Mutating DAC And MAC Security Policies: A Generic Metamodel Based Approach

Tejeddine Mouelhi[1], Franck Fleurey[3], Benoit Baudry[2] and Yves Le Traon[1]
[1] IT- Telecom Bretagne, Rennes, France.
[2] INRIA/IRISA, Rennes, France.
[3] SINTEF, Oslo, Norway.

Abstract. In this paper we show how DAC and MAC security policies can be specified, implemented and validated through mutation testing using a generic approach. This work is based on a generic security framework originally designed to support RBAC and OrBAC security policies and their implementation in Java applications.

Keywords: Security, Model-driven engineering, Meta-modeling.

1 Introduction

Security is becoming a critical aspect of most software systems. Modern programming languages, coding guidelines and source code analysis techniques are able to detect and avoid low-level vulnerabilities such as buffer overflow and code injection. However, for higher-level vulnerabilities, related for example to access control, because the security code is sprayed across the application, code analysis cannot provide a complete solution. To deal with this issue, a number of security languages (such as DAC [1], MAC [2, 3], RBAC [4] or OrBAC [5]) have been defined. They allow the specification of security policies early in the development cycle. These specifications are used to verify the security policies and to generate a significant part of the security code of the final application. Systematic code generation avoids a large range of mistakes in the implementation of the security mechanisms. Unfortunately not all of the security code can be generated in such an automated way. For instance, the points in the application code where the security code has to be integrated can only be defined manually by the application developer. The security of the application thus needs to be validated after this step in order to check that the final implementation matches the initial security model. This is especially critical since when dealing with security, any minor defect is likely to compromise the security of the whole application.

In previous work [6], we propose a generic approach for modeling security policies and using mutation analysis in order to validate that the final implementation of a system conforms its security model. Mutation analysis consists of creating faulty versions of the reference application (called mutants). The efficiency of the security test cases is estimated with the percentage of seeded security faults they are able to detect. The security fault model is defined in terms of *mutation operators*, each of them specifying how to modify a security mechanism (access control rights for example). The approach is based on a generic security meta-model and on mutation operators defined for this meta-model. The meta-model is generic in the sense that it allows dealing with security policies expressed in different languages. The originality of a meta-modeling based approach is to have the same principles (captured by the meta-model and the associated fault model) to test security mechanisms, whatever the access control model is. Initially the approach was validated using RBAC and OrBAC security policies.

This paper shows how the approach can be extended to security policies expressed using DAC and MAC formalisms. The contribution of the paper is twofold. First it defines how DAC and MAC policies can be mapped to the generic security meta-model. And second it discusses which of the generic mutation operator have to be used in order to properly validate the final implementation of the security policies.

The paper is organized as follows. Section 2 summarizes the motivations and the approach presented in [6]. Section 3 presents the generic security meta-model. Section 4 details how MAC security policies can be represented and mutated. Section 5 details how DAC security policies can be represented and mutated.. Finally, section 7 discusses some related work and conclusion.

2 Context and Motivations

Figure 1 summarizes the approach presented in [6]. The two goals of this approach are to ensure quality by construction whenever possible and to provide systematic testing techniques for the rest. To ensure quality by construction the idea is to use a security modeling language in order to formalize security policies early in the development cycle and model-driven techniques to check the security policy and generate security code. For testing the final security code and its integration with the application code the idea is to use a security specific test criterion based on the mutation of the security model.

The first step of the approach (1) is to build the security model for the application. The security model is a platform independent model which captures the access control policies defined in the requirements of the system. This model is based on a generic meta-model which allows expressing any type of rule-based access-control policy. In practice, the meta-model allows modeling the type of rules to be used as well as the rules themselves. In previous work [6], we have shown how the generic meta-model can be used with RBAC and OrBAC policies and in this paper we detail how it can also support DAC and MAC security policies.

After the platform independent security model has been validated, automated transformations are used to produce platform specific security code such as the PDP – Policy Decision Point (2). A critical remaining step for implementing the security of the application is to connect the platform specific security code with the functional code of the application (3). To reduce the risk of mistake, we use AOP to make the security PEP introduction systematic (4).

In the proposed approach, the validation is done by testing the final running code with security specific test cases. To properly validate the security of the application, these test cases have to cover all security features of the application. The test criteria we use are based on the mutation of the security model. Mutation testing is a test qualification techniques introduced in [7] which has been recently adapted to security testing [8, 9].

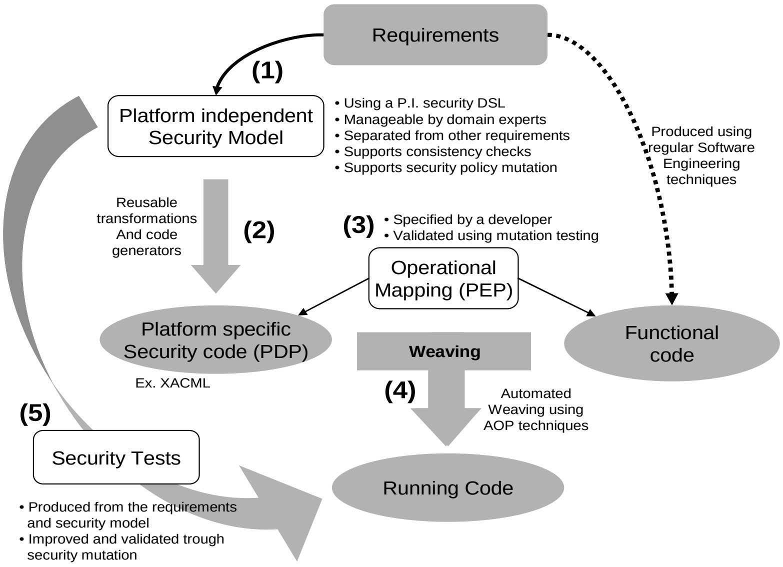
Requirements
(1)
Platform independent Security Model
• Using a P.I. security DSL
• Manageable by domain experts
• Separated from other requirements
• Supports consistency checks
• Supports security policy mutation
Produced using regular Software Engineering techniques
Reusable transformations And code generators
(2)
(3)
• Specified by a developer
• Validated using mutation testing
Operational Mapping (PEP)
Platform specific Security code (PDP)
Ex. XACML
Weaving
Functional code
(4)
Automated Weaving using AOP techniques
(5)
Security Tests
Running Code
• Produced from the requirements and security model
• Improved and validated trough security mutation

Figure 1 - Overview of the approach

The intuition behind mutation testing applied to security is that the security tests are qualified if there are able to detect any elementary modification in the security policy of the application (mutants). The originality of the proposed approach is to perform mutations on the platform independent security model using generic mutation operators. Since the transformation and weaving of the security policy in the application are fully automated, the tests can be automatically executed on the mutants of the application. If the tests are not able to catch a mutant then new test cases should be added to exercise the part of the security policy which has been modified to create this mutant. In practice the undetected mutants provide valuable information to create new tests and cover all the security policies.

In [6], a set of generic mutation operator are described and applied to RBAC and OrBAC security policies. In this paper we show how the same operators can be used to test security code based on DAC or MAC policies.

Overall, the main benefit of the approach is to allow validating the security policy using verification on the security model and testing that the policy implemented in the application conforms to the security model. Because the testing is performed on the final running code it allows validation both that the PDP is according to the model but also that the PEP, i.e. the integration with the rest of the application, is correct. The following sections detail the main steps of the approach.

3 A generic framework for security policies

The metamodel in Figure 2, displays the generic concepts for the definition of a security formalism and a security policy according to this formalism.

- The PolicyType, ElementType and RuleType classes are used to define a formalism. A PolicyType defines a set of element types (ElementType) and a set of rule types (RuleType).
- Based on a security formalism, it is possible to define a policy using the classes Policy, Rule and Parameter. A Policy is typed by a PolicyType. The type of a policy constrains the types of parameters and rules it can contain. If the *hierarchy* property of the parameter type is true, then the parameter can contain children of the same type as itself. Each rule has a type that belongs to the policy type and a set of parameters.

These two parts of the metamodel have to be instantiated sequentially: first define a formalism, then define a policies according to this formalism.

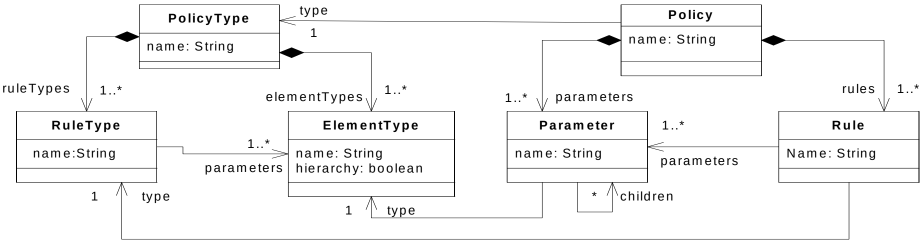

Figure 2 - The meta-model for rule-based security formalisms

3.1 Instantiating the metamodel

The two parts of the metamodel are instantiated at different moments. The classes that capture the concepts for a security formalism have to be instantiated first and define a modeling language that can be used to model a security policy for a particular system. The classes that capture the concepts to define a policy can only be instantiated if a formalism has been modeled.

3.2 Mutation testing for security

This section presents the fault models that we have defined at the meta-level and that can be executed to inject errors into security policies.

Mutation analysis involves qualifying a set of test cases for a program under test (PUT) according to the rate of injected errors they can detect. The assumption is that if test cases can detect errors that have been injected on purpose, they will be able to detect actual errors in the PUT. The validity of mutation analysis greatly depends on the relevance of faults that are injected. Faults are modeled as mutation operators that reflect typical faults that developers make in a particular language or domain. Several works (Xie et al. [8], Le Traon et al. [9]) have proposed mutation operators to validate test cases for security policy.

In this paper, we define five mutation operators for security policy testing, shown in Table 1. These operators are defined only in terms of the concepts present in the security metamodel, which means that they are independent of a specific security formalism. Thus, these operators can be applied to inject errors into any policy expressed with any formalism defined as an instance of our metamodel. The definition of mutation operators at this meta-level is critical for us since it allows the qualification of test cases with the same standard, whatever the formalism used to define the policy.

Table 1- The mutation operators

Operator Name	Definition
RTT	Rule type is replaced with another one
PPR	Replaces one rule parameter with a different one
ANR	Adds a new rule
RER	Removes an existing rule
PPD	Replaces a parameter with one of its descending parameters

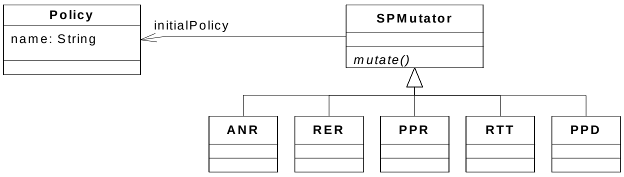

Figure 6. The mutation operator classes

Figure 6 shows the operator classes. The `mutate()` method is implemented in Kermeta. What is important to notice in this method is that it is defined only using concepts defined in the metamodel. Thus, this method can generate a set of mutated policies, completely independently of the formalism they are defined with. We apply it to MAC and DAC in the next section.

4 Applying the generic metamodel to DAC and MAC

This section is the core contribution of the paper. We present the DAC and MAC models and study how adapted is the meta-model to specify these two access control languages.

4.1 Generic metamodel applied to DAC

We first detail DAC (Discretional Access Control) main concepts and show how our metamodel can be used for modeling this language. Finally we present examples of mutants we obtain when applying our mutation operators.

a) Definition

The definition of DAC (according to [10]) :

“A means of restricting access to objects based on the identity of `subjects and/or groups to which they belong. The controls are ’discretionary’ in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on `to any other subject.”

A DAC policy expresses a set of Subjects and Objects and access types. A rule is the combination of one Subject, one object and one access type. In this paper, we consider DAC as used for file systems. Objects include files, directories or ports (or others) and Subjects include users or processes. The policy can be seen as a matrix where the values are access types. Access types include three access types (r: read, w: write and x: execute) and two special ones, which are *control* and *control with passing ability*. The control access type enables its holder to modify the users’ access types to that object. In addition to this, the control with passing ability enables the user to pass this control ability to other users.

The access types of the DAC:

- r : permission to read the object
- w: permission to write
- x: permission to execute
- c: control permission, the ability to modify ’r w x’ permission for this object.
- cp: control and the passing ability of control.

b) Modeling the DAC language

Figure 5 shows the DAC modeled using our generic metamodel of Figure 2. There is only one type of rule. This rule contains a Subject, an access type (r,w,x,c or cp) and a object (a file or a port etc.).

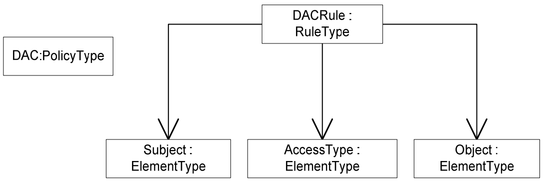

Figure 5 - The DAC formalism

c) Mutating DAC policies

To illustrate mutation results, we use a simple example of policy. The policy defines two subjects (Tim and Admin). Tim can read or execute file1, while admin has the right to read, write and execute the file in addition to the control and passing ability.

```
POLICY systemDAC (DAC)
R1 -> DACRule(Tim r file1)
R2 -> DACRule(Tim x file1)
R3 -> DACRule(Admin cp file1)
R4 -> DACRule(Admin r file1)
R5 -> DACRule(Admin w file1)
R6 -> DACRule(Admin x file1)
```

Some mutation operators cannot be applied to DAC policies. In fact, the RTT and PPD operator cannot be used since there is only one type of rule, and no hierarchy.

It is interesting to study the impact of the three mutation operators. For instance the RER operator will remove R1 resulting in a mutant policy that implies that Tim will no longer have the right to read “file1”. RER operator will produce 5 mutant policies, as there are 5 rules. The PPR operator will replace one of the rule parameter with a different one. One example of its mutant policies will be the one containing R1’:

DACRule(Tim w File1)

The mutant policy enables Tim to write in “File1” but denies him reading this file.
The ANR operator will produce mutants by adding one new rule to policy. One possible mutant is the one containing this new rule:

DACRule(Tim cp File1)

This will result in granting Tim the control and the passing ability.

4.2 Generic metamodel applied to MAC

We start with presenting MAC (Mandatory Access Control) and show how it can be modeled using our metamodel. We then study the mutants obtained based on the mutation operators defined at meta-level.

a) ***Definition***

In this paper, we consider MAC policies as they are used in multi-level systems (MLS) [11]. Next the definition of MAC (Taken from Trusted computer System Evaluation Criteria) :

“A means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (i.e., clearance) of subjects to access information of such sensitivity”.

MAC entities are Subjects, Objects and Clearances. Subjects are usually Processes or threads (executing user commands), and Objects can be files, ports, etc. MAC policies express the access of subjects to objects according to their clearance and to classification of objects. Subjects with a high clearance are able to read all kinds of objects. but they are not allowed to write in objects that can be accessed by low clearance subjects. A classification of clearance determines if the access is granted or denied. For example, if Subject S1 having clearance C1 requests reading Object O2 having clearance C2, then access if granted if C1 >= C2. Otherwise, if C2 > C1, access is denied.

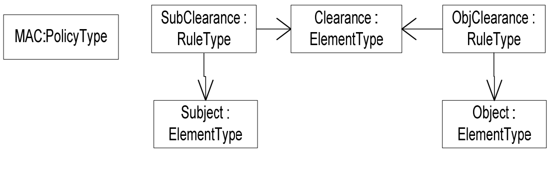

Figure 6 - The MAC formalism

b) ***Modeling MAC language***

Figure 6 displays the MAC metamodel which is also conformant to the Figure 2. There are two types of rules:

SubjClearance: An association between a subject and a clearance.

ObjClearance: An association between an Object and a clearance respectively.

There is a static classification of clearance. According to this classification, access is granted to subjects (to read or write). We consider two access types: read and write.

c) ***Mutating MAC***

We will use the following policy to show examples of mutants produced by the mutation operators. The policy defines two users and two objects and the rules specify their clearances.

```
POLICY systemMAC (MAC)
R1 -> SubClearance(process1 low)
R2 -> SubClearance (process2 high)
R3 -> ObjClearance (report1 low)
R4 -> ObjClearance (report2 high)
```

As for DAC policies, the mutation operators PPD and RTT cannot be applied in this case. The RER operator is not relevant either because it would create undefined policy responses. For instance, if R1 is removed, the subject ‘process1’ clearance will be unknown, resulting in undefined policy decision.

The relevant operators are PPR and ANR. The PPR operator will for example replace R1 second parameter with another one, which will produce this rule instead of R1: *R1’ -> SubClearance(process1 high)*

This implies process1 having a high clearance. This simulates a flaw in the security policy.

The ANR operator adds for example this new rule:

R5 -> SubClearance(report1 high)

This new rule is with conflict with the R3. So, the result depends on the implementation of the security mechanism, on the way it handles conflicts. If priority is given to most restrictive rule, then this implies report1 having high clearance.

4.3 Towards a unified validation framework for security policies

With these two examples of access control languages (DAC and MAC), we have shown that the metamodel we proposed is expressive enough to describe the most classical access control languages (DAC, MAC, RBAC, OrBAC). The interesting issue concerns the definition of a common validation scheme at metamodel level, which can be systematically applied, whatever the access control language is. It is especially useful when testing the security mechanisms in a heterogeneous environment, in which several systems with their own access control policies (expressed in different languages) interoperate. For example, this case occurs when Information Systems of several organizations are merged and when an overall security policy has to be built on the existing ones. More generally, the fact the mutation analysis allows faults models (mutation operators) to be described independently from the language is very promising. The intrinsic difficulty behind these approaches is related to the distance which separates the metamodel and the family of languages which can be modeled with it. In the case of access control policies, the family of languages manipulates a same subset of concepts. Metamodeling this family is thus feasible, and has been illustrated in this paper. It is less obvious to determine whether a common core of verification and validation can be defined at metamodel level in a relevant way. The problem is not new: you can metamodel everything but the semantics you can attach to the resulting metamodel may be too poor for relevant manipulations. In our case, the manipulations we attach are related to the validation of security policies, using mutation. The study presented in this paper shows that some mutation operators which are meaningful for RBAC and OrBAC access control policies cannot be instantiated for DAC or for MAC. The common definition of security faults in terms of mutation operators thus produces concrete faults which are very different from one access control language to another. To make the study complete, it would be necessary:

1- to model "equivalent" access control policies with OrBAC, RBAC, MAC and DAC languages,
2- to generate all the mutants versions for each policy
3- to compare whether a same test cases set is able to kill the same amount of mutants for each policy.

By applying such an empirical protocol, it will be possible to determine the quality of the faults which are seeded from a metamodel definition. In [12, 13], we already showed that the generated faults were relevant for OrBAC security policies. It has to be proven for the other languages. This study is the next step for empirically validating the metamodeling validation and verification environment we propose. It corresponds to the future work we will investigate. The empirical studies should allow concluding whether it is possible to obtain a unified validation environment for access control policies.

5 Conclusion

Guido Wimmel et al. proposed to use mutation to system specification in order to generate test suite for security-critical systems [14]. Faults are injected into the security requirement. Their approach does not handle well the scalability issue as it was not applied to large systems. In addition, Lodderstedt et al. [15] proposed SecureUML which provides a security modeling language to define the access control model. The resulting security model is combined with the UML business model in order to automatically produce the access control infrastructure.

This paper presented a step in the building of a unified framework for specifying, generating and validating a security policy. We studied how the metamodel we propose can be applied for the main access control languages. The metamodel has shown its ability to represent this family of languages. In parallel, we studied how the fault model attached to this metamodel could be used for applying mutation analysis on DAC and MAC access control policies. The first studies suggest that there are generic mutation operators that can apply to all security formalisms but also more specific operators that can still be expressed generically but apply only to a sub-set of access control formalisms.

6 References

1. B. Lampson. *Protection*. in *5th Princeton Symposium on Information Sciences and Systems,*. 1971.
2. K. J. Biba, *Integrity consideration for secure computer systems*, in *Tech. Rep. MTR-3153, The MITRE Corporation,*. 1975.
3. D. E. Bell and L. J. LaPadula, *Secure computer systems: Unified exposition and multics interpretation*, in *Tech. Rep. ESD-TR-73-306, The MITRE Corporation*. 1976.
4. D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli, *Proposed NIST standard for role-based access control.* ACM Transactions on Information and System Security, 2001. **4**(3): p. 224–274.
5. A. Abou El Kalam, et al., *Organization Based Access Control*, in *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*. 2003.
6. T. Mouelhi, F. Fleurey, B. Baudry, and Y. Le Traon, *A model-based framework for security policy specification, deployment and testing*, in *MODELS 2008*. 2008.
7. R. DeMillo, R. Lipton, and F. Sayward, *Hints on Test Data Selection : Help For The Practicing Programmer.* IEEE Computer, 1978. **11**(4): p. 34 - 41.
8. E. Martin and T. Xie. *A Fault Model and Mutation Testing of Access Control Policies*. in *Proceedings of the 16th International Conference on World Wide Web*. 2007.
9. T. Mouelhi, Y. Le Traon, and B. Baudry, *Mutation analysis for security tests qualification*, in *Mutation'07 : third workshop on mutation analysis in conjuction with TAIC-Part*. 2007.
10. C.S. Jordan., *A guide to understanding discretionary access control in trusted systems. Technical Report Library No.S-228, 576, National Computer Security Center (NCSC), Fort George G. Meade, Maryland, .* 1987.
11. D.E BELL and L.J. LaPADULA,
Secure Computer Systems: Unified Exposition and Multics Interpretation. 1976, The MITRE Corporation.
12. Y. Le Traon, T. Mouelhi, and B. Baudry, *Testing security policies : going beyond functional testing*, in *ISSRE'07 : The 18th IEEE International Symposium on Software Reliability Engineering*. 2007.
13. Y. Le Traon, T. Mouelhi, A. Pretschner, and B. Baudry, *Test-Driven Assessment of Access Control in Legacy Applications*, in *ICST 2008: First IEEE International Conference on Software, Testing, Verification and Validation*. 2008.
14. G. Wimmel and J. Jürjens. *Specification-based Test Generation for Security-Critical Systems Using Mutations*. in *ICFEM 2002.*
15. Torsten Lodderstedt, David Basin, and Jürgen Doser. *SecureUML: A UML-Based Modeling Language for Model-Driven Security*. in *Proceedings of the 5th International Conference on The Unified Modeling Language*. 2002.