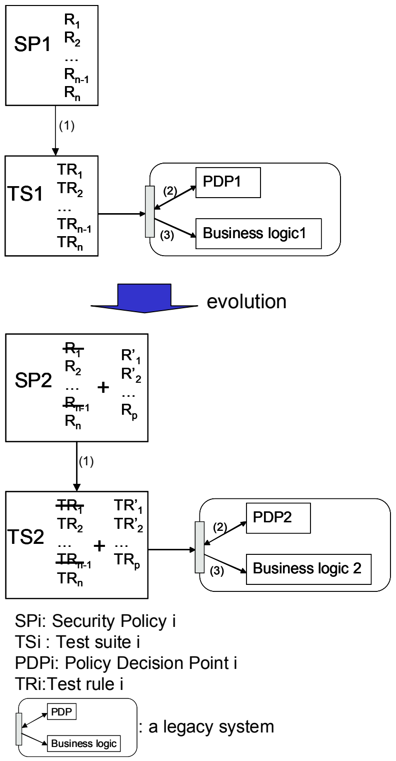

Figure 2. Regression testing of the legacy system

A test case is a sequence of method calls on the system that ends with a request that the PDP must check. To illustrate the notion of test case, let us consider the LMS: to be reserved, a book has first to be borrowed by another user. Any access control rule that forbids some users to reserve a specific category of books can only be tested after these books have been borrowed. The sequence is necessary to reach a state in which the rule to be tested can be exercised.

As shown in Figure 2, each test case is derived from a rule of a security policy (1), and calls the PDP (2) which, in turn, allows or forbids the access to the business logic part of the system (3).

As an example of a test case, consider testing that a secretary is not allowed to update personnel accounts. After preparing the test data, the respective service method is called. It takes two parameters: a personnel account object and a user object (the Secretary class extends the User class). The test fails if we do not get a security exception. Alternatively, the oracle could check that the action was not executed (check in the database that the account was not updated). A piece of Java code that tests this case is given below.

```
// preparing test data
Secretary sec1 = new Secretary();
PersonnelAccount pAccount =
new PersonnelAccount(accoutID);
// run security test
try {
PersonnelAccountSevice.updateAccount(pAccount,
                                         sec1);
// test oracle, we expect a security violation
exception here
Fail("test failed, security violation excep-
tion expected here, a secretary is not allowed
to update personnel accounts").
}
Catch(SecurityPolicyViolationException e) {
// pass: test successfully executed
}
```

In this example, the oracle is implemented using SecurityPolicyViolationException. Many other implementations are possible for the security outcome. Hence, the test can be automated or adapted to the specific implementation.

Formally, the PDP of an SUT, consisting of visible, implicit, and possibly hidden parts, implements a function s from requests, I, to decisions, O. That is, for any input vector $i \in I$, $s(i) \in O$ is the **actual** outcome for the request i.

A test case is a pair (i, p(i)) where i is a request that is drawn from a set I. p(i) is the expected outcome drawn from a set O={yes,no} with "yes" for permission and "no" for prohibition. $p(i) \in O$ is the **expected or intended** outcome as specified by the policy. p is the function that specifies the expected behavior, i.e., the oracle. For a given request i, a test case verdict is "pass" if s(i)=p(i), and it is "fail" otherwise.

2.4. Regression testing reveals some hidden and implicit mechanisms

We now turn our attention to the ideal case of dealing with a system's evolution using regression testing. It serves as an illustration of the problems that are caused by hidden and implicit security mechanisms.

If policies evolve, testing can be used to assess that the new security policy is correctly implemented. The bottom part of Figure 2 presents the ideal scheme of regression testing in cases where security policy evolves. The tests cases are first applied to the current system and then are partially reused on the new system, as explained in the following.

If only the access control policy evolves, the evolution is *iso-functional* since the functions of the business logic are unchanged. The evolution of a security policy consists of adding and removing access rules, and of adding and removing roles, activities and contexts. The new PDP is modified in accordance with this evolution. The correctness of the PDP modifications must be ensured using regression testing. Test cases corresponding to deleted rules are removed. For instance, in Figure 2, the TR1 test case is removed since rule R1 is deleted. The test cases remaining ones guarantee the non-regression, since they execute the unchanged parts of the security policy. New test cases are added for the new rules. In Figure 2, TR'i test cases are added for testing R'i new rules.

This ideal regression scheme rarely applies. Even for an iso-functional evolution, the unchanged business logic can be in contradiction with the new access control policy, due to implicit or hidden mechanisms.

The first case of mismatch occurs if an explicit security mechanism is not visible, i.e., is *hidden* in the legacy system.

Figure 3 adds a hidden mechanism to the example of Figure 1. In the body of the method, after this security call has been executed, a new check is done which forbids borrowing books during week-ends. Disabling the first mechanism will not prevent this hidden prohibition from being executed. If the PDP component is modified in order to allow borrowing books during week-ends, the hidden mechanism will have to be located and deleted.

```
public void borrowBook(Book b, User user) {
// visible mechanism, call to the security
policy service
SecurityPolicyService.check(user,
SecurityModel.BORROW_METHOD, Book.class, Se-
curityModel.DEFAULT_CONTEXT);
// do something else
// hidden mechanism
If(getDayOfWeek().equals("Sunday") || getDay-
OfWeek().equals("Saturday")) {
// this is not authorized throw a business
exception
Throw new BusinessException("Not allowed to
borrow in week-ends);}    …}
```

Figure 3. Explicit hidden security mechanism

A second kind of mismatch is due to implicit constraints which restrict the access, due to the way the system has been designed, implemented and deployed.

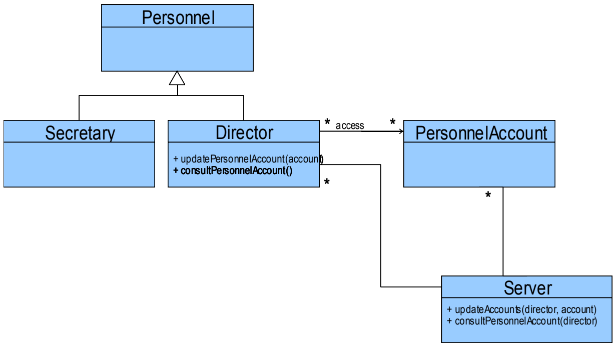
Personnel
Secretary
Director
+ updatePersonnelAccount(account)
+ consultPersonnelAccount()
*
access
*
PersonnelAccount
*
*
Server
+ updateAccounts(director, account)
+ consultPersonnelAccount(director)

Figure 4. Implicit security mechanism

By construction, a `Secretary` cannot directly call `updatePersonnelAccount`. A new access rule may specify that the director's secretary should now have access to personnel accounts. Since the program design does not have any direct reference from the secretary to the personnel accounts, we cannot add a rule "permission(secretary, update, account, always)". A simple refactoring would consist of moving the association "access" and the methods to the level of the class `Personnel`. Such a refactoring would also allow any Personnel instance to access the personnel accounts, which may be an unexpected change. So, the program will have to be carefully modified in several places to implement the desired evolution of the access control policy.

A last, more complicated case occurs when some access to a resource is granted by a hidden mechanism. In such a case, if the new security policy restricts this access, the visible PDP is modified but, depending on the execution flow, the user may still have his access granted due to the hidden mechanism.

In conclusion, the implementation of an evolving system policy is constrained by hidden and implicit security mechanisms. Even with a new PDP which implements the new security policy, these mechanisms may cause the system execution to fail. In this paper, we use testing for the early detection of such inconsistencies in the legacy system.

3. Flexibility analysis of the system to assess control policy changes

The difficulty of implementing an evolving policy is obviously related to the number of changes that must be applied to the current legacy system to enforce the new policy. Modifying the existing code may be more or less difficult, depending on the system size, on the programming languages (OO, Cobol etc.), and on the quality of documentation and design models. However, the mere modification of an existing application is costly because the legacy system business logic has to be analyzed.

In this paper, we argue that testing offers a pragmatic way of dealing with this problem and helps estimate the cost of a planned evolution. The test-driven evolution process consists of:

1. assessing the current legacy system,
2. measuring its flexibility w.r.t. (micro-) evolutions, and
3. diagnosing which functions and resources of the system are causing flexibility problems.

3.1. Exhaustive test-driven assessment of the current legacy system

The first question is to what extent the business logic implements hidden and implicit security mechanisms. These "hard-wired rules" may be redundant or even in conflict with some of the visible security mechanisms. The ratio of access control rules which are "hard-wired" in the business logic can be measured. We propose an exhaustive testing approach to assess the degree of hard-wiring of the security policy in the business logic. Exhaustive testing means that a test input is generated for each combination of roles, activities, views and contexts. In the top part of Figure 2, it means that step (1) is replaced by a systematic test input generation from the roles, activities, views and contexts. The system flexibility is thus measured independently from a given security policy but it depends on the roles, activities, views and contexts.

We start by disabling the visible security mechanisms of the legacy system under test. This can mean:

1. either that all requests should be granted now (universal permission),
2. or that all requests should be denied now (universal prohibition).

Recall that in contrast to function p which encodes the oracle, function s encodes the actual decisions taken by the PDP. Let s_+ denote the function that corresponds to the visible PDP that is disabled in the sense of universally **granting** access (i.e., **permission**). If there are no hidden or implicit mechanisms, we would expect

$$s_+(i)=\text{yes for all } i \in I.$$

Similarly, let s_- denote the function that corresponds to the visible PDP that is disabled in the sense of universally **prohibiting** access. If there are no hidden or implicit mechanisms, we would expect

$s_-(i)$=no for all $i \in I$.

1. Let us first apply an input i to the **original SUT** (no changes in the PDP). If s(i)=p(i), the PDP behaves as expected. If s(i)≠p(i), then something is wrong – the legacy system does not do what it is supposed to do, i.e., what is described in the policy. In our setting, we may assume that this does not happen. We assume that the current system is consistent with its actual security policy.
2. Let us now apply an input i to the SUT with the **visible PDP universally granting access**.
   a. If $s_+(i)$=yes, the result is inconclusive.
   Thus we cannot tell whether or not there is a hidden mechanism. Let *per1* be the number of test executions in this category.
   b. If, in contrast, $s_+(i)$=no, we have detected an implicit or hidden mechanism.
      i. If s(i)=no, we know for sure that disabling the visible PDP didn't have the desired effect. One scenario is that a request i1 is passed to the visible PDP and then forwarded to a hidden PDP which denies access. Disabling the visible PDP may mean that i1 is directly passed to the hidden PDP.
      ii. Symmetrically, let us assume that s(i)=yes. This means that disabling the visible PDP (in the sense of universally granting access) all of a sudden leads to a prohibition, a case that seems somewhat unlikely. However, one possibility is that in the original system, i1 is passed to the explicit PDP which transforms it into a distinct request i2 that is sent to the hidden PDP. The hidden PDP grants access to i2, and hence in sum, s(i1)=yes. Now, if the visible PDP is replaced by universal permission, this may mean that i1 is directly sent to the hidden PDP which denies access.
      Either way, a hidden mechanism is detected. Let *per2* be the number of test executions in this category.
   Note that this case is independent of the value of p(i). We don't need to know the actual policy.
3. Finally, let us apply an input i to the SUT with the **visible PDP universally prohibiting access**.
   a. If $s_-(i)$=yes, we have detected an implicit or hidden mechanism. In this situation, the visible PDP is somehow bypassed. Let *pro1* be the number of test executions in this category.
      i. If s(i)=yes, one scenario is that in the original system, a request i1 is first passed to the hidden mechanism which is directly granted, that is, control is transferred to the program logic without consulting the visible PDP. Modifying the visible PDP then obviously does not have any consequences.
      ii. If s(i)=no, one possible, albeit admittedly construed, scenario is that a request i1 is passed to the hidden PDP in the original system. Assume that this hidden PDP logs the reception and forwards i1 to the visible PDP who grants access. The positive response is passed back to the hidden PDP who decides that access cannot be granted. However, the hidden PDP may be configured in a way that a negative response of the explicit PDP (which is what we get by universal prohibition) is transformed into a positive one. This scenario appears rather unlikely but is not impossible.
   b. If, in contrast, $s_-(i)$=no, the result is inconclusive. Let *pro2* be the number of test executions in this category.
      i. There may not be a hidden mechanism.
      ii. On the other hand, there may well be a hidden mechanism whose functioning is masked by the universally prohibiting visible PDP.
   Note that this again is independent of the value of p(i).

In sum, the presence of hidden mechanism can be proven in those situations where $s_+(i)$=no and where $s_-(i)$=yes. Furthermore, the expected output p(i) does not matter in cases (2) and (3). The consequence is that the expected output part of a test case does not matter for our assessment. We "only" need to generate the input data for a test case but not the oracle. Furthermore, note that these results are independent of whether or not there is a default rule for policy evaluation, simply because the expected outcome p(i) does not matter.

In order to measure the flexibility of a system, let N be the number of exhaustive tests that is run. Clearly, N=per1+per2 = pro1+pro2.

The number of test cases that indicate the existence of hidden and implicit mechanisms is hence per2+ pro1. Conversely, the flexibility of a system is defined as the ratio

$$System\_flexibility = \frac{\text{per1} + \text{pro2}}{2\text{N}}.$$

A value of "1" means that our testing approach did not detect any hidden or implicit security mechanism (only inconclusive verdicts are emitted). A value of '0' means that every visible security mechanisms is doubled by a redundant mechanism which is hidden or implicit. When the ratio is close to zero, hidden and/or implicit security mechanisms are detected. These make the whole system rigid and its evolution problematic.

The problem with this approach to measuring system flexibility is that it forces all combinations of possible requests to be executed. It means, that a sequence must be produced for each possible input i. The number of test cases then amounts to

$$N = |RN| \times |PN| \times |VN| \times |CN|,$$

which may be huge, depending on the number of roles, activities, views and contexts.

3.2. Test Selection for assessing system flexibility

In order to overcome this problem, we now take into consideration the old and the new policies. The expert responsible for the security policy evolutions may want to know how much effort will be needed in terms of code and design refactoring. So only the hidden and implicit mechanisms that would block an evolution have to be detected.

a Test case criterion

In [8], we studied several test criteria to derive test cases from a security policy, including an analysis on the grounds of mutation analysis. Several mutation operators were proposed: a mutant differs from the initial code by the introduction of a single flaw into the security mechanism. The mutation score corresponds to the proportion of faulty versions of the system which are detected (or "killed") by the test cases. In the cited paper, an efficient criterion has been identified which consists of deriving at least one test case per concrete access control rule (some rule may be generic in the sense they apply to a category of roles or activities or context).

In the following, we use this criterion for our experiments. The approach could be easily adapted to other criteria. The underlying assumption is that we have a set of test cases which are able to exercise and test each access control rule.

b Decomposing policy evolutions

The evolution of a security policy can be decomposed into micro steps as presented in Figure 5:

1. $\delta+$ for relaxing a policy (addition of a permission or removal of a prohibition),
2. $\delta-$ for restricting a policy (addition of a prohibition or removal of a permission).

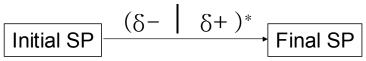

Figure 5. Decomposition of the SP evolution into micro steps

Let *SP* be a security policy, i.e., a set of permission and prohibition rules. Each micro-evolution results in a new security policy. We denote by SP^- the set of policies resulting from the restriction of the initial security policy (addition of a new prohibition, removal of a permission), and by SP^+ the security policies obtained by relaxing the initial policy (addition of a new permission, removal of a prohibition). We assume that a policy SP consists of two disjoint sets *Perm* and *Pro* that contain the permissions and prohibitions, respectively. Hence, $SP = Perm \cup Pro$. The set *NewPerm* contains all those permissions that can be added to the security policy. Similarly, *NewPro* contains all the prohibitions that can be added to the policy. Activities and views are not enumerated here for the reason that some activities cannot be used with every view (for example borrow can only be used with the book view). When adding a new rule we hence reuse activities with compatible views only.

$$NewPerm = \bigcup_{\substack{(\_,\_,a,v,\_)\in SP \\ r\in RN, c\in CN}} \{(permission, r, a, v, c)\} - SP$$

$$NewPro = \bigcup_{\substack{(\_,\_,a,v,\_)\in SP \\ r\in RN, c\in CN}} \{(prohibition, r, a, v, c)\} - SP$$

Activities and views are not enumerated here for the reason that activities are linked to specific views (for example borrow can only be used with the book view). When adding a new rule we reuse activities with compatible views. SP^- and SP^+ are then defined as follows:

$$SP^+ = \bigcup_{pro\in Pro} \{SP - \{pro\}\} \cup \bigcup_{nperm\in NewPerm} \{SP \cup \{nperm\}\}$$

$$SP^- = \bigcup_{perm\in Perm} \{SP - \{perm\}\} \cup \bigcup_{npro\in NewPro} \{SP \cup \{npro\}\}$$

c Measuring system's flexibility for a given SP

Based on a given security policy and on all possible evolutions, we are now ready to estimate the overall flexibility of a legacy system. By applying micro-evolutions, we also expect to get an idea of the functionalities and resources which are the subject of hidden or implicit access restrictions. The analysis of flexibility should help pinpointing those parts of the system which are flexible when the SP evolves.

Figure 6 depicts a test-driven technique to provide such an estimate. From an initial security policy

1. all possible security policy micro-evolutions are built,
2. for micro evolutions, the associated visible mechanisms are disabled,
3. for each micro-evolution, a test case is generated to test the evolution and applied to the business logic,
4. if the test case fails (detects an error), it means that the micro-evolution cannot be supported by the existing business logic without analysis

and possibly refactoring. The legacy system cannot easily evolve in that direction.

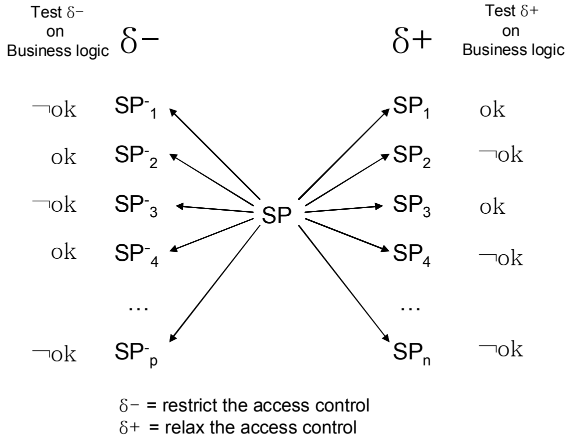

Figure 6. Micro-evolutions of a legacy system access control policy

Let TSP$^+$ (resp. TSP$^-$) denote the test cases set built for testing each δ+ (resp. δ−) micro-evolution. We denote by tsp(pass) a pass verdict for a test case tsp, the overall flexibility of the legacy system for a given security policy SP, tested with the test suite TSP, is equal to:

$$flexibility(SP,TSP) = \frac{\left|\{tsp \in TSP^+ / tsp(pass)\} \cup \{tsp \in TSP^- / tsp(pass)\}\right|}{\left|\bigcup_{i=1..n} SP_i^+\right| + \left|\bigcup_{j=1..p} SP_j^-\right|}$$

$$rigidity = 1 - flexibility$$

Example: Consider 20 test cases used to test the 20 possible micro-evolutions: If 5 of them are executed and detect a failure (either a prohibition when permission is expected or a permission when a prohibition is expected), the flexibility for micro-evolutions is equal to 0.75. One evolution in four cannot be done without analyzing the business logic.

3.3. Locating rigidity in the system

At each step, the analysis of micro-evolutions provides a useful diagnosis of those parts of the legacy system which are constrained by hidden or implicit security mechanism. The parts which cause the system to be "rigid" can be classified. The analysis we propose is two-fold.

1. *Access to resources analysis:* measuring the flexibility related to each resource (view) of the legacy system: detection of problematic resources.
2. *Access to functions analysis:* measuring the flexibility related to each function (activity) interacting with a resource.

First, the degree to which a given resource is "protected" by hidden or implicit mechanisms is obtained. This is an estimate of the flexibility related to a given resource. For a view v (which corresponds to the data and resources of the system), we define:

Resource_flexibility(v)= percentage of test cases which pass when testing a rule related to the view (resource) *v*. Let TSP*v* be the test cases set exercizing the rules related to a view *v*, we have:

$$resource\_flexibility(v) = \frac{\left|\{tsp \in TSPv / tsp(pass)\}\right|}{\left|TSPv\right|}$$

Second, the flexibility of system functions can be estimated in the same way. For an activity *a* (corresponding to a system's functions), we have:

Function_flexibility(a)= percentage of test cases which pass when testing a rule related to the activity (function) *a*. Let TSP*a* be the test cases set exercizing the rules related to an activity *a*, we have:

$$Function\_flexibility(a) = \frac{\left|\{tsp \in TSPa / tsp(pass)\}\right|}{\left|TSPa\right|}$$

3.4. Test-driven evolution in practice

The flexibility measurement we propose provides an analysis of the micro-evolutions that the business logic may accept without refactorings and modifications. In practice, it is not necessary to make a complete analysis of the current system flexibility when a new policy is defined. Let SPinit be the initial security policy, and SPtarg the target security policy. The evolution from SPinit to SPtarg can be decomposed into micro steps (maybe with many possible solutions)

$$\Delta(SPinit) = SPt\arg = \delta_n^{-+} \circ \delta_{n-1}^{-+} \circ \delta_{n-2}^{-+} \ldots . \delta_1^{-+}(SPinit)$$

where δ_i^{-+} denotes either a δ^- or a δ^+ micro-evolution

and $\circ$ a function composition.

The test driven process is similar to the test driven development principles promoted with XP. This test driven evolution technique involves repeatedly:

- writing a test case associated to the δ_i^{-+} micro-evolution;
- detecting a potential rigidity in the legacy system;
- fixing the problem to pass the test;
- then implementing – and documenting – only the micro-step in the PDP.

This pragmatic approach assists the safe evolution of a legacy system.

4. Experiments and discussion

We applied our technique to three examples:

- An auctions sales management system (ASMS) containing 10703 lines of code, 122 classes and 797 methods;

- a virtual meeting management system (VMMS) containing 6077 lines of code, 134 classes and 581 methods; and
- a library management system (LMS) containing 3204 lines of code, 62 classes and 335 methods.

The three case studies have a typical 3-tiers architecture widely used for web applications. We use the OrBAC [4, 5] environment as a specification language to define the access control rules.

4.1. VMS results

The virtual meeting system offers simplified web conference services. It is used in an advanced software engineering course at the University of Rennes. The virtual meeting server allows meetings to be organized on a distributed platform. When connected to the server, a user can enter or exit a meeting, ask to speak, speak, or plan new meetings. Each meeting has a manager. The manager is the person who has planned the meeting and has set its main parameters (such as its name, its agenda, etc). Each meeting may also have a moderator, appointed by the meeting manager. The moderator gives the floor to a participant who has asked to speak.

In the following table we present the results for the VMS. The overall flexibility of the initial policy is 0.35, which means that 35% of the security rules should be modifiable without code and design refactorings. It also reveals that the remaining is constrained by hidden/implicit security mechanisms.

	Flex. rules	**Rigid Rules**	**System flexibility**
results	20	36	0.35

To obtain a more accurate diagnosis, we consider which resources and views are the subjects of hidden/implicit mechanisms:

Resource\View	Flex. rules	Rigid rules	All	Flexibility
Meeting	12	36	48	0.25
PersonnelAccount	6	0	6	1
UserAccount	2	0	2	1

This table leads to the understanding that any evolution concerning `PersonnelAccount` or `UserAccount` is possible and that the problems of rigidity concern the `Meeting` resource. We can go further in the analysis and check which functions/activities are flexible in terms of security policy micro-evolutions. The results show –without surprise – that the functions which are more rigid are related to the `Meeting` manipulation. It also reveals that some of the functions related to this resource are flexible (like opening a meeting).

Function/Activity	**Flexibility**
updatePersonnelAccount	1
updateUserAccount	1
askToSpeak	0.13
leaveMeeting	0.14
overSpeaking	1
closeMeeting	1
setMeetingAgenda	0.14
setMeetingModerator	0.14
speakInMeeting	0.14
setMeetingTitle	0.14
deleteUserAccount	1
openMeeting	1
handover	1
deletePersonnelAccount	1

4.2. Auction Sales management system

The ASMS allows users to buy or sell items online. A seller can start an auction by submitting a description of the item he wants to sell and a minimum price (with a start date and an ending date for the auction). Then a typical bidding process starts, and people can bid on this auction. One specific feature of this system is that a buyer must have enough money in his account before bidding.

We obtained the following results for the ASMS. The system flexibility value is very close to the previous example and the diagnosis allows determining that only the `Comment` resource can support security evolutions without restriction. Concerning the other resources, `UserAccount` cannot evolve at all without modifying the code. Only 16% of the possible evolutions can be applied without problems to the Bid resource and 31% for `PersonnelAccount`. The activities/functions which are the cause of these rigidity problems can be located more precisely with the last table. For both VMMS and ASMS, we found implicit mechanisms similar to the one presented in Figure 4. Only $\delta+$ evolutions were problematic, which means that no hidden mechanism has been detected which grants permission while a prohibition is expected. No hidden mechanisms were detected, which may be explained by the fact the systems are new (not real "old" legacy systems), the business model designs are object-oriented, and that the PDPs are centralized in dedicated classes. In such cases, the main rigidity is caused by design constraints, i.e. implicit mechanisms, which only restrict the $\delta+$ evolutions (relaxing access).

Resource\View	Flex. rules	Rigid rules	All	Flexibility
Comment	5	0	5	1
Bid	1	5	6	0.16
PersonnelAccount	5	11	16	0.31
UserAccount	0	5	5	0

Function/Activity	**Flexibility**
updatePersonnelAccount	1

consultBid	0
consultComment	1
postComment	1
consultOldBids	0
updateBid	0
deleteBid	0.69
deleteComment	1
deleteUserAccount	0
consultPersonnelAccount	0
deletePersonnelAccount	0

	Flexible	Rigid rules	System flexibility
Results	12	22	0.34

4.3. LMS: Library Management System

The library management system is interesting since it is fully flexible. In fact, based on the return of experience of the two first studies, we built it to be flexible: the business model contains a `Role` class and allows the access to be fully controlled from the PDP. The principle for building a business model with no implicit mechanism may consists of making the access control concepts explicit (reification) in the business model.

5. Related work and conclusion

As far as we know, no previous studies focused on the problem of automatically identifying implicit/hidden access control mechanisms in legacy systems. Xie et al.[10] propose a machine learning algorithm to infer properties of XACML policies. This approach focuses on the PDP and infers the policy properties by analyzing request-response pairs. Their approach does not consider the whole system (PDP + system). As pointed out by [11], guiding the systems security policy evolution is a challenging issue. Our test-driven technique suggests that testing is a pragmatic technique to detect and locate (either using exhaustive or security policy based testing) hidden/implicit security mechanisms which might make the legacy evolution a nightmare for domain experts. The approach is still dependent on the generation of SP test cases. The issue of automating test generation can be addressed using combinatorial [12] or computational intelligence algorithms [13, 14]. Future work will focus on more complex evolution scenarios, including merging several organizations policies.

6. References

1. D. F. Ferraiolo, et al., *Proposed NIST standard for role-based access control.* ACM Transactions on Information and System Security, 2001. **4**(3): p. 224–274.
2. S. I. Gavrila and J.F. Barkley. *Formal Specification for Role Based Access Control User/Role and Role/Role Relationship Management*. in *Third ACM Workshop on Role-Based Access Control*. 1996.
3. R. Sandhu, E.J.C., H. L. Feinstein, and C.E. Youman, *Role-based access control models.* IEEE Computer, 1996. **29**(2): p. 38-47.
4. A. Abou El Kalam, et al., *Organization Based Access Control*, in *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*. 2003.
5. F. Cuppens, N. Cuppens-Boulahia, and M.B. Ghorbel. *High-level conflict management strategies in advanced access control models.* in *Workshop on Information and Computer Security (ICS'06)*. 2006.
6. Basin, D., J. Doser, and T. Lodderstedt. *Model driven security: From UML models to access control infrastructures*. in *ACM Transactions on Software Engineering and Methodology (TOSEM)*. 2006
7. T. Mouelhi, Y. Le Traon, and B. Baudry, *Mutation analysis for security tests qualification*, in *Mutation'07 workshop*. 2007.
8. T. Mouelhi, Y. Le Traon, and B. Baudry, *Testing security policies: going beyond functional testing*, in *International Symposium on Software Reliability Engineering*. 2007.
9. Briand, L. and Y. Labiche, *A UML-based approach to System Testing.* Software and Systems Modeling, 2002. **1**(1): p. 10 - 42.
10. Martin, E. and T. Xie. *Inferring Access-Control Policy Properties via Machine Learning*. in *7th IEEE Workshop on Policies for Distributed Systems and Networks* 2006.
11. Devanbu, P.T. and S. Stubblebine. *Software engineering for security: a roadmap*. in *International Conference on Software Engineering*. 2000.
12. A. Pretschner, T. Mouelhi, and Y. Le Traon, *Model-Based Tests for Access Control Policies*, in *International Conference on Software Testing Verification and Validation*. 2008.
13. B. Baudry, F. Fleurey, Y. Le Traon, and J.-M. Jézéquel. *Automatic Test Cases Optimization using a Bacteriological Adaptation Model: Application to .NET Components*. in *ASE'02 (Automated Software Engineering)*. 2002. Edimburgh, Scotland, UK: IEEE Computer Society Press, Los Alamitos, CA, USA.
14. B. Baudry, F. Fleurey, Y. Le Traon, and J.-M. Jézéquel., *From Genetic to Bacteriological Algorithms for Mutation-Based Testing.* Software Testing, Verification and Reliability, 2005. **15**(1): p. 73-96.